Marco E. T. Gerards

Members of the graduation committee:

Prof. dr. ir.	G. J. M. Smit	University of Twente (promotor)
Dr. ir.	J. Kuper	University of Twente (assistant-promotor)
Prof. dr. ir.	B. R. H. M. Haverkort	University of Twente
Prof. dr.	J. L. Hurink	University of Twente
Prof. dr.	C. Witteveen	Delft University of Technology
Prof. dr.	B. Juurlink	Berlin University of Technology
Dr.	A. D. Pimentel	University of Amsterdam
Prof. dr.	P. M. G. Apers	University of Twente (chairman and secretary)

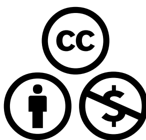
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics and Computer Science, Computer Architecture for Embedded Systems (CAES) group

CTIT

CTIT Ph.D. Thesis Series No. 14-314
Centre for Telematics and Information Technology
PO Box 217, 7500 AE Enschede, The Netherlands

The research in this thesis was supported by the Netherlands Organisation for Scientific Research (NWO) under project number 639.022.809.



Copyright © 2014 by Marco E. T. Gerards, Enschede, The Netherlands. This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/4.0/deed.en_US`.

This thesis was typeset using LaTeX, Ti*k*Z, and GNU Emacs. This thesis was printed by Gildeprint Drukkerijen, The Netherlands.

ISBN 978-90-365-3679-0
ISSN 1381-3617; Ph.D. Thesis Series No. 14-314
DOI 10.3990/1.9789036536790

Algorithmic Power Management

Energy Minimisation under Real-Time Constraints

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 18 juni 2014 om 14.45 uur

door

Marco Egbertus Theodorus Gerards

geboren op 22 september 1982
te Doetinchem

Dit proefschrift is goedgekeurd door:

Prof. dr. ir.	G. J. M. Smit	(promotor)
Dr. ir.	J. Kuper	(assistent promotor)

Copyright © 2014 Marco E. T. Gerards
ISBN 978-90-365-3679-0

Abstract

Energy consumption is a major concern for designers of embedded devices. Especially for battery operated systems (like many embedded systems), the energy consumption limits the time for which a device can be active, and the amount of processing that can take place. In this thesis we study how the energy consumption can be reduced for certain classes of real-time applications.

To minimise the energy consumption, we introduce several algorithms that combine power management techniques with scheduling (*algorithmic power management*). The power management techniques that we focus on are speed scaling and sleep modes. When the processor (or some peripheral) is active, its speed, and with it the supply voltage, can be decreased to reduce the power consumption (*speed scaling*), while when the processor is idle it can be put in a low power mode (*sleep modes*). The resulting problem is to determine a schedule, speeds for the processors (which may vary over time) and/or times when a device is put to sleep.

We discuss energy minimisation for three classes of real-time systems, namely (1) real-time systems with agreeable deadlines, (2) real-time systems with precedence constraints, and (3) frame-based real-time systems. In the subsequent paragraphs we elaborate on these classes of real-time systems.

(1) For real-time systems with agreeable deadlines it holds that an earlier arrival time implies an earlier deadline (and vice versa). Many telecommunication and multimedia applications can be modelled as tasks with agreeable deadlines. In Chapter 4 we present uniprocessor speed scaling techniques for such applications, where we use the fact that a lower speed results in a decreased power consumption. For energy efficiency, it is well-known that—due to the convexity of the power function—it is also important not to change the speed unnecessarily. All our algorithms use this fact to minimise the energy consumption. Furthermore, our algorithms take static power into account. We do not only avoid speed changes in the offline situation, where the workload of tasks is known before they are executed, but also in the online situation, where the workload is not known before execution

and only *predictions* and an upper bound of the workload is available. Compared to existing methods our algorithms can reduce the energy consumption by up to 54% for the considered multimedia workloads, and our evaluation shows that these algorithms are *near optimal* even with inaccurate predictions.

(2) The second class of real-time systems we focus on are tasks with *precedence constraints* that must be scheduled on a *multicore* system and for which the speeds have to be determined. To specify the optimal speeds for an application with a *given* schedule, the amount of parallelism must be taken into account. In case the system uses a relatively high speed at times when the parallelism is low, additional time is available to decrease the speed at times when the parallelism is high. Then more energy is used by only a few cores when the parallelism is low, while energy is saved by many cores when the parallelism is high. This may lead to a decreased total energy consumption.

In the literature a theoretical study of energy-efficient scheduling of tasks with precedence constraints on multicore systems is missing. In Chapter 5 we present an in-depth study of energy-aware scheduling for real-time systems with precedence constraints that are executed on a global speed scaling system (where all cores use the same speed simultaneously), with the aim to minimise the energy consumption under deadline constraints. The focus of this chapter is on the restricted problem where all tasks have a common arrival time and a common deadline (which is already NP-hard). To minimise the energy consumption, both scheduling and speed selection must be considered simultaneously. We derive a scheduling criterion that implicitly assigns speeds and minimises the energy consumption. In this context no new multicore scheduling algorithms are introduced because there are already many good existing algorithms. Instead, we present general techniques to relate the makespan (schedule length) criterion to the aforementioned scheduling criterion. A major insight is that a scheduling algorithm that minimises the makespan is energy optimal for two cores, while a counter example shows that this does not generally hold for more than two cores.

Furthermore, we present expressions for the optimal speeds of a given schedule and show that an energy reduction of up to 30% can be achieved with respect to state-of-the-art methods. We use these results in Chapter 6 to derive a technique to calculate the optimal speeds for the more general case, wherein each task has an individual arrival time and an individual deadline. This technique uses a substitution of variables to transform the global speed scaling multicore problem to the uniprocessor problem with agreeable deadlines. The previously developed algorithms for the uniprocessor problem with agreeable deadlines can then be used to solve the offline problem and to solve a restricted version of the online problem.

(3) In the third setting (Chapter 7), we study the optimal combination of speed scaling, sleep modes and scheduling for frame-based real-time systems. While the literature considers only trivial schedules for this problem, we study energy optimal schedules for such systems. Our scheduling algorithms create optimal idle periods in which devices can be put to sleep to minimise the energy consumption.

Furthermore, we prove that for frame-based real-time systems, scheduling first and then determining the speed scaling and sleep mode settings is optimal, and give algorithms that find these settings. Applying these algorithms can lead to energy savings of up to 50% compared to techniques from the literature.

Samenvatting

Het energieverbruik van een embedded systeem is erg belangrijk voor ontwerpers van dergelijke systemen. Hoelang en hoezeer een systeem dat op batterijen werkt (zoals vele embedded systemen) actief kan zijn, wordt met name beperkt door het energieverbruik. In dit proefschrift bestuderen we hoe we het energieverbruik voor bepaalde klassen realtime-systemen kunnen reduceren.

Om het energieverbruik te minimaliseren introduceren we een aantal algoritmen die energiebeheertechnieken combineren met het maken van een planning (*algorithmic power management*). De energiebeheertechnieken waar we ons op richten zijn snelheidsschaling en slaapmodi. Wanneer de processor (of een ander randapparaat) actief is, kunnen snelheid en voedingsspanning worden verlaagd om het vermogensverbruik terug te brengen (*snelheidsschaling* of *speed scaling*), terwijl een inactieve processor naar een slaapmodus geschakeld kan worden (*slaapmodi* of *sleep modes*). Dit leidt tot een probleem waarin zowel een planning, snelheden voor processoren (variërend over tijd) en/of tijden wanneer apparaten slapen moeten worden bepaald.

We bespreken energieminimalisatie van drie klassen van realtime-systemen, namelijk (1) realtime-systemen met *agreeable deadlines*, (2) realtime-systemen met *volgorde-eisen* en (3) *frame-based* realtime-systemen. In de onderstaande paragrafen gaan we in op deze klassen van realtime-systemen.

(1) Voor realtime-systemen met agreeable deadlines impliceert een eerdere aankomsttijd een eerdere deadline (en vice versa). Veel telecommunicatie- en multimediatoepassingen kunnen gemodelleerd worden als taken met agreeable deadlines. In hoofdstuk 4 presenteren we een aantal snelheidsschalingstechnieken voor dergelijke applicaties die op een enkele processor draaien, waarbij we gebruik maken van het feit dat een lagere snelheid resulteert in een lager vermogensverbruik. Het is welbekend dat voor energie-efficiëntie, vanwege convexiteit van de vermogensfunctie, het ook belangrijk is om onnodige snelheidsveranderingen te vermijden. Al onze algoritmen gebruiken dit feit om energieverbruik te minimaliseren. Tevens

houden onze algoritmen rekening met statisch vermogensverbruik. We vermijden niet alleen onnodige veranderingen van de snelheid in de *offline* situatie waarbij de werklast van taken bekend is voordat ze worden uitgevoerd, maar ook in de *online* situatie, waarin de werklast voor de uitvoering van een taak onbekend is en alleen *voorspellingen* en een bovengrens van de werklast beschikbaar zijn. In vergelijking met andere methoden en algoritmen zijn onze algoritmen in staat om het energieverbruik met 54% terug te brengen voor de beschouwde multimedia-werklast. De evaluatie laat zien dat onze algoritmen nagenoeg optimaal zijn, zelfs met onnauwkeurige voorspellingen.

(2) De tweede klasse van realtime-systemen waar we ons op richten, zijn taken met *volgorde-eisen* die ingepland moeten worden op een *multicore*-systeem en waarvoor de snelheden moeten worden bepaald. Om de optimale snelheden voor een applicatie met een *gegeven* planning te bepalen, moet de hoeveelheid parallellisme worden beschouwd. Wanneer het systeem op een relatief hoge snelheid loopt wanneer er weinig parallelle taken worden uitgevoerd, geeft dit extra tijd om de snelheid te verlagen wanneer er relatief veel parallelle taken worden uitgevoerd. In dat geval wordt er meer energie gebruikt door een paar cores gedurende de perioden waarin het parallellisme laag is, terwijl energie bespaard wordt door veel cores gedurende perioden waarin het parallellisme hoog is. Dit kan het totale energieverbruik terugbrengen.

Een theoretisch onderzoek naar het energie-efficiënt inplannen van taken met volgorde-eisen op multicore-systemen ontbreekt in de literatuur. In hoofdstuk 5 onderzoeken we het energiebewust plannen van realtime-systemen met volgorde-eisen die uitgevoerd worden op een processor waarvan alle cores gelijktijdig dezelfde snelheid gebruiken, met als doel om het energieverbruik gegeven deadlines te minimaliseren. Het hoofdstuk richt zich op het ingeperkte probleem waarbij alle taken een gezamenlijke aankomsttijd en een gezamenlijke deadline hebben (dit probleem is al NP-moeilijk). Om het energieverbruik te minimaliseren moeten zowel het planningsprobleem, als de snelheidskeuze, gelijktijdig beschouwd worden. We leiden een planningscriterium af dat impliciet optimale snelheden veronderstelt en energieverbruik minimaliseert. In deze context introduceren we geen nieuwe planningsalgoritmen, omdat er al vele goede algoritmen bestaan. In plaats daarvan presenteren we algemene technieken om het zogenaamde makespan-criterium aan het eerder genoemde planningscriterium te koppelen. Een belangrijk inzicht is dat een planningsalgoritme dat de makespan minimaliseert ook energie-optimaal is voor twéé cores, terwijl een tegenvoorbeeld aantoont dat dit niet in het algemeen geldt voor meer dan twee cores.

Verder geven we een karakterisatie van de optimale snelheden voor een gegeven planning en tonen we aan dat een energiereductie van 30% ten opzichte van de status quo mogelijk is. Deze resultaten gebruiken we in hoofdstuk 6 voor het afleiden van een techniek voor het bepalen van de optimale snelheden in het generieke geval, waarin elke taak een individuele aankomsttijd en deadline heeft. Deze techniek maakt gebruik van een substitutie van variabelen om het globale snelheidsschalings-

probleem te vertalen naar het enkeleprocessorprobleem met agreeable deadlines. Het eerder genoemde algoritme voor dit probleem met agreeable deadlines kan toegepast worden om zowel het offline probleem als het ingeperkte online probleem op te lossen.

(3) In de derde situatie, beschreven in hoofdstuk 7, bestuderen we de optimale combinatie van snelheidsschaling, slaapmodi en planning van frame-based realtime-systemen. Terwijl de literatuur slechts triviale planningen voor dit probleem beschouwt, bestuderen wij energie-optimale planningen voor dergelijke systemen. Onze planningsalgoritmen creëren optimale perioden van inactiviteit waarin apparaten in slaapmodus gezet kunnen worden om het energieverbruik laag te houden. Verder bewijzen we dat het voor frame-based realtime-systemen optimaal is om eerst een planning te bepalen en vervolgens de instellingen voor snelheidsschaling en slaapmodi. We geven algoritmen die deze instellingen vinden. Het toepassen van deze algoritmen kan een energiebesparing tot 50% opleveren in vergelijking met technieken uit de literatuur.

Dankwoord

Tijdens mijn afstuderen moest ik een keuze maken over wat ik daarna zou gaan doen. Gelukkig waren de mogelijkheden heel duidelijk voor mij. De eerste mogelijkheid was wiskunde studeren, omdat ik tijdens mijn studie informatica de wiskunde pas echt ben gaan waarderen. De tweede mogelijkheid werd aangeboden door Gerard; namelijk promoveren binnen zijn groep, waar ik met heel veel plezier aan het afstuderen was. Uiteindelijk was een keuze maken niet nodig en kwam ik samen met Gerard snel tot een oplossing: deeltijd promoveren (in vijf in plaats van vier jaar) en daarnaast een wiskundemaster volgen.

Gerard, bedankt voor de bijzondere mogelijkheid die je me hebt geboden, voor de vrijheid en het vertrouwen dat ik heb gekregen tijdens mijn promotie, de feedback en de discussies over het onderzoek. Minstens zo belangrijk: bedankt dat je er voor zorgt dat er een gezellige ontspannen sfeer heerst binnen de leerstoel. Dit is niet iets dat vanzelfsprekend geldt voor elke leerstoel. Jan Willem en Anton wil ik ervoor bedanken dat ze het mogelijk hebben gemaakt om wiskunde te studeren naast mijn promotie.

Tijdens mijn afstuderen bij informatica heb ik Jan leren kennen. Hij werd daarna de dagelijkse begeleider voor mijn promotie. Ik wil hem bedanken voor de feedback die hij heeft geleverd, de discussies en dat hij altijd tijd maakt om over persoonlijke dingen te praten. Door Jan ben ik gaan inzien dat je met inhoud alleen er niet komt en de presentatie misschien wel nog belangrijker is om een paper geaccepteerd te krijgen; een wijze les waar ik nog lang iets aan zal hebben.

In een laat stadium van mijn promotie was ik bezig met de combinatie van scheduling en optimalisatie. Tijdens een zoektocht naar een expert in beide gebieden kwam ik snel bij Johann uit. Johann maakt altijd veel tijd vrij, zowel om teksten van commentaar te voorzien, als voor goede, diepgaande discussies. Ik ben hem erg dankbaar voor de goede en intensieve samenwerking.

Ook Philip raakte in een laat stadium bij mijn promotie betrokken. Tijdens het laatste jaar van mijn promotie hebben we veel discussies gehad en heeft hij veel

van mijn teksten van commentaar voorzien. Ik wil Philip bedanken voor de fijne samenwerking. Verder wil ik hem bedanken voor vele praktische schrijftips, die mij in de toekomst ongetwijfeld nog blijven helpen.

Verder wil ik mijn vele collega's bedanken voor de fijne samenwerking. Allereerst mijn kamergenoten Jochem en Arjan voor een goede sfeer en vele diepgaande discussies; mijn deeltijdkamergenoten Robert en Koen voor de vele interessante discussies en hun gewaagde uitspraken die tot taart hebben geleid; Bert voor de prettige samenwerking op onderwijsgebied, ik blijf het erg leuk vinden om de practica DDS (of heet het ODS?) en BBDT te begeleiden; vele (oud)collega's (waaronder Pascal, Philip, Albert, Vincent, Maurice en Jochem) voor het ontwikkelen van een LaTeX-template waar ik dankbaar gebruik van heb gemaakt om dit proefschrift vorm te geven; Hermen voor het vinden van vele typfouten; de vele collega's – te veel om hier bij naam te noemen – die ooit conceptpapers hebben gelezen en van commentaar hebben voorzien; Marlous, Thelma en Nicole voor de geweldige ondersteuning; verder al mijn collega's voor de gezelligheid en leuke discussies tijdens onder andere (thee!)pauzes, lunchwandelingen en borrels.

Alexander, Ronald en Almer wil ik bedanken voor de gezelligheid en welkome afleiding van mijn werk. Het is jammer dat ik ze het afgelopen jaar minder zag, omdat ik zo druk was. Ik ben ze dankbaar voor hun begrip hiervoor. Almer, bedankt voor het commentaar op mijn proefschrift en dat je mijn paranimf wilt zijn.

Mijn ouders en broertje (en tevens paranimf) Rob wil ik bedanken voor alle steun.

Als laatste wil ik de voor mij allerbelangrijkste persoon bedanken: Ellen. Afgelopen jaar was ik druk, waardoor ik minder tijd had voor de dingen die voor jou belangrijk zijn. Bedankt voor jouw geduld, hulp en steun tijdens mijn promotie.

Marco
Enschede, mei 2014

Contents

1 **Introduction** 1
1.1 Real-time streaming applications . . . 2
1.2 Speed scaling . . . 2
1.2.1 Globally optimal speed scaling . . . 3
1.2.2 Speed scaling and multiprocessor scheduling . . . 4
1.3 Sleep modes . . . 5
1.4 Problem statement . . . 7
1.5 Claims and contributions . . . 8
1.6 Structure of this thesis . . . 9

2 **Background** 11
2.1 Introduction . . . 11
2.2 Tasks . . . 11
2.2.1 Notation . . . 12
2.2.2 Types of aperiodic real-time systems . . . 12
2.3 Speed scaling . . . 13
2.3.1 Processor models . . . 13
2.3.2 Speed scaling notation . . . 16
2.4 Sleep modes . . . 16
2.5 Problem notation . . . 18
2.6 Theoretical results . . . 19
2.6.1 Constant speed . . . 20
2.6.2 Nonconvex power function . . . 21
2.6.3 Critical speed . . . 21
2.6.4 Discrete speed scaling as a linear program . . . 22

2.6.5 *Relation between continuous and discrete speed scaling* 23
2.6.6 *Power equality* 23
2.6.7 *Nonuniform power* 24
2.6.8 *Flow problems* 25
2.7 Conclusions 25

3 **Related Work** **27**
3.1 Introduction 27
3.2 Uniprocessor problems 27
3.2.1 *General tasks* 28
3.2.2 *Agreeable deadlines* 32
3.2.3 *Laminar instances* 33
3.3 Multiprocessor problems 33
3.3.1 *General tasks* 34
3.3.2 *Agreeable deadlines* 35
3.4 Online uniprocessor speed scaling algorithms 35
3.5 Global speed scaling of tasks with precedence constraints 36
3.6 Frame-based real-time systems 37
3.7 Conclusions 38

4 **Uniprocessor Speed Scaling** **39**
4.1 Introduction 39
4.2 Modelling assumptions 41
4.2.1 *Model* 41
4.2.2 *Discussion on simplifications* 42
4.3 Offline optimisation 43
4.3.1 *Fixed static energy* 43
4.3.2 *Variable static energy* 45
4.4 Online speed scaling 50
4.4.1 *RA-SS* 50
4.4.2 *PRA-SS* 52
4.5 Evaluation 54
4.5.1 *Application for evaluation* 54
4.5.2 *Greedy algorithms* 56
4.5.3 *Evaluation of online algorithms* 57
4.5.4 *Simplifying assumptions* 60
4.6 Conclusions 61

5 **Scheduling for Global Speed Scaling** 63
5.1 Introduction . . . 63
5.2 Model . . . 66
5.2.1 Application model . . . 66
5.2.2 Power model . . . 67
5.2.3 Parallelism based model . . . 68
5.3 Optimal speeds . . . 70
5.4 Scheduling and speed scaling . . . 73
5.4.1 Scheduling criterion . . . 73
5.4.2 Using the makespan . . . 75
5.4.3 Two cores . . . 82
5.5 Evaluation . . . 83
5.5.1 Analytic evaluation . . . 83
5.5.2 Simulations . . . 84
5.6 Conclusions . . . 86

6 **Speed Selection for Global Speed Scaling** 87
6.1 Introduction . . . 87
6.2 Pieces . . . 88
6.3 Optimisation model . . . 91
6.4 Online speed scaling . . . 94
6.5 Conclusions . . . 95

7 **Sleep Modes and Speed Scaling for Frame-Based Systems** 97
7.1 Introduction . . . 97
7.2 System model and notation . . . 99
7.2.1 Application model . . . 99
7.2.2 Sleep modes . . . 100
7.2.3 Speed scaling . . . 102
7.3 Sleep modes . . . 103
7.3.1 Properties of optimal sleep modes . . . 103
7.3.2 Non-variable Work . . . 105
7.3.3 Variable Work . . . 107
7.4 Speed scaling . . . 111
7.4.1 Non-variable work . . . 111
7.4.2 Optimal continuous speed scaling . . . 114
7.4.3 Optimal discrete speed scaling . . . 117
7.4.4 Variable work . . . 119
7.5 Evaluation . . . 119
7.6 Conclusions . . . 121

8 Conclusions and Recommendations 123
8.1 Summary 123
8.2 Conclusions 125
8.3 Recommendations for future research 127
8.3.1 Online global speed scaling 127
8.3.2 Local speed scaling for tasks with precedence constraints 127
8.3.3 Measurements on real systems 129
8.3.4 Influence of shared resources 129

A Mathematical Background 131
A.1 Convex optimisation 131
A.1.1 Convex sets 131
A.1.2 Convex functions 131
A.1.3 Convex optimisation 133
A.2 Heuristic algorithms 134
A.3 List scheduling 134

B Problem Notation 137

Acronyms 139

Nomenclature 141

Bibliography 145

List of Publications 155

Index 157

Chapter 1

Introduction

Reducing the energy consumption of computing devices is of major importance, in particular for computers in data centers and embedded systems. In 2006, 7.3% of the total Dutch energy consumption was due to Information and Communications Technology (ICT) equipment and ICT related services [28], and this total energy consumption is still increasing. For embedded systems, energy imposes major design restrictions. The energy that is available for battery operated embedded systems is limited and for many devices, like smartphones, it does not increase at the same pace as their energy consumption. Smartphone users are faced with this development, and as a result many users charge their smartphone daily [77].

To deal with these problems caused by the increasing energy consumption, we propose techniques to lower the energy consumption of computing devices *without* reducing the quality of service. This quality of service depends on whether software tasks meet their (strict) deadlines. Software is capable of changing hardware settings like the speed of a device, or putting the device in a low power sleep mode. With these settings, software can provide a trade-off between time and energy. Since computing devices are often highly overdimensioned, and the deadlines are met by a wide margin, large energy reductions are possible by adapting the speed of the devices and using sleep modes. This is the topic of this thesis:

Methods for energy minimisation of computing devices under real-time constraints.

Since it is impossible to cover all possible applications, we restrict ourselves to a subset of applications (Section 1.1). We apply power management techniques to minimise the energy consumption that is due to the execution of these applications. For this thesis, the two most relevant power management techniques are speed scaling, implemented as Dynamic Voltage and Frequency Scaling (DVFS) see Section 1.2, and sleep modes, implemented as Dynamic Power Management (DPM) see Section 1.3. The problem statement of this thesis is discussed in Section 1.4 and the contributions are listed in Section 1.5.

1.1 Real-time streaming applications

A large number of the applications for embedded systems have very specific streaming characteristics. Many of these applications are Digital Signal Processing (DSP) applications, for example, audio and video decoding/encoding, communication, RADAR and GPS. These applications have in common that a stream of data enters the system, is processed, and the result appears as a stream of output data.

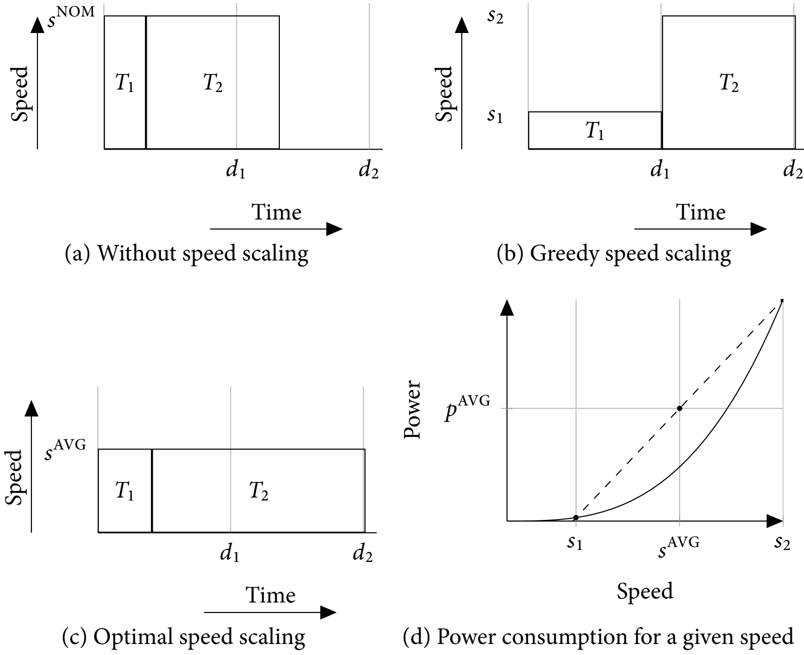
A streaming application typically consists of tasks, which are relatively small portions of computation that together produce the desired result. Many streaming applications can be modelled using a Directed Acyclic Graph (DAG). In this graph, tasks represent vertices (nodes) in the graph, while the precedence (or ordering) constraints of the tasks are described using edges. The data is streamed through this graph: vertices with incoming edges receive data and vertices with outgoing edges produce the results.

Many streaming applications are *real-time* applications, which means that the tasks have deadlines. There are many types of real-time applications, for example, hard real-time, firm real-time and soft real-time applications. Missing a deadline in a hard real-time system leads to a (possibly catastrophic) system failure, in a firm real-time application a late result is useless but not catastrophic, while in a soft real-time system a late result is less useful.

A good example of a firm real-time streaming application is a video decoder. In the context of a video decoding application, a task can be the processing of a frame and the deadlines can be the display times of the video frames. When a deadline is missed in a video application, this may result in dropped frames, since some frames are not decoded before the intended display time. Hence, the result (the decoded frame) is useless, making the application firm real-time. Similarly, a missed deadline in an audio application may result in distortion.

1.2 Speed scaling

The speed (operations per second) of many devices can be decreased to lower the power consumption. This technique is called *speed scaling*. Usually speed scaling results in a decreased energy consumption, despite the fact that the power is consumed for a longer time[1]. A popular speed scaling technique that is used in modern microprocessors is DVFS. DVFS is used to decrease the clock frequency (and with it, the voltage), leading to a reduced speed and power consumption. Speed scaling is also used in other devices, such as flash storage, hard drives, and network cards [55, 74].



(a) Without speed scaling (b) Greedy speed scaling (c) Optimal speed scaling (d) Power consumption for a given speed

Figure 1.1 – Three alternative speed scaling settings.

1.2.1 Globally optimal speed scaling

Many existing speed scaling approaches are of a greedy nature, and determine the lowest speed for each individual task in an attempt to minimise the energy consumption for that task. Take for example the application consisting of two tasks (T_1, T_2) that is depicted in Figure 1.1a. As both tasks finish well before their deadline (at respectively d_1, d_2) at the nominal speed (s^{NOM}), speed scaling can be deployed to reduce the energy consumption. This figure gives the time (horizontal axis), speed (vertical axis) and the amount of work of a task (time×speed, i.e. the area of the task in the figure). Figure 1.1b shows the greedy approach to speed scaling, whereby the lowest allowed speed for task T_1 is chosen such that the deadline d_1 is just met, and similarly for task T_2. However, from this figure, the impact of greedy speed scaling on the energy consumption is not immediately clear. For this, we need to know the power consumption at each speed. For illustration purposes, we use a cubic relation between the speed and the power consumption (energy per unit time), as is depicted in Figure 1.1d. For an application that consists of two tasks, this figure also shows the *average power* consumption p^{AVG} (the dot on the dashed line) at the *average speed* (s^{AVG}) of the application. The cubic power curve is significantly below the average power, and therefore executing both tasks at the

[1]Energy is power multiplied by time.

average speed reduces the power consumption significantly. As a consequence, the energy consumption is also significantly reduced. This new speed assignment is feasible because both deadlines are still met (Figure 1.1c). Since the energy savings can be tremendous, this result is emphasised by the following proposition.

Proposition 1.1 (Average speed). *The energy consumption of a single processor with a convex power function* never *decreases when the average speed is used.*

In Chapter 2, we show that this proposition generally holds.

1.2.2 Speed scaling and multiprocessor scheduling

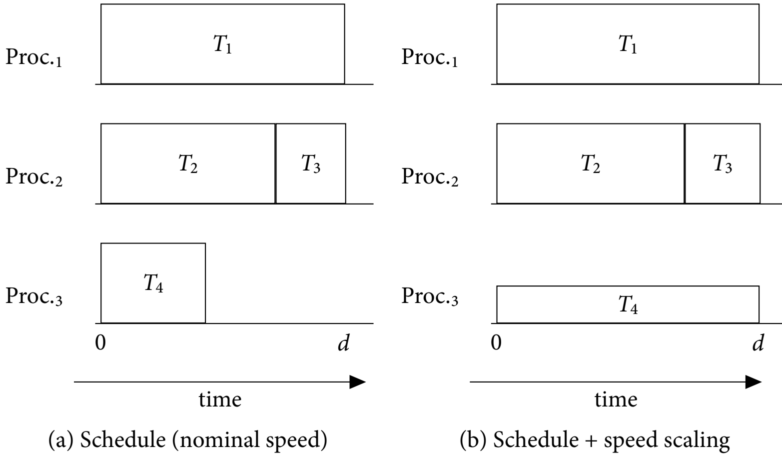
In multicore situations, especially when precedence constraints are involved, optimal speed scaling becomes a nontrivial problem. Instead of using a single (average) speed for all tasks, it is worthwhile to increase the speed during the time period a single processor is active, and decrease it during the periods where multiple processors are in use. In this situation, slightly more energy is consumed when a single core is active, while the energy consumption decreases when multiple cores are active. This can lead to a reduction of the total energy consumption (see Section 2.6.6).

However, not only the selection of speeds is relevant when minimising the energy consumption: also the schedule has a great influence. To illustrate this, consider a simple application with three processors where four tasks arrive at time 0, have a common deadline at time d and are executed without interruptions. In this case, only the assignment of tasks to processors is relevant, while the order in which tasks are executed does not influence the energy consumption. When speed scaling is applied to the schedule from Figure 1.2a, each processor receives the lowest speed that ensures that their tasks meet their common deadline. This is depicted by Figure 1.2b. Note, that the first and second processor use a relatively high speed, while the third processor uses a relatively low speed. In this situation it is impossible to use an average speed (over all processors), but it is possible to balance the workload, and then use speeds closer to the average speed. This is done in the schedule that is shown in Figure 1.3a, with the corresponding speed scaling as shown in Figure 1.3b. This new schedule and speed assignment requires less energy than our first attempt, because the speed deviation from the average is smaller.

In general, first scheduling to minimise the execution time, and then assigning the speeds is suboptimal (and vice versa). This leads to the following proposition, that is thoroughly discussed in Chapter 5.

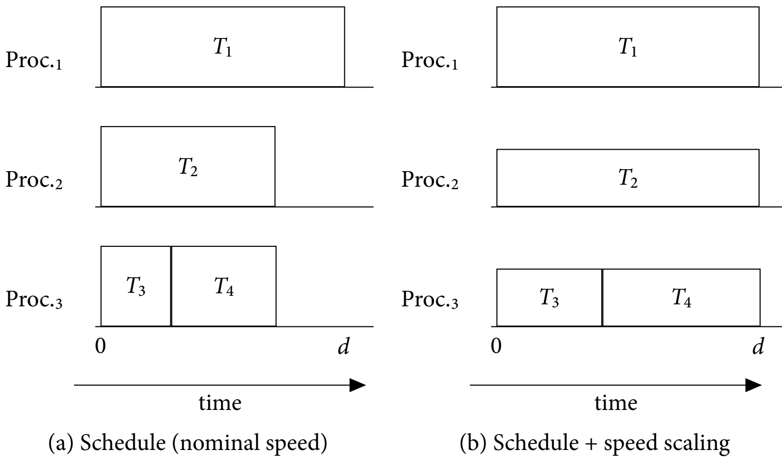
Proposition 1.2. *Generally, for optimal results, scheduling and speed scaling should be considered simultaneously.*

In many cases the optimal combination of scheduling and speed scaling is NP-hard. This follows from the fact that multiprocessor scheduling, which is already NP-hard, is a special case of the general combined speed scaling and scheduling problem of tasks with a common deadline.



(a) Schedule (nominal speed) (b) Schedule + speed scaling

Figure 1.2 – Schedule with speed scaling.



(a) Schedule (nominal speed) (b) Schedule + speed scaling

Figure 1.3 – Alternative schedule with speed scaling.

1.3 Sleep modes

Besides speed scaling, many devices support switching to a low power *sleep mode* to reduce the energy consumption. Whereas speed scaling reduces the energy consumption when the device is *active*, switching to a sleep mode reduces the energy consumption when the device is *idle*. A device may have multiple sleep modes, where higher energy savings and higher transition latencies[2] are associated

[2]Transition latency: time required to transition to a sleep mode and back.

Table 1.1 – Task characteristics.

Task	Amount of work	Arrival time	Deadline
T_1	3	0	29
T_2	4	0	29
T_3	7	0	29
T_4	2	0	29
T_5	2	10	12

with deeper sleep modes. Both speed scaling and sleep modes can be combined to attain a further energy reduction. Typically, it is harder to minimise the energy consumption using sleep modes than with speed scaling.

When a device is idle, it may be put to sleep if the energy reduction of the idle period outweighs the energy costs for the transition to the sleep mode and back. The length of the time interval for which switching to a sleep mode becomes sensible is called the *break-even time*. In general, not only the break-even time, but also the schedule influences the effectiveness of power management with sleep modes.

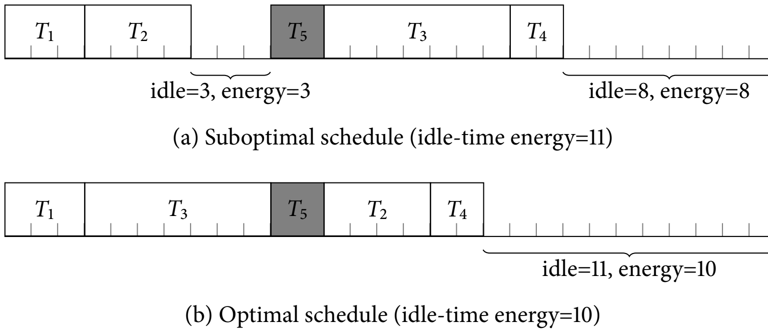
The following example illustrates the complexity of the scheduling trade-offs. Consider five tasks with the characteristics given in Table 1.1. We require that tasks may not be interrupted after they have started their execution, i.e. preemptions are not allowed. The tasks are to be scheduled on a single processor, and we assume that the processor is active before time 0 and after time 29. The processor has a power consumption of 1 when idle, 0 when asleep (i.e. in this example we assume a sleeping processor consumes no power) and has a break-even time of 10 time units. The active power during the execution of the tasks is ignored, since it cannot be influenced in the context of this example.

Because task T_5 must be scheduled at time 10, each other task is either executed before or after task T_5. An example of a schedule is given by Figure 1.4a. Both the idle periods in this schedule are shorter than the break-even time, therefore sleep modes cannot be used to reduce the energy consumption for this schedule.

The unique optimal schedule (modulo task ordering) is shown in Figure 1.4b. The difficulty of obtaining this optimal schedule is that the set of tasks without task T_5 has to be partitioned in sets with a total execution time of respectively 10 and 6, because only this partition creates an idle period longer than the break-even time. In general, the problem is NP-hard, as the subset sum problem can be reduced to it. The above example informally illustrates the basic idea of the reduction.

The example shows that, as with speed scaling, scheduling plays a fundamental role when sleep modes are used, as is stated by the following proposition.

Proposition 1.3. *Generally, for optimal results, scheduling and sleep modes should be considered simultaneously.*



(a) Suboptimal schedule (idle-time energy=11)

(b) Optimal schedule (idle-time energy=10)

Figure 1.4 – Schedules for speed scaling.

1.4 Problem statement

The problem studied in this thesis is energy minimisation under time constraints, whereby algorithms are used to determine the optimal power management settings. This approach of using optimal algorithms or approximation algorithms to determine a schedule together with power management settings is commonly referred to as *algorithmic power management*, which explains the title of this thesis.

Since the schedule influences to which extend power management techniques can be effectively used, energy efficient scheduling is researched in this thesis. According to the survey by Chen and Kuo [25] "*…energy-efficient scheduling for jobs with precedence constraints with theoretical analysis is still missed in multiprocessor systems*". There are several variants of speed scaling for multiprocessor systems, where processors (i) receive the same speed (global speed scaling), (ii) receive an individual speed (local speed scaling) and (iii) are clustered in groups ("islands") that each receive a common speed. Global speed scaling (in the form of global DVFS) is commonly used by modern microprocessors and systems such as the Intel Itanium, the PandaBoard (dual-core ARM Cortex A9), IBM Power7 and the NVIDIA Tegra 2 [50, 51, 66]. Implementing the global DVFS hardware in a processor is less complex and less expensive than implementing local DVFS [24, 66], which explains why global DVFS occurs more often in practice. This research focuses on *global* speed scaling, because it is often used in practise and is not yet widely researched in the algorithm oriented literature.

The following research questions are studied in this thesis:

» What are the optimal speeds for global speed scaling?
» What characterises the energy minimising schedule?
» How well do existing scheduling algorithms minimise the energy consumption?

Interestingly, even for speed scaling with a single device, not all important problems are solved in the literature. For example, no algorithm takes static power into

account properly, and no practical algorithm comes close to minimising the energy consumption in the online situation where the exact amount of work is unknown before a task is executed and only a prediction of this amount of work is available. As it is worthwhile to first solve the uniprocessor case before dealing with the multiprocessor problem, in this thesis, the following research questions for single devices are studied:

» What are the optimal speeds when static power is present?

» How to choose the optimal speeds online when only (possibly inaccurate) predictions of the amounts of work for tasks are available?

» How can speed scaling be combined with sleep modes?

In most cases, we restrict ourselves to tasks with agreeable deadlines[3] and frame-based real-time systems.

1.5 Claims and contributions

The research that is described in this thesis is mostly theory oriented, and based on commonly accepted models. It explores globally optimal power management, and presents efficient algorithms together with proofs of optimality. As an introduction to this theoretical field, we give an overview of the models and related theory (Chapter 2), and present—besides an overview of directly related research—an extensive survey of existing offline energy minimisation algorithms (Chapter 3).

Many papers (somehow) predict the amount of work of a task, and use this prediction to set the speed of the task greedily, such that it minimises the energy consumption for this task. We show that this approach consumes much more energy than what can be theoretically obtained when considering all tasks globally. With the offline solution—which knows all future workload—in mind, we derive an algorithm called RA-SS that uses the predictions to obtain the speeds that guarantee deadlines are met while the energy is minimised (Chapter 4). We evaluate a variant of this algorithm (with constant time complexity) that does not require any predictions of the amount of work of tasks, but only requires a prediction of the *average* amount of work. Furthermore, it keeps the number of speed changes to a minimum, while keeping the speeds as low as possible.

We evaluate this algorithm with an MPEG2 workload. The greedy approach with perfect predictions (i.e. the predicted amount of work is the actual amount of work) is used as a baseline, and we show that the optimal solution requires up to 55% less energy. Our, easy to implement, constant time algorithm saves only one percentage point less energy than the optimal solution.

We extend some of these results to multiple processor cores with global speed scaling, where tasks have precedence constraints. First, we study a simplified (yet

[3] Tasks have agreeable deadlines when the tasks can be ordered such that the arrival times and deadlines of these tasks have an increasing order.

still NP-hard) problem where all tasks share a common arrival time and a common deadline (Chapter 5). This problem involves both scheduling and speed scaling, which have to be considered simultaneously to obtain the optimal solution. We prove that for two cores any schedule of minimal length is energy optimal, and show that this does no longer hold for more than two cores. Instead, we give a scheduling criterion that *does* minimise the energy consumption, and implicitly takes the optimal speeds into account. In addition, we show how to calculate the optimal speed for any schedule, and give an approximation ratio[4] for a class of scheduling algorithms with respect to the energy consumption.

Second, for the global speed scaling case, in which all tasks have individual arrival times and deadlines, we present a transformation to the aforementioned single core problem (Chapter 6). This problem can be solved in quadratic time when no static power is present, and in cubic time when static power is present.

Instead of choosing between speed scaling and sleep modes, both techniques can be combined to reduce the energy consumption even further. We show how to use a combination of these techniques to minimise the energy consumption for uniprocessor frame-based real-time systems in either constant or linear time, depending on workload characteristics (Chapter 7).

In general, the research in this thesis aims to unify the theory on power management for problems that arise with modern computer architectures. A part of the theoretical work the from literature does not consider important practical restrictions. On the other hand, application oriented research projects rarely use the existing theory. Summarising, the general contributions of this thesis are *algorithms* and *concepts* that are straightforward to implement and use in practice.

1.6 Structure of this thesis

The theory from Chapter 2 is required to understand Chapters 3–7. It is advised to read Chapter 3 to get an understanding of existing algorithms. When time is limited, the following reading guidelines can be used (see also Figure 1.5).

For Chapter 2, we assume that the reader has a basic understanding of approximation algorithms, convex optimisation, and scheduling. An introduction to these subjects can be found in Appendix A. Chapters 3, 4, 5 and 7 can be read independently after reading Chapter 2. Since Chapter 6 combines the theory from chapters 4 and 5, these chapters must be understood before reading Chapter 6. Finally, in Chapter 8, conclusions are given, and some suggestions for future work are provided.

[4] The costs of an algorithm with the approximation ratio ρ are at most ρ times the optimal costs (see Appendix A).

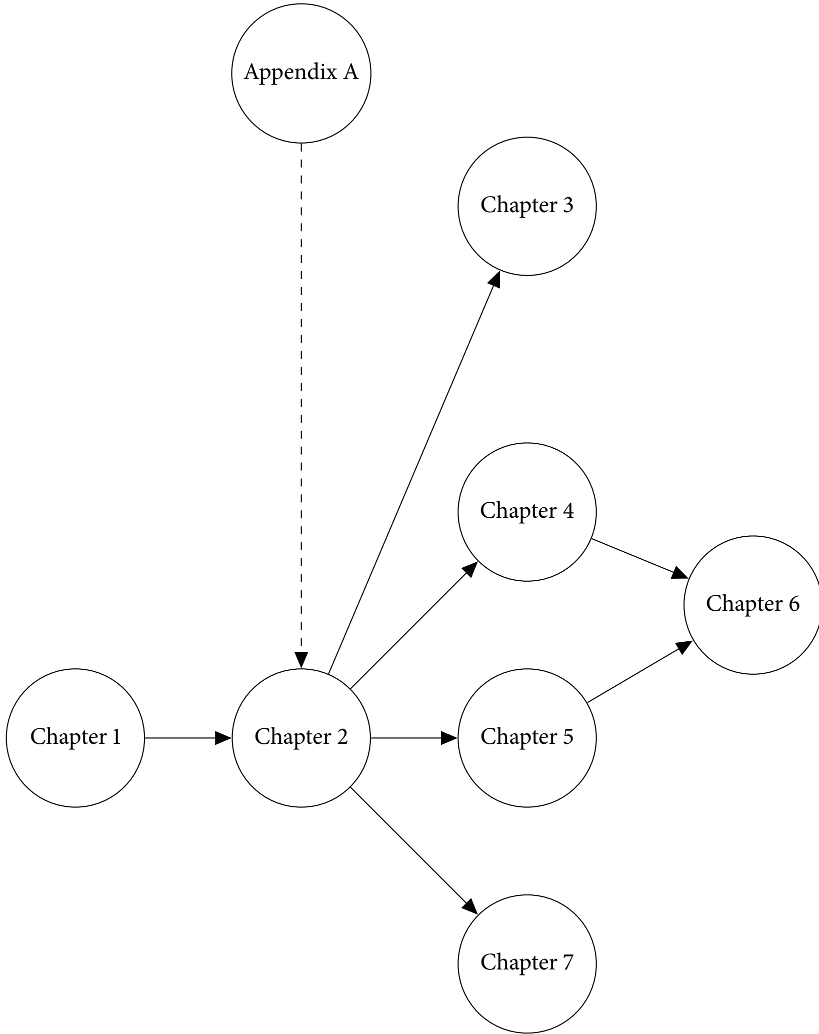


Figure 1.5 – Chapter reading dependencies.

Chapter 2

Background

Abstract – *This chapter provides the necessary background on algorithmic power management that is required to understand the topics presented in this thesis. Herein, modelling and notation of tasks and energy are discussed in the context of speed scaling and sleep modes. For these models, many theoretical results from the literature are discussed.*

2.1 Introduction

This chapter describes often used power management models and results. Tasks and the notation used for properties of tasks are introduced in Section 2.2. In Section 2.3 speed scaling is introduced, where the focus is on models for speed scaling for microprocessors. Sleep modes are discussed in Section 2.4, where real-world devices and their characteristics are given as examples. Finally, in Section 2.5 a notation to describe general algorithmic power management problems is presented.

There are a lot of algorithmic power management results that are not limited to a single power management problem. Section 2.6 covers many different algorithmic power management results. This theoretical section is required for understanding Chapters 3–7. Because of the mathematical content of this chapter (especially Section 2.6), a basic understanding of convex optimisation and scheduling is essential. Appendix A provides an introduction to these subjects.

2.2 Tasks

In this thesis, we assume that an application is subdivided into small chunks of work, called *tasks*. The mathematical notation for task properties is introduced in Section 2.2.1. Some particular types of aperiodic real-time systems are of special importance, and are introduced in Section 2.2.2.

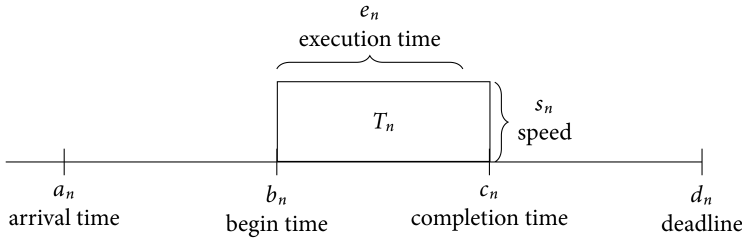


Figure 2.1 – Overview of notation.

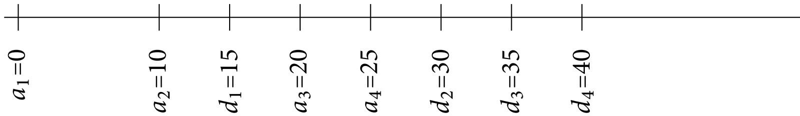


Figure 2.2 – Example of a timeline with agreeable tasks.

2.2.1 notation

In this thesis we consider applications that consist of N tasks that we denote by $T_1, \ldots, T_N$. These tasks have to be scheduled on M processors, where in many cases M=1. Each task has an *execution time* e_n, an *arrival time* a_n and a *deadline* d_n. These times define the *active interval* of a task, which is the time interval $[a_n, d_n]$ during which task T_n must be executed. The tasks have to be completely scheduled within this interval, meaning that a *begin time* b_n and *completion time* c_n have to be specified such that $a_n \leq b_n \leq c_n \leq d_n$. The time between the begin time of the first task that begins until the completion time of the last task that is finished is called the *makespan*. If the tasks have to be executed without interruptions, we furthermore have $c_n = b_n + e_n$. To ease the notation for boundary situations, we define $a_0 := 0$ and $a_{N+1} := d_N$. For a relation between the above concepts and notation, see Figure 2.1.

In some cases, tasks have *precedence constraints*, denoted by $T_n \prec T_m$, meaning that task T_m can only start after task T_n is finished.

2.2.2 types of aperiodic real-time systems

In addition to precedence constraints, the arrival times and deadlines of tasks may have further restrictions. The most general real-time system has arbitrary arrival times and deadlines. We refer to tasks in such real-time systems as *general tasks*. Problems of this general form are relatively hard to solve, while this generality is not always required or even desired. Because of that, real-time systems with additional restrictions on arrival times and deadlines are studied. One of the most extreme

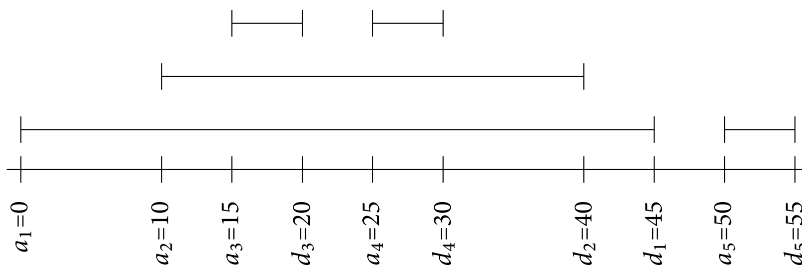


Figure 2.3 – Example of the active intervals of a laminar instance.

examples is a real-time system that has a common arrival time and a common deadline for all tasks, i.e. $a_n = a$ and $d_n = d$ for all n and for some constants a and d.

When for a real-time system holds that $a_n \leq a_m$ if and only if $d_n \leq d_m$ (i.e. tasks with earlier arrival times have earlier deadlines and vice versa), this real-time system is said to have *agreeable deadlines*. For real-time systems with agreeable deadlines, we assume (without loss of generality) that the tasks are ordered such that $a_n \leq a_{n+1}$ and $d_n \leq d_{n+1}$. For an example of arrival times and deadlines of a real-time application with agreeable deadlines, see Figure 2.2.

A real-time system is called a *laminar instance* whenever for each set of tasks, the active interval ($[a_n, d_n]$ for task T_n) of any two tasks do not overlap, or one is completely contained within the other. Formally, when for every two tasks T_i and T_j it either holds that $[a_i, d_i] \subseteq [a_j, d_j]$, $[a_j, d_j] \subseteq [a_i, d_i]$ or $[a_i, d_i] \cap [a_j, d_j] = \emptyset$ [11]. In a graphical representation of this property, the active interval of task T_i is drawn on top of the active interval of task T_j when $[a_i, d_i] \subset [a_j, d_j]$, which creates layers of tasks and explains the term “laminar instances” (for an example, see Figure 2.3). According to Li et al. [60] these structures occur in recursive programs. Since the tasks can be arranged in a tree structure that expresses this recursive structure, laminar instances are also referred to as tree-structured tasks [60].

2.3 Speed scaling

In Section 2.3.1, speed scaling is introduced, with a focus on speed scaling of microprocessors. The notation we use for speed scaling is introduced in Section 2.3.2.

2.3.1 processor models

An important objective of the majority of papers considered in the survey in the next chapter is energy minimisation of microprocessors. Hence, in the following we concentrate on speed scaling of microprocessors.

Microprocessors have a clock frequency, which represents the speed of the processor. For many systems the speed of the computer memory (and other peripherals)

does not scale with the clock frequency of the processor because it is a separate device that does not necessarily use the same clock frequency. In other words, the speed of the overall system (and of tasks) does *not* scale linearly with the clock frequency [32]. However, all algorithms that we survey assume that the speed *does* scale linearly with the clock frequency, and hence we will also assume this throughout this thesis. This assumption leads to an underestimation of the speed when the clock frequency is decreased with respect to some reference clock frequency, which means that in practice tasks finish earlier than was predicted using the models. Note, that for a multicore processor with only local memories (e.g., scratchpad memory) the speed does scale linearly with the processor clock frequency.

As a consequence of the above mentioned assumption, clock frequency and speed are synonyms, and therefore we use s to denote both the speed and clock frequency. In this thesis, we mostly use the terms speed and *speed scaling*, instead of clock frequency and DVFS, in line with the majority of papers on algorithmic power management. We come back to the practical implications of the assumption that speed scales linearly with the clock frequency in Chapter 4.

For multicore processors, there are two main flavours of speed scaling, namely *local speed scaling* and *global speed scaling*. While local speed scaling changes the speed per individual core, global speed scaling makes these changes for the entire chip. For this reason, the optimal solutions to the local and global speed scaling problems are not interchangeable. Global speed scaling is in practice the most common of these techniques, since it is cheaper to implement [24, 66]. Examples of modern processors and systems that use global speed scaling are the Intel Itanium, the PandaBoard (dual-core ARM Cortex A9), IBM Power7 and the NVIDIA Tegra 2 [50, 51, 66, 92].

Nowadays, most modern microprocessors are built using Complementary Metal Oxide Semiconductor (CMOS) transistors. When the clock frequency of a CMOS processor is decreased, the voltage may be decreased as well. Dynamic Voltage and Frequency Scaling (DVFS) [84] is a power management technique that allows the clock frequency and voltage to be changed at run-time. Both the clock frequency and the voltage influence the power consumption of a processor. Hereby, the energy consumption is obtained by integrating power over time.

In general, there are two major types of power consumption, namely dynamic power and static power. *Dynamic power* is consumed due to activities of the processor, i.e., due to transitions of logic gates. A CMOS transistor charges and discharges (parasitic) capacitances when it switches between logical zero and logical one. The dynamic power is given by $ACV_{\text{dd}}^2 s$, where V_{dd} is the supply voltage, s is the clock frequency (i.e., speed), C is the capacitance and A is the activity factor (average number of transitions per second) [48]. For a given clock frequency, the minimal voltage is bounded and many papers (implicitly) simplify this relation using $V_{\text{dd}} = \beta s$ for some constant $\beta > 0$ (e.g., [43, 90]). This gives the dynamic power model

$$p^{\text{dyn}}(s) = \gamma_1 s^{\alpha}, \tag{2.1}$$

where α is a system dependent constant (usually, $\alpha \approx 3$) and $\gamma_1 = AC\beta^{\alpha-1}$ contains both the *average* activity factor and switched capacitance. Most papers assume that γ_1 is constant for the entire application. Some papers use a separate constant $\gamma_1(n)$ for each task (referred to as *nonuniform loads* [54] or *nonuniform power*), because the activity may deviate for different types of tasks. This makes the power function in practice (to some extent) nonuniform, but throughout this thesis we assume γ_1 is constant. This is done to keep the notation simple, and when the power function is nonuniform we assume that the theory that we present in Section 2.6.7 is applied.

Static power is the power that is consumed *independently* of the activity of the transistors, which is independent of the clock frequency. However, there are two different definitions of static power that are used in the literature. The first definition of static power, popular in algorithmic papers (e.g., [26]), takes static power as a constant function (i.e., independent of the clock frequency), and is given by

$$p^{\text{static}}(s) = \gamma_2,$$

where γ_2 is a system dependent constant. The second definition—often used in computer architecture papers—uses the voltage to express the static power. Although it is physically modelled using an exponential equation, the following linear approximation with system dependent constants γ_2 and γ_3 is popular [68]:

$$p^{\text{static}}(V_{\text{dd}}) = \gamma_2 + \frac{\gamma_3}{\beta} V_{\text{dd}},$$

and the relation between the voltage and the clock frequency ($V_{\text{dd}} = \beta s$) gives

$$p^{\text{static}}(s) = \gamma_2 + \gamma_3 s.$$

Note, that this relation makes the static power—that is directly independent of the clock frequency—indirectly dependent on the clock frequency. For this reason static power depends, in our context, on the clock frequency. The resulting static energy for w work is $\gamma_2 \frac{w}{s} + \gamma_3 w$, when it is assumed that static power is consumed until all work is completed (see the discussion in Section 2.6.3). This shows that the constant γ_3 does not influence the choice of the optimal clock frequency in the case of energy minimisation, which is the focus of this thesis. Thus, we can assume without loss of generality that $\gamma_3 = 0$ and use $p^{\text{static}}(s) = \gamma_2$ to model the static power. Since both models lead to the same optimal solution, it is for optimisation not relevant which of the two static power models is used.

For microprocessors, the power function does not fully describe all energy that is used, since changing the clock frequency also has an energy and time overhead. The recent article by Park et al. [68] shows that the time and energy overheads of DVFS are in the same order of magnitude as the overhead of context switching. For example, the transition delay overhead is at most 62.68µs on an Intel Core2 Duo E6850 [68]. Furthermore, most algorithms avoid changing the clock frequency

often because of the convexity of the power function (see Section 2.6.1), hence the number of speed changes is relatively low. Because of these two reasons, we may assume that the energy overhead of changing the clock frequency is negligible in case of DVFS. We make this assumption throughout this thesis.

In practice, it is important to consider whether DVFS can be used to decrease the energy consumption or not. Increasing the speed, such that all tasks finish earlier and the processor can be turned off, is not always possible. For example, in the common situation where there are arrival times, increasing the speed may only result in relatively small idle periods during which the processor cannot be put to sleep. In such situations, it is empirically shown that DPM cannot be applied and DVFS still works well [80].

2.3.2 speed scaling notation

Generally, we define the total power consumption (both static and dynamic) as a *power function* $p : \mathbb{R}_0^+ \to \mathbb{R}_0^+$, that maps the speed to power, i.e. for a speed s the power consumption is given by $p(s)$. The static power is consumed from time t^B, the time the device is powered on, until time t^C, the time the device is powered off. Both t^B and t^C are problem dependent, and typically $t^C = \max_n\{d_n\}$ or $\max_n\{c_n\}$ (i.e. the processor is powered down after the last task is finished, or after the last deadline).

For task T_n we denote by w_n the amount of work (e.g., in number of clock cycles). To ease the notation, we generally use the term *work* instead of amount of work. We denote the speed at which the task is executed by s_n, leading to an execution time of $e_n = \frac{w_n}{s_n}$. In some cases, the speed is changed during the execution of a task. Then we slightly abuse notation, and use the speed function $s : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ that gives the speed as a function of the time.

The speed can be chosen from a set $\mathcal{S}$, which is either a continuous set ($\mathcal{S} = \mathbb{R}_0^+$) or a finite discrete set with K speeds ($\mathcal{S} = \{\bar{s}_1, \dots, \bar{s}_K\}$, where we assume without loss of generality that $\bar{s}_1 < \dots < \bar{s}_K$). When a speed must be chosen from a continuous (discrete) set, we call this speed a continuous (discrete) speed, and refer to a problem with such restriction as a continuous (discrete) speed scaling problem.

2.4 Sleep modes

Many devices allow transitions to a low power mode, which is referred to as *sleep mode*. A device that transitions to a sleep mode is usually (partially) powered down. When, for example, a processor is transitioned to a sleep mode, its state is stored. This state is recovered when the processor is awakened, which costs energy.

A device can have multiple sleep modes. The deeper the sleep mode, the more time and energy it costs to wake up. Many devices have in common that a cost in both latency and energy is associated with switching to a sleep mode and waking up.

Table 2.1 – Power consumption and break-even time for some devices in given sleep modes.

Device	Power	Latency	Break-even time
Sensor node [78]	1040/400/270/ 200/10 mW	5/15/ 20/50 ms	8/20/ 25/50 ms
Harddisk (Hitachi DK23AA-60) [63]	0.77/0.0 W	10.61 s	24.41 s
Network card (Linksys NP 100) [63]	0.76/0.0 mW	2.75 s	3.61 s
Harddisk (IBM Ultrastar 36Z15) [94]	10.2/2.5 W	12.4 s	15.2 s
Beowolf cluster node [42]	1/0.766/0.1/0.1[1]	3/7/70 s	6/10/100 s
Laptop LCD [56]	21.1/17.1 W	7.6 s	15.6 s
WLAN card [83]	0.9/0 W	0.3 s	0.7 s
Ethernet card (WaveLAN) [62]	1.43/0.05 W	0.34 s	0.39 s

Often for the wakeup, a state has to be restored, or some physical action is required, such as spinning up a harddisk.

The power required by a device m in sleep mode ℓ is denoted by $P_{m,\ell}$. Furthermore, the total time required to transition a device m from the active mode to the sleep mode ℓ, and back to the active mode is denoted by $T_{m,\ell}$. This time is called the (*transition*) *latency*.

Transitioning to a sleep mode and back consumes energy. To balance between the energy savings and the energy costs of transitioning to/from sleep modes, the *break-even time* is often used in the literature. This is the minimal time for which it is worthwhile to transition to a sleep mode (i.e. the energy consumption decreases). It is commonly assumed (e.g., [12]) that the transition latency is lower than the break-even time. It was shown empirically that algorithms that use this assumption still work well when the latency is taken into account [46]. Table 2.1 shows some example devices for which this assumption holds.

If an idle interval of length I occurs, we should use the sleep mode ℓ of a device m with $B_{m,\ell} \leq I$ (hereby $B_{m,\ell}$ denotes the break-even time of sleep mode ℓ of device m) that has the lowest energy consumption of all sleep modes. The idle-time energy consumption in the best sleep mode together with the transition energy for this mode can be expressed as a function of the length of the idle period, denoted by E^{sl}, and is referred to as the *idle-time energy function*. Figure 2.4 shows the energy consumption of the sensor node from Table 2.1 as a function of the length of the idle period. The function E^{sl} is in general an increasing concave piecewise-linear function. Clearly, an idle period of zero length consumes no energy, hence $E^{\mathrm{sl}}(0) = 0$.

When there are multiple devices involved, the total energy consumption is obtained by summing over all devices. Since the sum of increasing concave piecewise-linear

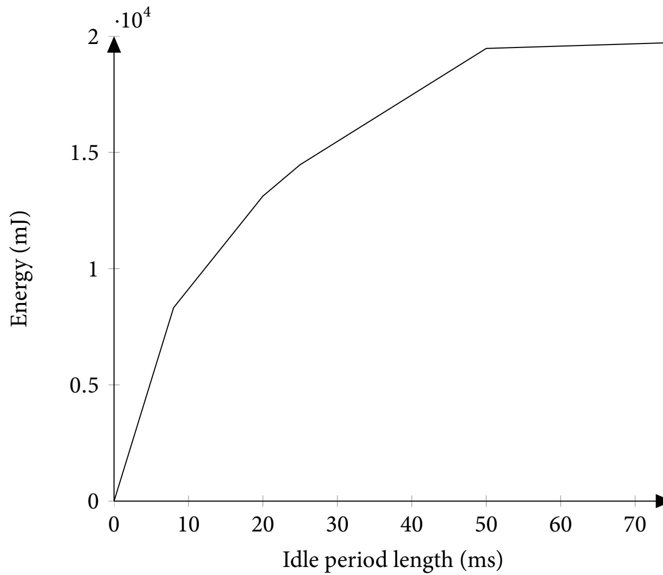


Figure 2.4 – Concave idle-time energy function (E^{sl}) for a sensor node [78].

functions is again an increasing concave piecewise-linear function, we define E^{sl} such that it includes the energy consumption of all devices.

The modelling of sleep modes is extensively discussed in Chapter 7.

2.5 Problem notation

This section introduces a compact notation (based on Grahams three field notation for scheduling problems [35]) to describe a wide variety of algorithmic power management problems. The notation is similar to what is used in the algorithmic power management literature (e.g., [16]), but avoids several ambiguities by making explicit what kind of power management techniques are used. We use this notation extensively to describe the power management problems in the following chapters.

We specify a general power management problem by three fields $\mathfrak{a}|\mathfrak{b}|\mathfrak{c}$, where $\mathfrak{a}$ denotes the system properties, $\mathfrak{b}$ describes the tasks and their constraints, and $\mathfrak{c}$ is the objective for optimisation. The fields with their possible entries and their meaning are given in Table 2.2. For convenience, this table is repeated in Appendix B. A brief discussion of this notation follows below.

» $\mathfrak{a}$: The system field $\mathfrak{a}$ describes the architecture of the system. This includes the number of processors, whether speed scaling (*ss*) and/or sleep modes (*sl*) are used, and properties of the system with respect to speed scaling and/or

Table 2.2 – Notation for algorithmic power management problems.

Field	Entry	Meaning
$\mathfrak{a}$	1	Single processor
	P_M	M parallel processors
	ss	Speed scaling is supported
	nonunif	A nonuniform power function is used (ss implied)
	disc	Discrete speed scaling is used (ss implied)
	global	Global speed scaling is used (ss implied)
	sl	Sleep modes supported
$\mathfrak{b}$	a_n	Arrival time
	a_n=a	Same arrival time a for all tasks
	d_n	Deadline constraint
	d_n=d	Same deadline constraint d for all tasks
	w_n=w	All tasks have workload w
	agree	Agreeable deadlines ($a_n \leq a_m \Leftrightarrow d_n \leq d_m$)
	lami	Laminar instances ($[a_i, d_i] \subset [a_j, d_j] \vee [a_j, d_j] \subset [a_i, d_i] \vee [a_i, d_i] \cap [a_j, d_j] = \emptyset$)
	prec	Tasks have precedence constraints
	pmtn	Preemptions are allowed
	prio	Tasks have a fixed priority
	migr	Task migration is allowed
	sched	A schedule is given
$\mathfrak{c}$	E	Minimise the energy consumption

sleep modes (see Table 2.2). The entries *nonunif*, *disc* and *global* all imply speed scaling (*ss*) to keep the notation concise.

» $\mathfrak{b}$: The second field, $\mathfrak{b}$, contains the task characteristics like arrival time, deadline, restrictions on the ordering of timing constraints of tasks (*agree*, *prec*, *lami*), and scheduling properties (*migr*, *pmtn*, *prio*, *sched*). When a_n occurs in this field, it means that tasks have arrival times, otherwise a_n=0 (for all n) is implied.

We study energy minimisation under deadline constraints. For this reason, d_n always occurs in $\mathfrak{b}$ and implies that deadlines must be met.

» $\mathfrak{c}$: The third field, $\mathfrak{c}$, contains the scheduling objective. In the context of this thesis, third field $\mathfrak{c}$ only contains "E" to denote that the energy should be minimised.

2.6 Theoretical results

Over the years, many theoretical results on algorithmic power management have been obtained. Some of these results form the basis of many algorithms, and some other results relate problems to each other such that the solution to one problem

can be used to find a solution to another problem. This section introduces the fundamental theoretical results and concepts in the area of algorithmic power management. One of the most important results is that it is optimal to use a constant speed between begin and completion time of tasks due to the convexity of the power function (Section 2.6.1). Although this result only holds for convex power functions, using the techniques presented in Section 2.6.2, all power functions can be "made" convex. Even when a constant speed is used, one has to be careful that this speed is not too low because then static power may dominate (Section 2.6.3).

When only a finite number of speeds is available, many speed scaling problems (with a given schedule) can be formulated as a linear program (Section 2.6.4). In the single processor case, it is furthermore straightforward to derive the solution to this discrete problem from the solution to the continuous problem (Section 2.6.5).

In the optimal solution of several multiprocessor problems, the power consumption remains constant over time. This fact is referred to as the *power equality* (Section 2.6.6). In Section 2.6.7, the situation where every task has a different power function is discussed. A simple transformation is presented that transforms this problem to the problem where all tasks have the same power function.

2.6.1 Constant speed

Whenever a single processor executes a single task using varying speeds, the energy consumption can be decreased by running it at the average speed. This even holds when the task is executed with interruptions (i.e. on times given by any finite set $\mathcal{T}$). This result holds for all convex power functions, where this property does not form a restriction as is discussed in Section 2.6.2. We formalise this result, which is a direct consequence of Jensen's inequality [47], in the following theorem (see Appendix A).

Theorem 2.1. *Given a task with w work which is executed at the times given by the set $\mathcal{T}$ (i.e. $w = \int_{\mathcal{T}} s(\tau)d\tau$) and is executed on a processor with a convex power function. Then the following inequality holds:*

$$p\left(\frac{w}{e}\right)e \leq \int_{\mathcal{T}} p(s(\tau))d\tau.$$

Proof. The infinite version of Jensen's inequality states:

$$p\left(\frac{1}{\int_{\mathcal{T}} 1d\tau}\int_{\mathcal{T}} s(\tau)d\tau\right) \leq \frac{1}{\int_{\mathcal{T}} 1d\tau}\int_{\mathcal{T}} p(s(\tau))d\tau.$$

Multiplying this equation by $\int_{\mathcal{T}} 1d\tau$ directly leads to the result of the lemma. □

Theorem 2.1 shows that for continuous speed scaling, there always exists a constant speed that is optimal for a single task. Many papers (e.g., [43, 60, 90]) use the idea behind Theorem 2.1, and show that minimising unnecessary speed fluctuations on

a single processor is optimal also for situations with more than one task, i.e. $N > 1$. However, when there are arrival times, deadlines, etc., the optimal constant speed may change on these specific times, meaning that the optimal speed function is piecewise constant.

2.6.2 Nonconvex power function

The previous section (and with it, a large part of the literature) assumes that the power function is convex, but for technical reasons this is not always the case. However, it is possible to circumvent this by not using the speeds where the function is not convex, since we can show that these speeds are not efficient. This process is first explained for discrete speed scaling. When the assumption that p is convex does not hold, an additional step is required to "make" the power function convex, based on the following observation.

Assume three given speeds $\bar{s}_i < \bar{s}_j < \bar{s}_k$ (let $\bar{s}_j = \lambda\bar{s}_i + (1-\lambda)\bar{s}_k$ for some $\lambda \in (0,1)$) and w work, where

$$p(\bar{s}_j)w \leq p(\bar{s}_i)\lambda w + p(\bar{s}_k)(1-\lambda)w, \tag{2.2}$$

does not hold. Then executing the work at speed $\bar{s}_j$ would cost more energy than executing a part of the work at $\bar{s}_i$ and the remaining work at $\bar{s}_k$. In this case we call $\bar{s}_j$ an *inefficient speed*.

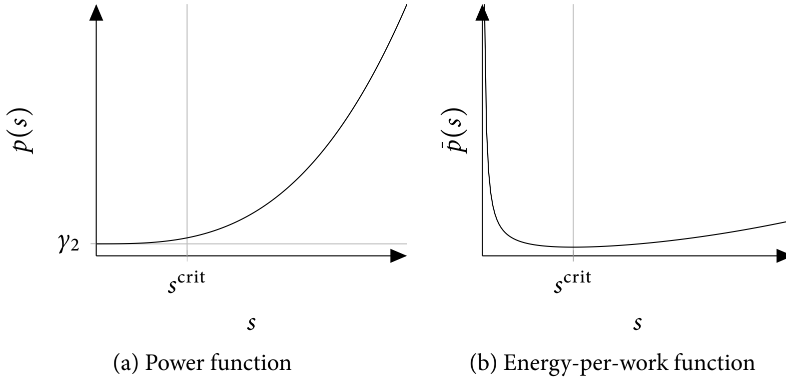
Based on the above, we may assume that all speeds in $\mathcal{S}$ are *efficient speeds*, thus (2.2) holds for all speeds (i.e. inefficient speeds are "discarded") [41]. This implies that we can always assume without loss of generality that the power function is convex.

Bansal et al. [18] state that a similar procedure can be followed for continuous speed scaling. Note, that the static and dynamic power models from Section 2.3.1 are already convex.

2.6.3 Critical speed

Only using the fact that the power function is convex may not be enough to find the optimal solution to some speed scaling problem. This has to do with the static power, which is the power consumed independent of the speed.

In practice, processors consume static power ($\gamma_2 > 0$), i.e. the power consumption at the speed 0 is nonnegative ($p(0) > 0$). Unfortunately, most papers do not clearly define for which time period they take the static power into account. For example, Yao et al. [90] only assume that the power function is convex and do not mention static power. However, their result only holds when the static power cannot be influenced, i.e. when it is accounted for until the deadline of the last task and not only to the completion time of the last task. In this case, static power cannot be influenced, hence the situation where $p(0) = 0$ gives the same solution as the case where $p(0) > 0$. This scenario is mentioned by Irani et al. [47].



(a) Power function (b) Energy-per-work function

Figure 2.5 – Speed scaling functions.

For the other scenario, where the static power is active until the last task has finished, not only the power function should be studied, but also the *energy-per-work function*:

$$\bar{p}(s) = \frac{p(s)}{s}.$$

This function gives the energy consumption of a unit work (instead of a unit time), has a global minimiser s^{crit} (called the *critical speed* [49]), and is increasing on $s \geq s^{\text{crit}}$ [47]. Since s^{crit} is a global minimiser, all speeds below s^{crit} require more energy per unit work, while it takes longer to execute. Hence, if the schedule length can be decreased by increasing speeds to s^{crit}, the energy consumption is reduced.

Example 2.1. *For the power function given in Figure 2.5a, the respective energy-per-work function is given by Figure 2.5b. In Figure 2.5a, the static power is indicated using γ_2. This shows that more energy is required per unit work for speeds below s^{crit}.*

2.6.4 Discrete speed scaling as a linear program

Besides static power, many processors have the restriction that only a small set of speeds is allowed (*discrete speed scaling*). Many discrete speed scaling problems with a given schedule can be formulated as a linear program, as we show below.

When discrete speed scaling is considered with K discrete speeds, the decision to be made is to determine the amount of work of task T_n that is executed at speed $\bar{s}_k$. If we denote this amount by $w_{n,k}$ (i.e. $\sum_{k=1}^{K} w_{n,k} = w_n$), the total energy consumption of all tasks together is given by

$$\sum_{n=1}^{N} \sum_{k=1}^{K} p(\bar{s}_k) w_{n,k},$$

which is a linear function of the decision variables $w_{n,k}$. These variables, together with the begin time of tasks, form the decision variables of the linear program.

Constraints like arrival time, deadline and precedence constraints can all be formulated as linear constraints. Therefore, many discrete speed scaling problems (with or without a given schedule) can be formulated as a linear program [54, 75] and, thus, be solved in polynomial time.

2.6.5 relation between continuous and discrete speed scaling

Writing *discrete speed scaling* problems as a linear program and solving it with linear programming software provides few insights. Instead, a tailored algorithm for finding the optimal speeds is desirable. Such algorithms are described in many papers (e.g., [43, 72, 90]) for continuous speed scaling, while in practice most processors support only discrete speed scaling. When a single task is considered, the optimal speed s resulting from the continuous case can be used to determine the optimal speeds for the discrete case. When the speed s is not one of the available discrete speeds, using the neighbouring speeds $\bar{s}_i \leq s \leq \bar{s}_{i+1}$ leads to an optimal solution. More precisely, the first part of the work is executed at speed $\bar{s}_{i+1}$ and the remaining work is executed at speed $\bar{s}_i$. Hereby, the fractions are calculated such that the overall time remains the same. We refer to this as *simulating* continuous speed scaling.

Simulating is optimal within the execution interval of a single task. For multiple tasks we require a convex power function in the case of continuous speed scaling. In case of discrete speed scaling, we use multiple speeds for a task, hence we must somehow relate these multiple speeds to a convex power function. We do this by taking the average speed of a task, and derive a function that gives the average power for each of these speeds (called *average power function*). Hsu and Feng [41], Kwon and Kim [54] have proven that this average power function is a convex piecewise linear function. Hence, any continuous speed scaling algorithm that assumes convexity can be used to find the optimal average speeds, after which the discrete assignment can be determined using simulation.

2.6.6 power equality

The previous sections mainly focussed on the single processor case. In the multiprocessor case with precedence constraints, new issues arise that are best illustrated with an example.

Example 2.2. *Consider the three tasks from Figure 2.6 each with w work, which are to be executed on a local speed scaling* multiprocessor *system. Task T_1 has to be finished before tasks T_2 and T_3 can be executed, and the application as a whole has a global arrival time* 0 *and a global deadline d. An example of a naive speed assignment is $s_1 = s_2 = s_3 = \frac{2w}{d}$. Note that Theorem 2.1 cannot be used to argue that this assignment is optimal. In fact, this assignment is not optimal, since it can be improved by slightly increasing s_1 such that task T_1 consumes slightly more energy, while two tasks T_2 and*

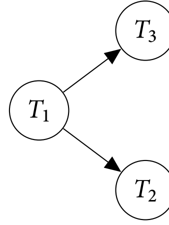


Figure 2.6 – Task graph.

T_3 can decrease their energy consumption. The speed of task T_1 should not be too high, because then its energy consumption is no longer (over)compensated by tasks T_1 and T_2.

This example illustrates that the optimal speeds depend on the amount of parallelism of the scheduled tasks. Pruhs et al. [72] introduce the *power equality* for tasks with a common arrival time and deadline: in the optimal solution, the power consumption remains constant. Thus, the power is constant, and the speeds can be calculated using this power and the number of parallel executed tasks. This generalises Theorem 2.1. For this concrete situation of Figure 2.6, this means that $p(s_1) = p(s_2) + p(s_3)$.

Example 2.3. *Consider again the task graph from Figure 2.6 with the power function $p(s) = s^3$, and assume that all the tasks have 10 work (i.e. $w_1 = w_2 = w_3 = 10$), and the global deadline is 40. A naive speed assignment uses the constant speed $s_1 = s_2 = s_3 = \frac{1}{2}$.*

As in an optimal solution, tasks T_2 and T_3 complete simultaneously, we get $s_2 = s_3$. Due to the power equality, for the optimal solution it holds that

$$p(s_1) = p(s_2) + p(s_3) = 2p(s_2).$$

Using $p(s) = s^3$ and some elementary algebra gives $s_1 = \sqrt[3]{2}s_2$. Furthermore, the energy consumption is minimised when $\frac{w_1}{s_1} + \frac{w_2}{s_2} = 40$. Thus $s_1 = \frac{1+\sqrt[3]{2}}{4}$.

2.6.7 nonuniform power

Most papers assume that uniform power is used (see Section 2.3.1), while in practice the parameter γ_1 of the power function is not constant (i.e. nonuniform) for all tasks [54], and a task specific factor $\gamma_1(n)$ for the dynamic power should be used for each task T_n. The resulting power function can also be related to the previous section on the power equality; where γ_1 often contains the number of active cores (see Chapter 6).

The dynamic energy consumption for N tasks with nonuniform power functions is given by (see (2.1) and Section 2.3.1):

$$E = \sum_{n=1}^{N} \gamma_1(n) s_n^{\alpha} \frac{w_n}{s_n}. \tag{2.3}$$

Fortunately there is an elegant transformation due to Kwon and Kim [54] that can reduce this expression to one with a constant power parameter γ_1. Using the substitution of variables $\mathring{w}_n = \sqrt[\alpha]{\gamma_1(n)} w_n$ and $\mathring{s}_n = \sqrt[\alpha]{\gamma_1(n)} s_n$, (2.3) becomes

$$E = \sum_{n=1}^{N} \mathring{s}_n^{\alpha} \frac{\mathring{w}_n}{\mathring{s}_n}, \tag{2.4}$$

the execution time of task T_n becomes $\frac{\mathring{w}_n}{\mathring{s}_n}$, and $\gamma_1 = 1$ for all tasks.

The newly obtained problem has uniform power, can be solved using classic algorithms, and the resulting solution can be transformed back to a solution to the problem with nonuniform power.

2.6.8 flow problems

Several power management problems can be reduced to a (convex) flow problem. However, as these formulations as a flow problem depend on the concrete algorithmic power management problem, we do not discuss this technique in more detail. We refer the interested readers to three papers [6, 10, 15] where such techniques are used to solve the problem $P_M; ss \mid a_n; d_n; \text{pmtn}; \text{migr} \mid E$; in Section 3.3.1 a brief discussion on these papers is given.

2.7 Conclusions

Using the models presented in this chapter, an application that consists of tasks and power management decisions can be modelled. The considered power management techniques are speed scaling and sleep modes. Speed scaling is available as DVFS on multiprocessors, while sleep modes are available as DPM. Both techniques can be used individually and in isolation, and in all cases the schedule may influence the energy savings. Many theoretical results from the algorithmic power management literature were discussed, and will be used in the following chapters.

Chapter **3**

Related Work

Abstract – *This chapter discusses previous research that is related to the subjects of this thesis. First, algorithmic power management research on energy minimisation respecting real-time constraints is discussed (both the uniprocessor and multiprocessor cases). Second, the work related to chapters 4–7 is discussed. This literature overview shows that online speed scaling for uniprocessor systems is not fully explored, global speed scaling for tasks with precedence constraints is missing and desired, and the optimal combination of speed scaling and sleep modes for frame-based real-time systems is not given in the literature.*

3.1 Introduction

This chapter provides an overview of power management research. We start with a survey of algorithmic power management papers on offline energy minimisation under deadline constraints, closely related to the core topic of this thesis. The first part of the survey discusses uniprocessor algorithms (Section 3.2) and the second part discusses multicore algorithms (Section 3.3).

The remaining part of this chapter contains sections on specific subtopics which are in the focus of this thesis, namely online energy minimisation (Section 3.4), global speed scaling (Section 3.5), and energy minimisation of frame-based real-time systems (Section 3.6).

3.2 Uniprocessor problems

This section surveys algorithms for uniprocessor power management problems (see Table 3.1), and relates these (when applicable) to the results that were presented in Section 2.6.

Table 3.1 – Uniprocessor power management problems.

Section	Problem	Papers
Arbitrary ordering of tasks (Section 3.2.1)	$1; \text{ss} \mid a_n; d_n; \text{pmtn} \mid E$	[17, 61, 90]
	$1; \text{disc} \mid a_n; d_n; \text{pmtn} \mid E$	[41, 41, 61]
	$1; \text{ss} \mid a_n; d_n; \text{pmtn}; \text{prio} \mid E$	[73]
	$1; \text{ss} \mid a_n; d_n \mid E$	[11, 16]
	$1; \text{ss}; \text{nonunif} \mid a_n; d_n; \text{pmtn} \mid E$	[54]
	$1; \text{ss}; \text{nonunif}; \text{disc} \mid a_n; d_n \mid E$	[54]
	$1; \text{sl} \mid a_n; d_n; \text{pmtn} \mid E$	[19]
	$1; \text{ss}; \text{sl} \mid a_n; d_n; \text{pmtn} \mid E$	[4, 47]
Agreeable deadlines (Section 3.2.2)	$1; \text{ss} \mid a_n; d_n; \text{agree} \mid E$	[43, 86]
	$1; \text{sl} \mid a_n; d_n; \text{agree} \mid E$	[9]
	$1; \text{sl}; \text{ss} \mid a_n; d_n; \text{agree} \mid E$	[14]
Laminar instances (Section 3.2.3)	$1; \text{ss} \mid a_n; d_n; \text{pmtn}; \text{lami} \mid E$	[60]
	$1; \text{ss} \mid a_n; d_n = d; \text{pmtn} \mid E$	[60]
	$1; \text{ss} \mid a_n = a; d_n; \text{pmtn} \mid E$	[60]

Recall that for each task T_n we have a workload w_n, an arrival time a_n, and a deadline d_n before which the task has to finish. In the case of speed scaling, a speed s_n is to be determined, leading to an execution time e_n. We use b_n and c_n to denote the begin and completion time of task T_n respectively.

The problems in this section are grouped depending on restrictions on the ordering of the timing constraints of tasks. For *all* problems discussed in this section, the problem consists of finding a schedule together with speeds and/or sleep decisions.

First, the problems without any restrictions on the timing constraints are discussed in Section 3.2.1. Several variations of this problem have a high polynomial time complexity, or are NP-hard. Second, in Section 3.2.2, simpler problems with agreeable deadlines are discussed. For many variants of these problems, algorithms with a quadratic time complexity are known. Third, laminar instances are discussed in Section 3.2.3.

3.2.1 General tasks

First we start with general tasks, tasks have arbitrary arrival times and deadlines, and preemptions are allowed ($1; \text{ss} \mid a_n; d_n; \text{pmtn} \mid E$). According to Albers et al. [6], this is the most extensively studied speed scaling problem in the algorithm oriented literature. Yao et al. [90] present the well-known YDS algorithm (named after the authors) to solve this problem. This algorithm is often used as a subroutine by other algorithms, and in complexity proofs.

The considered problem involves both scheduling and speed scaling. When the optimal speeds are given, and a feasible schedule exists, scheduling the tasks in

Table 3.2 – Tasks for Example 3.1.

Task	Arrival time	Deadline	Workload
T_1	0	30	30
T_2	5	10	10
T_3	15	55	10
T_4	25	35	10

order of their deadline using the Earliest Deadline First (EDF) algorithm always leads to such a feasible schedule [90]. YDS uses this to solve the scheduling part of the problem, subsequently only speeds remain to be calculated. For this, the YDS algorithm avoids unnecessary speed changes (see Section 2.6.1), and has the property that the speeds in the optimal solution cannot be lowered to decrease the energy consumption without violating deadlines.

The YDS algorithm works with time intervals of the form $I_{i,j} = [a_i, d_j]$, where $a_i < d_j$. The density of such an interval is defined as

$$g(I_{i,j}) = \frac{\sum_n w_n}{d_j - a_i},$$

where the sum is taken over the workload of all tasks T_n with $[a_n, d_n] \subseteq I_{i,j}$. The density determines the minimal average speed that has to be used to execute these tasks completely within this interval. The YDS algorithm takes a so-called *critical interval*—an interval $I_{i,j}$ with the highest density—and assigns to all tasks that have to be executed completely within this interval this density as speed. The algorithm removes these tasks from the task set and for the remaining tasks which have deadlines in the interval, the deadlines are adopted in such a way that they coincide with the beginning of the critical interval, and for tasks which have arrival time within the interval, the arrival times are adopted to coincide with the end of the interval (i.e. are set to d_n). By construction, YDS avoids unnecessary speed fluctuations.

Example 3.1 (YDS algorithm). *Consider the tasks from Table 3.2 of which the arrival times and deadlines are depicted in Figure 3.1a. The YDS algorithm first determines the critical interval, which is $I_{2,2}$ in the first iteration of the algorithm (see Table 3.3). Since the density of this interval is $g(I_{2,2}) = 2$, task T_2 is assigned the speed $s_2 = 2$. Next, the interval $I_{2,2}$ is removed, and the arrival times and deadlines of the other tasks are adapted accordingly (see Figure 3.1b).*

In the second iteration, Interval $I_{1,4}$ yields the critical density $g(I_{1,4}) = \frac{4}{3}$ (see Table 3.3), which is assigned as speed to task T_1 and T_4 (i.e. $s_1 = s_4 = \frac{4}{3}$). After removing these tasks, only task T_3 remains in the last iteration (see Figure 3.1c), which is assigned the speed $s_3 = \frac{1}{2}$. An EDF schedule with the aforementioned speeds ensures that the deadlines are met and the energy consumption is minimised.

Table 3.3 – Interval densities for Example 3.1.

Interval	Iteration 1 $g(I_{i,j})$	Iteration 2 $g(I_{i,j})$	Iteration 3 $g(I_{i,j})$
$I_{1,1}$	$\frac{40}{30} \approx 1.333$	$\frac{30}{25} = 1.2$	
$I_{1,2}$	$\frac{10}{10} = 1$		
$I_{1,3}$	$\frac{50}{55} \approx 0.909$	$\frac{50}{50} = 1$	
$I_{1,4}$	$\frac{50}{35} \approx 1.429$	$\frac{40}{30} \approx 1.333$	
$I_{2,1}$	$\frac{10}{25} = 0.4$		
$I_{2,2}$	$\frac{10}{5} = 2$		
$I_{2,3}$	$\frac{30}{50} = 0.6$		
$I_{2,4}$	$\frac{20}{30} \approx 0.667$		
$I_{3,1}$	0	0	
$I_{3,2}$	0		
$I_{3,3}$	$\frac{20}{40} = 0.5$	$\frac{20}{40} = 0.5$	$\frac{10}{20} = 0.5$
$I_{3,4}$	$\frac{10}{20} = 0.5$	$\frac{10}{20} = 0.5$	
$I_{4,1}$	0	0	
$I_{4,2}$	0		
$I_{4,3}$	$\frac{10}{30} \approx 0.333$	$\frac{10}{30} \approx 0.333$	
$I_{4,4}$	$\frac{10}{10} = 1$	$\frac{10}{10} = 1$	

In a schedule created by this YDS algorithm, the processor as scheduled by YDS is active from the arrival of the first task to the deadline of the last task (unless there are no tasks in some interval). Hence, because of static power, this algorithm is only optimal when it is assumed that the processor remains active until the last deadline [47]. To the best of our knowledge, there is no known optimal algorithm for the situation where no static energy is consumed after the last executed task.

The original implementation of the YDS algorithm has a time complexity of $O(N^3)$ [61]. Bansal et al. [17] use the Karush-Kuhn-Tucker (KKT) conditions [21] to prove optimality of YDS for the power function $p(s) = s^\alpha$. Li et al. [61] give a different proof, and present an efficient implementation of YDS with time complexity $O(N^2 \log N)$. They also provide an $O(KN \log N)$ algorithm for the variant with discrete speed scaling with K speeds ($1; \text{disc} \mid a_n; d_n; \text{pmtn} \mid E$). An alternative method for obtaining the optimal speeds in the discrete case is by applying the YDS algorithm, and then simulate the obtained speeds as discussed in Section 2.6.5 [41, 54].

The YDS algorithm schedules tasks in order of their deadlines. This implies that when tasks have a fixed priority, these tasks must be scheduled in the order of these priorities, and the YDS algorithm cannot be used [73]. Yun and Kim [91] show that the fixed priority variant of this problem ($1; \text{ss} \mid a_n; d_n; \text{pmtn}; \text{prio} \mid E$) is NP-hard, and give an FPTAS[1] for the problem.

[1]Fully Polynomial-Time Approximation Scheme, see Appendix A.

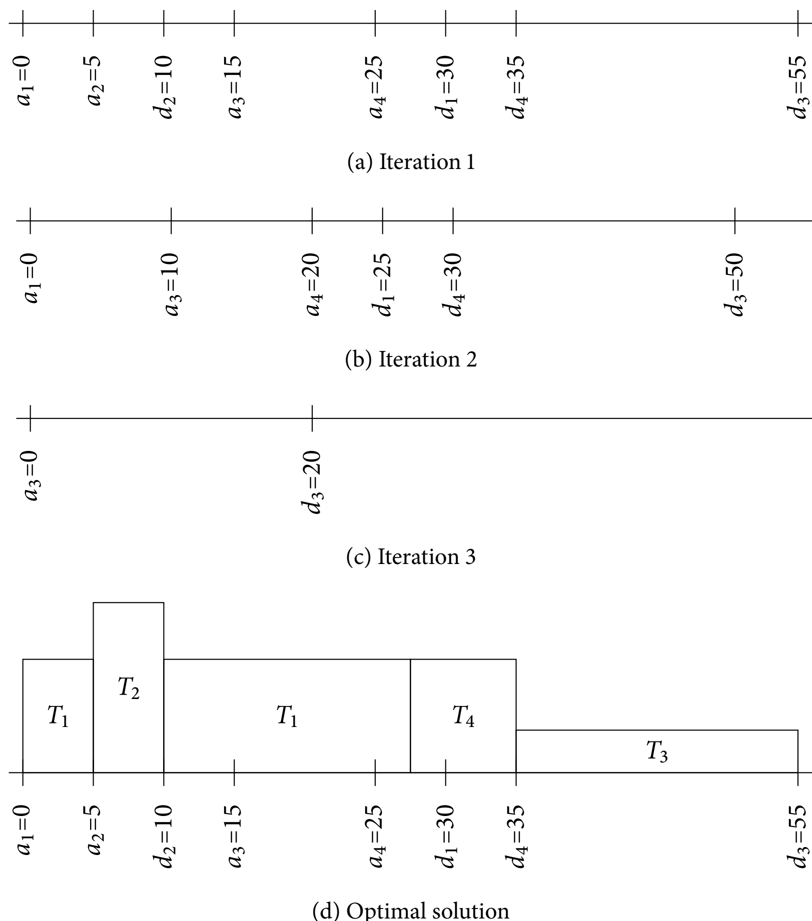


Figure 3.1 – Arrival times, deadlines and optimal solution for Example 3.1.

There exist several other variations to the problem that was introduced by Yao et al. [90]. The variant of the original problem that does not allow preemptions of tasks $(1;\text{ss}\,|\,a_n;d_n\,|\,E)$ is NP-hard [11], and this problem was studied by Bampis et al. [16]. They show that the best known approximation ratio (see Section A.2) for this problem is $(1 + w^{\max}/w^{\min})^{\alpha}$, where $w^{\max}$ and $w^{\min}$ are upper and lower bounds on the work of tasks.

Kwon and Kim [54] study another variation, where the dynamic power is not equal for all tasks $(1;\text{ss};\text{nonunif}\,|\,a_n;d_n;\text{pmtn}\,|\,E)$, for example due to switched capacitances. They solve this problem using a substitution of variables (see Section 2.6.7) and for discrete speed scaling $(1;\text{ss};\text{nonunif};\text{disc}\,|\,a_n;d_n;\text{pmtn}\,|\,E)$ they formulate the problem as a linear program (see Section 2.6.4).

The sleep mode counterpart of the YDS problem is $1; \text{sl} \,|\, a_n; d_n; \text{pmtn} \,|\, E$. Baptiste et al. [19] present an algorithm that is commonly referred to as BCD (named after the authors), that uses dynamic programming to solve the problem in $O(N^4)$ time. Their algorithm is restricted to instances where processors have only a single sleep mode.

Other authors [4, 47] study the combination of speed scaling and sleep modes, namely $1; \text{ss}; \text{sl} \,|\, a_n; d_n; \text{pmtn} \,|\, E$, which is an NP-hard problem. The heuristic by Irani et al. [47] is a 2-approximation (see Section A.2) and is relatively easy to implement. This heuristic uses YDS to determine the speeds, and whenever YDS determines a speed $s_n < s^{\text{crit}}$, this speed is replaced by the speed s^{crit} (this is called an s^{crit}-schedule). These changes create idle time, that can be used to put the processor into a sleep mode. As long as there are tasks available, they are consecutively executed, followed by an idle period of maximal length. This scheduling method is used to create relatively large idle periods. Albers and Antoniadis [4] use a similar method, but with the cut-off speed s^* instead of s^{crit}, where s^* is determined by solving $\bar{p}(s^*) = \frac{4}{3}\bar{p}(s^{\text{crit}})$. Furthermore, they use BCD instead of the scheduling algorithm by Irani et al. [47]. This results in a 4/3-approximation, but has a higher time complexity ($O(N^4)$) because the use of BCD.

3.2.2 Agreeable deadlines

In applications like multimedia and telecommunication, the arrival times and deadlines are usually in the same order (i.e. $a_n < a_m \Leftrightarrow d_n \leq d_m$). Such applications are said to have *agreeable deadlines*. For systems with such agreeable deadlines there exist efficient speed scaling and sleep mode algorithms.

Speed scaling for systems with agreeable deadlines ($1; \text{ss} \,|\, a_n; d_n; \text{agree} \,|\, E$) is studied by many authors (e.g., [43, 86]). Huang and Wang [43] present an algorithm that calculates the optimal speeds in quadratic time. Their algorithm first chooses a single speed for all tasks, such that the last deadline is met exactly. Then, a task T_n with the largest violation of an arrival or a deadline is used to divide the set of tasks into two subsets: the tasks before and the tasks after the violation. For a deadline violation, the completion time of task T_n is fixed, while for an arrival time violation the begin time of task T_n is fixed. Then the procedure is recursively repeated for both subsets. Furthermore, a solution to the problem with preemption can be easily translated to a solution without preemption with the same energy consumption [16].

In a variant of this problem, the maximal rate of change of the speed has the constant R as an upper bound (i.e. $\max_t |s'(t)| \leq R$, for some $R > 0$). For this case, the algorithm by Wu et al. [86] finds the optimal solution in quadratic time.

Next to agreeable deadlines with speed scaling, also the problem with sleep modes and the combination of speed scaling and sleep modes are studied. For the problem where the processor has a single sleep mode ($1; \text{sl} \,|\, a_n; d_n; \text{agree} \,|\, E$), the algorithm by Angel et al. [9] can be applied to find the energy optimal schedule. The authors

Table 3.4 – Multiprocessor power management problems.

Section	Problem	Papers
Arbitrary ordering of tasks (Section 3.3.1)	$P_M; \text{ss} \mid a_n = a; d_n = d \mid E$	[5, 72]
	$P_M; \text{ss} \mid a_n; d_n; \text{pmtn}; \text{migr} \mid E$	[6, 10, 15, 20]
	$P_M; \text{ss} \mid a_n; d_n; \text{pmtn} \mid E$	[5, 37]
Agreeable deadlines (Section 3.3.2)	$P_M; \text{ss} \mid a_n; d_n; w_n = 1; \text{agree} \mid E$	[16]

observe that there always exists an optimal solution in which every task T_n starts at either (i) a_n, (ii) c_{n-1}, or (iii) $d_n - e_n$. Note, that the options for the completion time c_{n-1} depends on the begin times of tasks $T_1, \dots, T_{n-1}$. By this, for each task T_k (tasks ordered in EDF order), there are $O(k)$ possible begin times, leading to a quadratic time complexity. This result by Angel et al. [9] is extended by Bampis et al. [14] such that the optimal combination of speed scaling and sleep modes ($1; \text{sl}; \text{ss} \mid a_n; d_n; \text{agree} \mid E$) is found in cubic time.

3.2.3 laminar instances

In this section we study tasks with a nested structure, called *laminar instances* (see Section 2.2.2). Just as for the problem with agreeable deadlines, this restriction makes the problem easier to solve. In fact, the case where all deadlines or all arrival times are the same has both agreeable deadlines *and* is a laminar instance.

Li et al. [60] give an efficient polynomial algorithm to find the optimal speeds for laminar instances ($1; \text{ss} \mid a_n; d_n; \text{pmtn}; \text{lami} \mid E$). Two important subproblems are (i) all tasks have a shared deadline ($1; \text{ss} \mid a_n; d_n = d; \text{pmtn} \mid E$) and (ii) all tasks arrive at the same time ($1; \text{ss} \mid a_n{=}a; d_n; \text{pmtn} \mid E$). For both problems, Li et al. [60] give a linear time solution.

3.3 Multiprocessor problems

This section discusses multiprocessor algorithmic power management problems (see Table 3.4 for an overview). The problems in this section consist of finding a multiprocessor schedule together with speeds and/or sleep decisions. The problem with general tasks, tasks without special restrictions on arrival times and deadlines, is discussed in Section 3.3.1. Algorithms for tasks with agreeable deadlines are discussed in Section 3.3.2.

3.3.1 General tasks

We first consider the relatively simple variant of the problem, where all tasks arrive at time 0, have a shared global deadline, and optimal scheduling and local speed scaling is used (P_M; ss | $a_n = a$; $d_n = d$ | E). This problem is NP-hard [5], since the 3-partition problem can be reduced to it. Pruhs et al. [72] show that the problem of minimising the makespan under an energy constraint can be formulated as the problem of minimising the ℓ_α norm of the processor loads (where α is the exponent in the dynamic power function, see Section 2.3). For the last problem, a Polynomial Time Approximation Scheme (PTAS) exists [8]. In a similar fashion, a PTAS can be derived for energy minimisation under a global deadline constraint.

There are several variations of the problem with arbitrary arrival times and deadlines of task. They differ depending on whether preemptions and migrations of tasks are allowed or not. A widely studied problem considers the combination of local speed scaling and scheduling, where preemptions and migrations of tasks are allowed (P_M; ss | a_n; d_n; pmtn; migr | E). This problem was first studied by Bingham and Greenstreet [20]. They show that the problem is convex, and present an algorithm that involves solving a linear programming problem. Their algorithm is polynomial in the number of tasks, but according to the authors, the complexity is too high for practical applications. However, as they also discuss properties of the optimal solution, their paper is important when studying multiprocessor speed scaling with preemptions and migrations. Albers et al. [6] present a more efficient polynomial time algorithm for the same problem. Their algorithm uses repeated maximum flow computations to minimise the energy consumption. A closely related approach by Angel et al. [10] also uses maximum flow computations to find the optimal solution in polynomial time. The resulting algorithm only is more efficient than that of Albers et al. [6] when a reduced accuracy is allowed. Another approach to the same problem is discussed in the paper by Bampis et al. [15], wherein the optimal speeds are determined by solving a convex cost flow problem. In this approach, execution times correspond to amounts of flow, which have to be sent through the network. The algorithm that solves this problem has a time complexity that depends on the latest deadline. Although the dependency of the complexity on the deadline is a drawback, the presented approach is straightforward and its concepts are interesting for future research in this direction.

Albers et al. [5] study a variant of the problem where migrations are not allowed (P_M; ss | a_n; d_n; pmtn | E). They show that the problem is NP-hard, even for tasks with unit workload. The difficult part of this problem is the assignments of tasks to processors. If such an assignment is given, determining the optimal speeds and scheduling order is straightforward, since YDS can be used for the tasks on each individual processor. The heuristic by Albers et al. [5] sorts the tasks in order of non-decreasing deadlines, and assigns the tasks in this order to the processor with the lowest amount of work assigned to it. This heuristic has an approximation ratio of $2(2 - \frac{1}{N})^\alpha$. A more general version of this problem that has a weighted sum of the energy consumption and flow time as objective is studied by Greiner et al. [37].

3.3.2 Agreeable deadlines

Just as for the uniprocessor problem with agreeable deadlines, in the multiprocessor case a solution to the preemptive problem with no migration can be transformed to a non-preemptive solution with no migration with the same costs [16].

Albers et al. [5] present an optimal algorithm for the multiprocessor agreeable deadline problem where tasks have a unit workload (P_M; ss | a_n; d_n; w_n=1; agree | E). This algorithm sorts the tasks in order of nondecreasing deadlines, assigns them to the processors using round robin scheduling and applies an algorithm that solves 1; ss | a_n; d_n; $w_n = 1$; agree | E (e.g., YDS) to the task sets for each individual processor. For tasks with an arbitrary workload they give an algorithm with an $\alpha^{\alpha}2^{4\alpha}$-approximation.

3.4 Online uniprocessor speed scaling algorithms

One of the main topics of Chapter 4 is online speed scaling for a uniprocessor. The term "online speed scaling algorithm" is used in various ways, depending on the context. In scheduling books (e.g., [70]), "online" means that tasks arrive at an unknown time, but when they arrive their processing time (execution time) is known. In this thesis we use a different definition: tasks are known beforehand (arrival times and deadlines are known), except for their exact workloads. Only a prediction or an upper bound of the work may be given. Online applications that are considered in this thesis are streaming applications, such as multimedia (e.g., video decoders) and telecommunication applications.

A vast amount of the research on online speed scaling assumes no prior knowledge of the future work. Especially for video decoders, it is popular to use feedback control [7, 23, 64, 79, 85]. Decoded video frames are placed inside a buffer, and the buffer occupancy levels are used to control the speed. When the buffer is depleted, a relatively high speed is used, while a relatively low speed is used when the buffer is almost full. The feedback controller serves two purposes: it (implicitly) predicts the work and steers the speed accordingly. The (P)RA-SS algorithm presented in Chapter 4 separates the work prediction from the calculation of speeds. For the work prediction, any prediction technique can be used. Separating the prediction from control gives additional insights, and makes further energy reduction possible. The latter is illustrated using the approach by Tan et al. [81] that uses predictions. Their approach has lower costs than when using a feedback controller. Furthermore, they note that it is hard to determine the feedback gain.

Using predictions of work to decrease the energy consumption is not new. Several papers [27, 71] predict future work and use it to scale the speed of a video decoder in a greedy fashion. They set the speed to the minimal speed that still executes the tasks in a feasible fashion. The greedy approach by Pillai and Shin [69] is designed for real-time systems that are scheduled using EDF or Rate Monotonic (RM). However, with greedy algorithms, it may happen that the speed is decreased signifi-

cantly, while a later task has to use a much higher speed, which increases the energy consumption significantly. The algorithm presented in Chapter 4 avoids such fluctuations of the speed (see also Section 1.2.1). Furthermore, greedy approaches based on predictions may miss deadlines, while the algorithm from Chapter 4 is robust against mispredictions. A further advantage of the approach from Chapter 4 is that it works for a larger class of applications whereas, for example, the approach by Tan et al. [81] can only be used in a video decoding context to calculate a speed for a single MPEG Group Of Pictures (GOP). Finally the approach in Chapter 4 is less complex than other approaches and it reserves slack for tasks arriving in the future.

The considered problem and solution of Zitterell and Scholl [96] is the most similar to what is presented in Chapter 4. Similar to our approach, they do not assume that the work is known beforehand and determine the optimal speed. They use stochastic information of the tasks to calculate the optimal speed, while we allow arbitrary predictors. However, the method by Zitterell and Scholl does not always find an optimal solution because the constraints in their optimisation model are too restrictive. When this occurs, they must fall back to a (sub-optimal) heuristic. The approach in Chapter 4 does not have this restriction and always finds an optimal solution.

3.5 Global speed scaling of tasks with precedence constraints

Chapters 5 and 6 of this thesis focus on minimising the energy for tasks that have to respect precedence constraints on a global speed scaling system. Several papers have studied global speed scaling under different circumstances, e.g., [24, 44, 51, 66, 92]. The recent survey article by Zhuravlev et al. [95] discusses many energy-cognisant scheduling techniques, but mentions no work that researches the optimal combination of global speed scaling and scheduling. To the best of our knowledge, the research presented in Chapter 5 is the first research that studies the theoretical interplay between optimal scheduling and determining optimal speeds for global speed scaling.

Chapter 5 of this thesis focuses on tasks with precedence constraints that share a common deadline and that can be described using a task graph. Herein, we study the problem $P_M;\text{global}\,|\,a_n = a; d_n = d; \text{prec}\,|\,E$. In the survey article by Kwok and Ahmad [53] and in the article by Tobita and Kasahara [82] references to a lot of these applications are given. A popular benchmark set is the “Standard Task Graph Set” by Tobita and Kasahara [82], which contains both synthetic task graphs and task graphs derived from applications. We use this benchmark set in our evaluation in Chapter 5.

Several authors [13, 38] discuss scheduling and speed selection of general tasks on multiprocessor and multicore systems with a single deadline. In other works [22, 30, 57, 65, 75, 93], tasks with precedence constraints are considered. All these publications have in common that they either focus on local speed scaling or that they use a single speed for the entire run-time of the application. The works that

use a single speed for the entire run-time can only minimise the energy consumption when specific conditions are met. In contrast, our research in Chapter 5 does consider multiple speeds and minimises the energy consumption under all circumstances.

The state of the art with respect to solving the so-called *server problem* (energy minimisation under a deadline constraint) for local speed scaling (P_M; ss | $a_n = a$; $d_n = d$; prec | E) is given by Li [59]. He discusses and evaluates several scheduling algorithms and speed selection algorithms. These algorithms perform notably well for a special class of applications, namely the applications that can be described by using *wide* task graphs. Most of his work focuses on local speed scaling, however a part of the work also describes how to schedule and use speed scaling with the assumption that the entire application uses the same speed. Solving this specific case also gives a solution which can also be used for global speed scaling. We use the case where a single speed is used for the application as a reference in our evaluation in Chapter 5. Our approach varies the multicore chip-wide speed over time and is optimal for global speed scaling, in contrast to using either a single speed for each task or a single speed for the entire application as is done in the work by Li [59].

Few papers study the combination of speed scaling and scheduling in a theoretical way. Chapter 5 presents an extensive study of speed scaling and scheduling of precedence constrained applications on a global speed scaling system. In Chapter 6, the problem is solved where tasks have individual arrival times and deadlines, and a schedule is given (P_M; global | a_n; d_n; sched; prec | E).

3.6 Frame-based real-time systems

Chapter 7 discusses energy minimisation for frame-based real-time systems. These are periodic real-time systems, where tasks are executed within a frame. A frame-based real-time system means that for a *period* T the n-th invocation of each task does not start before the (shared) *arrival time* $a_n = (n-1)T$ and does not finish later than the *deadline* $d_n = nT$. Hence, the active interval of tasks in different frames do not overlap, making speed scaling a relatively simple problem. However, the sleep mode problem is worth investigating because the idle time intervals of adjacent frames can be merged to a single (longer) idle time interval.

The state of the art work by Devadas and Aydin [32] addresses energy minimisation using a combination of speed scaling and sleep modes for frame-based systems. In this work, the authors demonstrate the effectiveness of their models and algorithms experimentally. We used their theoretical results in Chapter 7 to find the optimal speeds in the case speeds can be chosen from a continuous interval. Devadas and Aydin [32] implicitly assume that invocations of tasks start as soon as they arrive. This is a valid assumption since the authors study the interplay of speed scaling and sleep modes for individual frames. In contrast, we allow tasks to start at any time within the frame as long as their deadlines are met, which allows that the global optimum can be found. Although scheduling becomes slightly harder to

implement, we show that by allowing tasks to start at any time within the frame, the energy consumption can be significantly reduced.

The approach from [32] produces a local minimiser, while our approach produces a global minimiser, leading to a lower overall energy consumption. In some situations, their algorithm cannot find any opportunities to use sleep modes and save energy that way, while our approach considers the problem globally and finds every opportunity to save energy and never requires more energy compared to the approach presented by Devadas and Aydin [32]. In addition, we give a solution for the case that only a finite number of speeds is available. A comparison between both approaches shows that our approach can reduce the energy consumption by up to 50%.

Similar work was done in [52], where minimising the energy using speed scaling and sleep modes in presence of multiple tasks that use multiple devices is modelled as an Integer Linear Program (ILP). In contrast to this work, we assume that all devices are active during the execution of any task, and we do not solve an ILP, but give an analytic solution that can be found in polynomial time. The combination of speed scaling and sleep modes has been studied in a stochastic setting by Xu et al. [87], while our approach is deterministic. Our results on sleep modes can also be applied in a stochastic setting.

In the research by Baptiste et al. [19], an optimal sleep mode schedule for aperiodic jobs is found in polynomial time, but only works when there is only an active mode and a sleep mode. In our work, we do not restrict the number of low power modes and also optimise for speed scaling.

3.7 Conclusions

In this chapter we surveyed many power management algorithms that aim to minimise energy under time constraints. These algorithms combine speed scaling and/or sleep modes with energy-aware task scheduling to minimise the energy consumption.

Missing in the literature are approaches for online energy minimisation for real-time systems with agreeable deadlines, energy-aware scheduling to minimise the energy consumption of real-time systems with precedence constraints, and the optimal combination of speed scaling and sleep modes for frame-based real-time systems. These topics are discussed in the subsequent chapters.

CHAPTER 4

Uniprocessor Speed Scaling

ABSTRACT – *Many speed scaling algorithms assume that the work for each task is known. However, when this work is not known beforehand, a so-called online algorithm has to be used. This chapter presents an online algorithm for uniprocessor real-time systems with agreeable deadlines that uses optimal slack time reclamation. This algorithm uses predictions of future workload to determine the optimal speeds, while it remains robust against mispredictions. If the workload is correctly predicted, the presented algorithm calculates speeds that are provably optimal under the given restrictions. For applications with periodic arrival times and deadlines, it attains a near-optimal energy consumption while using easy to obtain predictions. The developed algorithm is compared with algorithms and techniques from the literature for a video decoder workload. Compared to existing methods, our algorithm can reduce the energy consumption by up to 54% for the considered multimedia workloads, where the time complexity of our algorithm is significantly lower.*

4.1 INTRODUCTION

In this chapter, we consider speed scaling for minimising the energy consumption of a uniprocessor system that executes tasks with agreeable deadlines. We develop an approach and evaluate the practical impact of these algorithms. For the design of the algorithms we only assume that the power function is convex. Section 2.6.2 shows that assuming this is not a real restriction and that we may assume convexity generally holds. It is well-known [43, 47, 48, 90] that because of convexity, speed fluctuations have to be avoided to minimise the energy consumption (see Section 2.6.1).

Parts of this chapter have been presented in [MG:3] and [MG:4].

Whereas in the previous chapter mainly *offline* optimisation algorithms were surveyed, the focus of this chapter is on *online* optimisation, meaning that the future workload is not known, but can only be predicted. For the problem that we consider, a *solution* is characterised by a feasible begin time and speed assignment for tasks and the *optimal solution* is a solution that minimises the total energy consumption. While offline optimisation algorithms commonly keep the fluctuations of the speed to a minimum, this is rarely done by online approaches. When real-time behavior is important, online approaches that use predictions still must ensure that deadlines are met when predictions are incorrect. The online algorithm must be robust against such incorrect predictions. If a task finishes before its deadline, the next tasks may use this spare time—slack time—to run at a lower speed to decrease the energy consumption when this is allowed by the arrival times of these tasks. Many online speed scaling algorithms [27, 71] are greedy with regard to this slack time. They use all slack time when it becomes available, or even run at a low speed before slack time is available. Such approaches are in general not optimal since they result in many unnecessary speed fluctuations.

We present algorithms that minimise the speed fluctuations while taking static power into account. These algorithms use predictions of the workload, and since the predictions can be incorrect, special precautions are required to ensure that all deadlines are met. The presented algorithms and solutions are *robust*, meaning that the deadlines are met regardless of the quality of the predictions.

In this chapter, we present two (alternative) robust algorithms for online energy minimisation that are executed each time before a new task is started. The first algorithm, which we call Robust and Adaptive speed scaling (RA-SS), is an algorithm that works for tasks with aperiodic deadlines, while the second algorithm called Periodic RA-SS (PRA-SS) focuses on applications with periodic deadlines, and has a very low complexity. The input to the algorithms is the available slack time, arrival times, deadlines and predictions of the future workload. The algorithms use predictions of future work to calculate the optimal speed. If the predictions are *perfect* (i.e. the predicted workload is the actual workload), the algorithms give the minimal energy consumption while respecting the given constraints (robustness and deadlines). The cost of robustness is marginal: our evaluation shows that the energy consumption, when perfect predictions are used, is only one percentage point higher than that of the solution found using offline optimisation. Furthermore, even when the workload is significantly lower than was predicted, the energy consumption increases only slightly.

The main contributions in this chapter are:

- An optimal algorithm for the offline problem that takes static power into account (Section 4.3).
- Two algorithms, called RA-SS and PRA-SS, to determine the optimal speed by using predictions of future work (Section 4.4).
- These algorithms are robust against mispredicting future work (Section 4.4).

- A derivation that shows that (and reasons why) (P)RA-SS performs close to the theoretical optimum (Section 4.4).
- The algorithms (P)RA-SS do not impose many speed changes, hence the overhead of speed scaling is relatively low (Section 4.4).
- A reasoning that PRA-SS with an extremely conservative prediction of the work—namely, the Worst Case Work (WCW)—performs significantly better than the greedy speed scaling approach that uses perfect predictions (Section 4.4).
- A method (PRA-SS) that can achieve a near-optimal energy reduction for a realistic workload, even with very inaccurate predictions, which means an improvement of up to 51% in terms of energy reduction compared to approaches that are used in a vast amount of the literature.
- An algorithm ((P)RA-SS) that is very robust against inaccuracies in the model.

The remainder of this chapter is organised as follows. Section 4.2 discusses our application model and energy model. Section 4.3 discusses a well-known algorithm, which we use to find an exact solution to the offline energy minimisation problem. Although offline optimisation is not the main topic of this chapter, we use it as the basis for robust and adaptive online optimisation, and we introduce a new algorithm for a specific offline optimisation problem with nonzero static power. We derive a model for online optimisation in Section 4.4, where we present the optimal solution under robustness constraints. The efficiency of our approach is evaluated in Section 4.5 using video decoder workloads. In this section, we show that our results are close to the theoretical minimum (without robustness constraints), and show how our method compares to related work. Section 4.6 concludes with a short summary and discussion.

4.2 Modelling assumptions

The modelling assumptions specific to this chapter are discussed in Section 4.2.1, followed by a discussion about modelling simplifications and their impact in Section 4.2.2.

4.2.1 model

We consider applications that consist of N tasks that are executed on a uniprocessor. Task T_n has w_n work, for which in the online situation this work is not known before the task T_n has finished. The work of all tasks has the same upper bound, the Worst Case Work (WCW), denoted by $w^{\max}$ (i.e. for all n: $w_n \leq w^{\max}$). We assume that predictions $\hat{w}_n \leq w^{\max}$ of the work are available (for all n), for example by using the research discussed in Section 3.4.

In this chapter, we assume that a continuous speed s_n (restricted to a given interval $[s^{\min}, s^{\max}]$) can be assigned to each task T_n. When based on the outcome of some

algorithm a speed $s_n < s^{\min}$ is assigned to a task, setting s_n to $s^{\min}$ is optimal in the restricted case. Because of this, we assume without loss of generality that $s^{\min} = 0$. Furthermore, we do not need to adapt the offline algorithm to take $s^{\max}$ into account. In Section 4.4 we show that the bound $s^{\max}$ is of special importance for online optimisation.

Besides the work w_n, each task T_n has an arrival time a_n and a deadline d_n. The task T_n gets a begin time b_n assigned to it, is executed without interruptions for $e_n = \frac{w_n}{s_n}$ time units and completes its execution at time $c_n = b_n + e_n$. We assume that a task T_n can always meet its deadline when it begins at the deadline of task T_{n-1}, i.e. $d_{n-1} + \frac{w_n}{s^{\max}} \leq d_n$. The chosen begin times and speeds are *feasible* when $a_n \leq b_n$ and $c_n \leq d_n$.

In this chapter, we study real-time applications with agreeable deadlines, and we assume without loss of generality that the tasks T_n and T_m are ordered such that whenever $n \leq m$, it also holds that $a_n \leq a_m$ and $d_n \leq d_m$. It is known that a non-preemptive EDF schedule is optimal for such system [16], hence tasks are scheduled in the order $T_1, T_2, \dots, T_N$. Many streaming applications can be modelled as real-time applications with agreeable deadlines.

In this chapter, we assume that the power function p is convex. For uniprocessor systems this is always a valid assumption, because there is a straightforward workaround when this is not the case (see Section 2.6.2).

Recall that the application begins at some given time t^B, and the power consumption of the processor is accounted for until some time t^C, meaning that the static energy consumption is given by $(t^C - t^B)p(0)$. As we measure the static power until the end of the application or until the deadline of the last task, we take $t^B = a_1$, while for the end time we assume that either $t^C = d_N$ or $t^C = c_N$; both situations are discussed when we present a solution to the problem.

4.2.2 Discussion on simplifications

In practice, many processors only support a discrete set $\{\bar{s}_1, \dots, \bar{s}_K\}$ of speeds, as opposed to continuous speeds. The standard approach is to solve the continuous variant of the speed scaling problem, and then simulate the continuous speeds (see Section 2.6.5). In this chapter, we use another approach, namely using the speed $\bar{s}_{i+1}$ (when $\bar{s}_i \leq s_n \leq \bar{s}_{i+1}$) since this significantly decreases the number of speed changes. In the evaluation we come back to this point, and discuss the implications of using discrete speed scaling.

We assume throughout this thesis that the speed is linearly dependent on the clock frequency. In practice, this is not the case and the execution time is shorter than was predicted (see the discussion in Section 2.3). This assumption has the same effect as taking a conservative prediction for the work; in our evaluation we show that this has little impact on the performance and applicability of our algorithms. The evaluation (Section 4.5.4) contains a discussion on this assumption.

The results in Section 4.5 show that because of the convexity of the power function, our algorithms reduce the number of speed changes significantly. In addition, reducing the number of speed changes has another effect, namely the overhead of changing the speed decreases. The recent article by Park et al. [68] shows that the transition delay overhead for changing the clock frequency is at most 62.68μs [68] on an Intel Core2 Duo E6850. Because of these two reasons, we may assume that the energy overhead of changing the clock frequency is negligible in case of DVFS.

The discussion in Section 4.5.4 shows that our algorithms are only marginally influenced by the assumptions that continuous speeds are available, the assumption that clock frequency scales linearly with the speed, and the assumption that the overheads for changing the speeds are negligible.

4.3 Offline optimisation

This section presents algorithms that solve the offline speed scaling problem with agreeable deadlines. In principle we can use any convex optimisation tool to solve this offline variant of our problem. However, there are a few reasons to develop a tailored method. Firstly, such an approach is specific to our problem, and may therefore be more efficient. Secondly, the tailored approach as presented in the following provides insights that we can use for online algorithms such that they can use slack in an optimal way. More precisely, our online speed scaling algorithms uses the offline algorithms as a subroutine.

With respect to static energy, we consider two sub-problems. Section 4.3.1 considers the case where the static energy consumption has to be taken into account for a given fixed period of time ($t^C = d_N$), therefore the solution does not influence the static energy consumption. In Section 4.3.2 we extend the results from Section 4.3.1 to take static energy into accounted until the last task has finished its execution ($t^C = c_N$).

4.3.1 fixed static energy

In this subsection we assume that the processor is active until the deadline of the last task T_N (i.e. $t^C - t^B$ is constant). This means that the static energy consumption can no longer be influenced by the selected speeds, i.e. we may assume without loss of generality that $p(0) = 0$ [47] (see Section 2.6.3).

For the offline problem we have to minimise the energy consumption of N tasks with agreeable deadlines, such that all deadlines are met. The energy consumption for all tasks together is given by

$$\sum_{n=1}^{N} p(s_n) \frac{w_n}{s_n}.$$

However, not all speeds are allowed. Constraints are required to ensure that tasks do not begin before they arrive, do not finish after their deadline, and that the processor

executes at most one task at a time. This leads to the following optimisation problem with decision variables $s_1, \ldots, s_N$ and $b_1, \ldots, b_N$.

Optimisation Problem 4.1.

$$\min_{\substack{s_1,\ldots,s_N \\ b_1,\ldots,b_N}} \sum_{n=1}^{N} p(s_n)\frac{w_n}{s_n},$$

$$\begin{aligned}
\textit{s.t.}\quad & b_n + \frac{w_n}{s_n} \leq d_n, && \textit{for all } n \in \{1,\ldots,N\},\\
& b_n \geq a_n, && \textit{for all } n \in \{1,\ldots,N\},\\
& b_n + \frac{w_n}{s_n} \leq b_{n+1}, && \textit{for all } n \in \{1,\ldots,N{-}1\},\\
& s_n \leq s^{\max}, && \textit{for all } n \in \{1,\ldots,N\}.
\end{aligned}$$

This problem is a convex problem (the cost function and constraints are convex, see Appendix A), which implies that every local minimiser is also a global minimiser. To find a minimiser, we adopt the "RecursiveSmoothing" algorithm of Huang and Wang [43], resulting in the function OPTSPEED (see Algorithm 4.1). For correctness and optimality of this algorithm, we refer to the article of Huang and Wang [43].

Algorithm 4.1 Optimal speeds.

$(s_x, \ldots, s_z)$ = **Function** OPTSPEED (x,z,t^B,t^C)

  $F = \dfrac{\sum_{i=x}^{z} w_i}{t^C - t^B}$

  $y := \operatorname{argmax}_{j\in\{x,\ldots,z\}} \max\left(t^B + \dfrac{\sum_{i=x}^{j} w_i}{F} - d_j, a_j - t^B - \dfrac{\sum_{i=x}^{j} w_i}{F}\right)$

  if $t^B + \dfrac{\sum_{i=x}^{y} w_i}{F} - d_y > 0$ **then** {y misses its deadline}

    $(s_x, \ldots, s_y)$ = OPTSPEED (x,y,t^B, d_y)

    $(s_{y+1}, \ldots, s_z)$ = OPTSPEED $(y+1,z,d_y, t^C)$

  else if $a_y - t^B - \dfrac{\sum_{i=x}^{y} w_i}{F} > 0$ **then** {y violates arrival}

    $(s_x, \ldots, s_{y-1})$ = OPTSPEED $(x,y-1,t^B, a_y)$

    $(s_y, \ldots, s_z)$ = OPTSPEED (y,z,a_y, t^C)

  else {no task is infeasible}

    $(s_x, \ldots, s_z) = (F, \ldots, F)$

  end if

  return $(s_x, \ldots, s_z)$

The idea behind Algorithm 4.1 is as follows. Unnecessary speed fluctuations have to be eliminated to reach optimality, because otherwise the speeds of consecutive tasks can be replaced by a common speed, leading to a decrease of the energy consumption (see Section 2.6.1 and Figure 1.1d). This means that the speed is only changed when a task arrives or when a task meets its deadline.

Table 4.1 – Characteristics of tasks in Example 4.1.

Task	Arrival time	Deadline	Work	Cumulative work
T_1	0	25	3	3
T_2	10	35	10	13
T_3	20	45	8	21
T_4	30	55	1	22
T_5	40	65	9	31

This idea is implemented by the algorithm as follows. First a candidate solution with a constant speed F is determined for the complete interval, hence F is chosen such that all tasks $(T_1, \dots, T_N)$ are executed between a_1 and d_N. However, in this solution some tasks can miss their deadline or are required to begin too early. To avoid unnecessary speed fluctuations, the task with the greatest deadline/arrival time violation is determined, this task is denoted by T_y. The algorithm enforces the begin or completion time for this task such that it does not violate a constraint anymore. Task T_y with the largest violation of an arrival or a deadline is used to divide the set of tasks into two subsets: the tasks before and the tasks after the violation. The algorithm is then recursively applied to both groups of tasks and the optimal solution follows. The working of the algorithm is clarified by the following example.

Example 4.1 (Optimal solution). *Consider an application that consists of N=5 tasks, with work, arrival times and deadlines as given by Table 4.1. This application is constructed such that the effects of the algorithm are emphasised and the optimal speed fluctuates a lot. For a realistic workload this is typically not the case as we demonstrate in Section 4.5. Figure 4.1 shows the arrival times, deadlines and the optimal solution. The squares on the optimal graph indicate the begin and completion times of tasks. The graph that represents a feasible solution should remain under the graph that gives the arrival times, while it should stay above the graph that represents the deadlines. The speed is the slope of the graph that represents the (optimal) solution. Algorithm 4.1 finds the optimal solution that minimises speed fluctuations, while the speeds are as low as possible.*

4.3.2 variable static energy

In this section, we consider the case that the processor is switched off after the last task is completed, i.e. we choose $t^C = c_N$. This means that it may pay off to increase the speed for the last tasks, such that the processor can be turned off earlier and the static energy consumption decreases.

Before we treat this problem, we restrict our attention to a much simpler problem, namely the uniprocessor problem with a_n=0 for all $n \in \{1, \dots, N\}$ (i.e. arrival times are not taken into account). In this relaxed problem, it is optimal to start

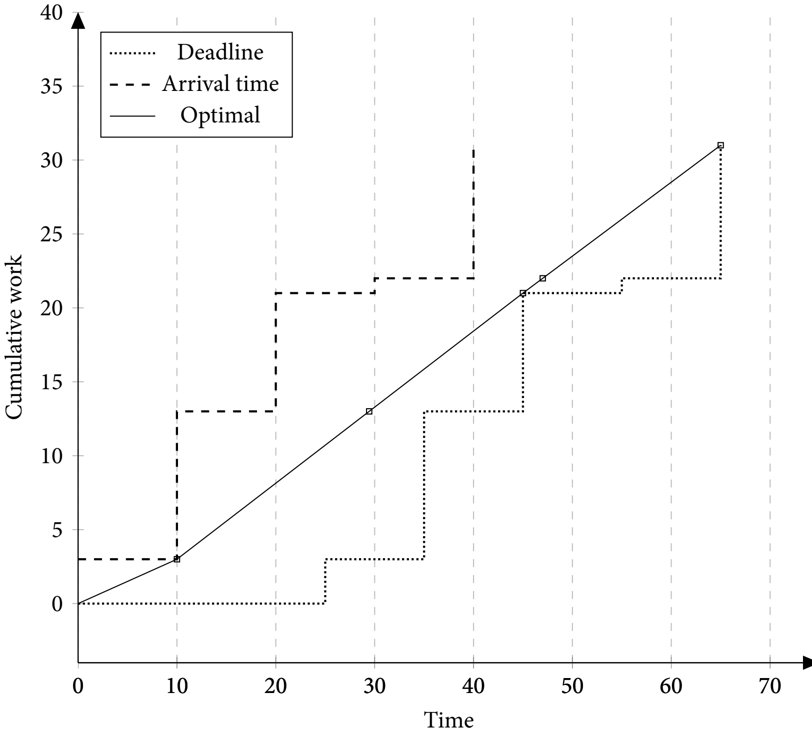


Figure 4.1 – The optimal solution from Example 4.1.

task T_{n+1} immediately when task T_n is finished, since that minimises the static energy consumption. Based on this, the problem becomes as follows.

Optimisation Problem 4.2.

$$\min_{\substack{s_1,\dots,s_N \\ b_1,\dots,b_N \\ t^C}} \quad (t^C - t^B)p(0) + \sum_{n=1}^{N} \left[p(s_n) - p(0)\right] \frac{w_n}{s_n},$$

$$\text{s.t.} \quad b_n + \frac{w_n}{s_n} \le d_n, \qquad \textit{for all } n \in \{1, \dots, N\},$$

$$b_n + \frac{w_n}{s_n} \le b_{n+1}, \qquad \textit{for all } n \in \{1, \dots, N{-}1\},$$

$$s_n \le s^{\max}, \qquad \textit{for all } n \in \{1, \dots, N\},$$

$$b_N + \frac{w_N}{s_N} \le t^C.$$

The cost function gives the total energy consumption for all N tasks, by summing static and dynamic energy consumption. These costs are minimised under the

constraints that all deadlines are met, speeds are below the maximal speed, and the last task does not finish after the end of the application (i.e. the time the static power is accounted for).

In the optimal solution for this problem there is only one idle period during which the processor is turned off, namely an idle period after the last task of the application. Increasing the speed also increases the length of the period during which the processor is off, and with it the static energy consumption decreases. More specifically, no speed $s_n < s^{\text{crit}}$ should be used, because changing s_n to s^{crit} decreases the energy consumption (see Section 2.6.3).

Summarising, for the optimal solution, fluctuations of the speed have to be avoided, and no speeds below s^{crit} are used. The optimal solution to this problem can be determined using Algorithm 4.2 (i.e. by calling $\textsc{optspeed2}(1, N, t^B, t^C)$). This algorithm chooses the lowest speeds, which leads to a schedule respecting the deadlines of the tasks, while avoiding unnecessary speed changes. The intuition behind Algorithm 4.2 is similar to the intuition behind Algorithm 4.1. The algorithm determines a task T_h, and a constant speed s that is used from the current task to task T_h, such that task T_h completes at its deadline and s is maximal.

Algorithm 4.2 Solution to Problem 4.2.

$(s_\ell, \ldots, s_N)$ = **Function** optspeed2(ℓ,N,t^B,t^C)
  $j = \ell$
  while $j \leq N$ **do**
    $h = \max\left(\arg\max_{n\in\{j,\ldots,N\}} \sum_{i=j}^{n} \frac{w_i}{d_n - d_{j-1}}\right)$
    $s_j, \ldots, s_h = \max\left(s^{\text{crit}}, \sum_{i=j}^{h} \frac{w_i}{d_h - d_{j-1}}\right)$
    $j = h + 1$
  end while

The following example illustrates Algorithm 4.2.

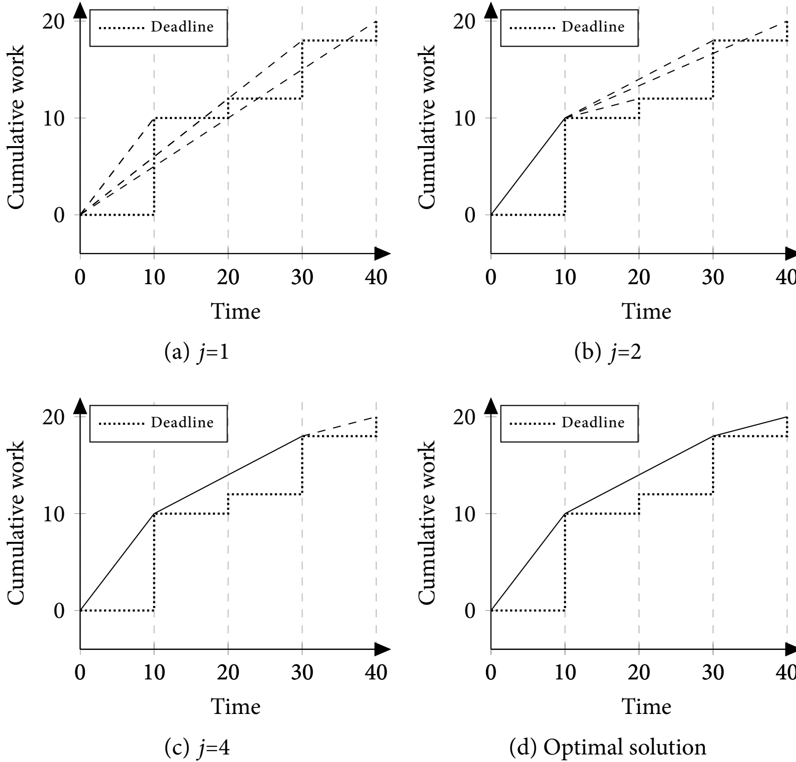
Example 4.2. *In this example we use Algorithm 4.2 to find the optimal solution for the tasks given by Table 4.2.*

In the first iteration, the algorithm starts with task T_1 ($j = 1$), and determines a set of 4 candidate speeds. The criterion for each speed is that when tasks $T_1, \ldots, T_h$ are executed at this speed ($h \in \{1, \ldots, 4\}$), task T_h finishes at its deadline. The three candidate solutions are depicted by the dashed lines in Figure 4.2a, where the slope of the dashed line is the candidate speed. The algorithm picks the highest speed (e.g., the top dashed line, which also has the highest slope), which leads to $h = 1$.

Figure 4.2b and Figure 4.2c show the next two iterations of the algorithm. Finally, Figure 4.2d shows the optimal solution.

Table 4.2 – Task characteristics for Example 4.2.

Task	Deadline	Workload
T_1	10	10
T_2	20	2
T_3	30	6
T_4	40	2



(a) j=1 (b) j=2 (c) j=4 (d) Optimal solution

Figure 4.2 – Iterations for Example 4.2.

Theorem 4.1. *Algorithm 4.2 gives the optimal solution to Optimisation Problem 4.2.*

Proof. We show that the solution $s_1, \ldots, s_N$ (with speed function $s(\tau)$), produced by Algorithm 4.2, is optimal. The optimal solution is unique since the cost function is a strictly convex function that is minimised on a closed convex set.

Assume the theorem is false and the unique optimal solution is given by $\tilde{s}_1, \ldots, \tilde{s}_N$ (we denote the respective speed function by $\tilde{s}(\tau)$). Since using any speed $\tilde{s}_i < s^{\text{crit}}$ is not optimal, we may assume that $\tilde{s}_i \geq s^{\text{crit}}$.

Now, consider the first smallest m such that $s_m \neq \tilde{s}_m$. We consider two possible cases:

(i) $\tilde{s}_m > s_m$:
In this case let n be the smallest $n > m$, such that $\tilde{s}_n < s_n$. Such a value does exist, since otherwise the optimal solution requires more energy than the solution found by Algorithm 4.2, which is a contradiction. It holds that $\tilde{s}_m > s_m > s_n > \tilde{s}_n$, since the sequence $s_1, \ldots, s_N$ produced by Algorithm 4.2 is non-increasing.
For all tasks $T_m, \ldots, T_{n-1}$, we have $\tilde{c}_i < c_i \leq d_i$. We now take a value $t > 0$, such that $\tilde{s}_m t \leq w_m$, and $\tilde{s}_n t \leq w_n$, and $t < \max_{i \in \{m, \ldots, n-1\}} d_i - \tilde{c}_i$.
We consider two small portions of work $w'_m = \tilde{s}_m t$ from task T_m, and $w'_n = \tilde{s}_n t$ from task T_n. For w'_m and w'_n, we change the speeds to $s = \frac{1}{2}\tilde{s}_m + \frac{1}{2}\tilde{s}_n$.
The total execution time of these two portions of work remains $2t$, while the energy consumption becomes

$$\begin{aligned} p(s)2t &= p\left(\frac{1}{2}\tilde{s}_m + \frac{1}{2}\tilde{s}_n\right) \\ &< \frac{1}{2}p(\tilde{s}_m)2t + \frac{1}{2}(\tilde{s}_n)2t \\ &= p(\tilde{s}_m)t + p(\tilde{s}_n)t. \end{aligned}$$

Here, the strict inequality is due to strict convexity of the power function p.
Note that we may choose the portions of work and the speeds such that no deadline is violated, while the energy consumption of the optimal solution decreases. This contradicts the assumption that the solution $\tilde{s}$ is optimal.

(ii) $\bar{s}_m < s_m$:
Note, that in this case the solution from Algorithm 4.2 is constant for tasks $T_m, \ldots, T_n$ (for some $n \geq m$), where $c_n = d_n$. Hence, there must be some time $\bar{b}_m < t \leq d_n$ such that: $\int_{\bar{b}_m}^{t} s(\tau)d\tau = \int_{\bar{b}_m}^{t} \bar{s}(\tau)d\tau$, otherwise task T_n misses its deadline in the optimal solution $\bar{s}$.
But this means that the optimal solution can be improved by using the constant speed s_m on the interval $[b_m, t]$, which contradicts the assumption that the solution is optimal.

Both cases contradict the assumption that Algorithm 4.2 is not optimal, which proves the theorem. □

In the following, we combine Algorithm 4.1 and Algorithm 4.2 to solve the original problem *with* arrival times. In an optimal solution there is some task T_ℓ which is the last task that finishes exactly on the arrival time of the next task, i.e. $c_\ell = a_{\ell+1}$. When no such task exists, we take $\ell = 0$.

Clearly, the processor is active from the start of task T_1 until the arrival of task $T_{\ell+1}$. The static energy consumption during this period is constant. Hence, the results

from Section 4.3.1 can be used to determine the best solution for this period; the optimal speeds for tasks $T_1, \ldots, T_\ell$ can be determined using OPTSPEED $(1,\ell,a_1,a_{\ell+1})$.

For tasks $T_n \in \{T_{\ell+1}, \ldots, T_N\}$ it holds by definition that $c_n > a_{n+1}$. Hence, when calculating the optimal solution, the arrival times do not have to be considered. This means that we can use Algorithm 4.2 to find the optimal speeds.

Now it has become straightforward to calculate the optimal speeds. For a given ℓ, Algorithm 4.1 and Algorithm 4.2 can be used for tasks $T_1, \ldots, T_\ell$ and tasks $T_{\ell+1}, \ldots, T_N$ respectively. The value ℓ is determined by iterating over all possible values of ℓ, namely $0, \ldots, N$, and choosing the value that gives a feasible solution with the lowest cost.

4.4 Online speed scaling

The previous section presented an algorithm that calculates the optimal speeds and begin times (together called the *optimal solution*) for a given workload. In Section 4.4.1 we derive the Robust and Adaptive Speed Scaling (RA-SS) algorithm, which is used in the online situation where only a prediction of the work is available. Based on this, in Section 4.4.2 we present the algorithm Periodic Robust and Adaptive Speed Scaling (PRA-SS), which is an algorithm that is very efficient, and can be applied when the arrival times and deadlines of the tasks have a certain periodic structure.

4.4.1 RA-SS

In the following we derive our RA-SS algorithm, that is executed every time a task T_{k-1} has finished its execution to determine the begin time and speed for the next task T_k. The inputs to this algorithm are arrival times, deadlines and predictions of the work of all future tasks. Although this input can be used to find a solution by using Algorithm 4.1, the resulting solution is not robust. This means that, when the speeds that are found by Algorithm 4.1 are used, deadlines may be missed if the predictions are not completely correct.

To be robust, the deadlines should be met, regardless of the quality of the predictions. This means that, when during the execution of task T_k already $\hat{w}_k$ work has been done and the task is not finished, there should still be enough time to finish the task before its deadline by using a higher "emergency" speed for the remaining work. It is possible to do this because the work of each individual task w_n is never higher than the Worst Case Work (WCW), i.e. $w_n \leq w^{\max}$ for all n.

To realise robustness we define a *robust deadline* $\hat{d}_k(\hat{w}_k)$ that depends on the predicted amount of work $\hat{w}_k$. The predicted work has to be finished before the robust deadline, i.e. $b_k + \frac{\hat{w}_k}{s_k} \leq \hat{d}_k(\hat{w}_k)$. When the actual work is higher than the predicted work, the remaining work of at most $w^{\max} - \hat{w}_k$ work is executed at the speed $s^{\max}$ after time $\hat{d}_k(\hat{w}_k)$. By choosing $\hat{d}_k(\hat{w}_k)$ properly, the deadline is guaranteed to be

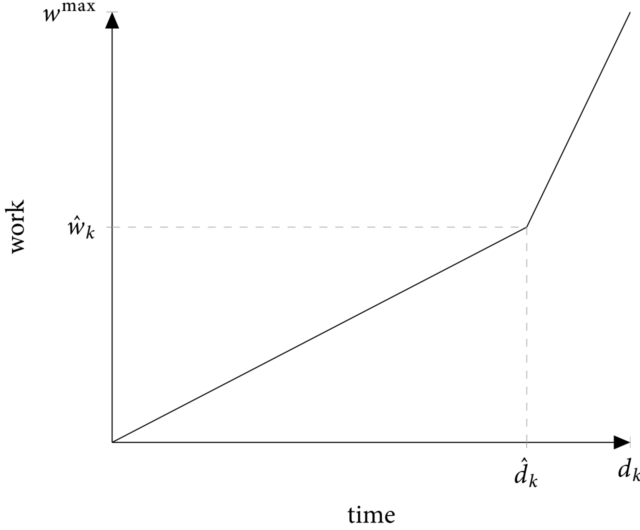


FIGURE 4.3 – Relation between the robust deadline and the deadline of task T_k.

met under worst case conditions, i.e. $\hat{d}_k(\hat{w}_k)$ should satisfy:

$$\hat{d}_k(\hat{w}_k) + \frac{w^{\max} - \hat{w}_k}{s^{\max}} \leq d_k.$$

In our algorithm, we use the highest value that is permitted for $\hat{d}_k(\hat{w}_k)$, such that the perfect predictor minimises the energy consumption, hence the robust deadline is given by:

$$\hat{d}_k(\hat{w}_k) := d_k - \frac{w^{\max} - \hat{w}_k}{s^{\max}}.$$

The RA-SS algorithm introduced below uses this constraint not only for the next task T_k, but for all future tasks T_n (where $n > k$) such that this takes the future where tasks T_n become active into account. This ensures that slack time is also reserved for future tasks, and not greedily consumed by task T_k. Summarising, the following constraints are required:

$$b_n + \frac{\hat{w}_n}{s_n} \leq \hat{d}_n(\hat{w}_n), \quad \text{for } n \in \{k+1, \dots, N\}.$$

The relation between d_k and $\hat{d}_k(\hat{w}_k)$ is illustrated in Figure 4.3.

Our goal is to minimise the energy consumption when a perfect predictor (i.e. $\hat{w}_k = w_k$) is used, while taking the robustness constraints discussed above into account. Such an algorithm is robust against mispredictions, and gives the best performance when the best possible predictor is used. This leads to the following minimisation problem.

Optimisation Problem 4.3.

$$\min_{\substack{s_k,\ldots,s_N\\ b_k,\ldots,b_N}} \sum_{n=k}^{N} p(s_n)\frac{\hat{w}_n}{s_n},$$

$$\begin{aligned}
\text{s.t.}\quad & b_n + \frac{\hat{w}_n}{s_n} \le \hat{d}_n(\hat{w}_n), && \textit{for all } n \in \{k,\ldots,N\},\\
& b_n \ge a_n, && \textit{for all } n \in \{k,\ldots,N\},\\
& b_n + \frac{\hat{w}_n}{s_n} \le b_{n+1}, && \textit{for all } n \in \{k,\ldots,N{-}1\},\\
& s_n \le s^{\max}, && \textit{for all } n \in \{k,\ldots,N\}.
\end{aligned}$$

This optimisation problem has the same form as Optimisation Problem 4.1, but now $\hat{d}_n(\hat{w}_n)$ is used as deadline to make the solution robust against mispredictions. To summarise, RA-SS works as shown in Algorithm 4.3.

Algorithm 4.3 The RA-SS algorithm.

{ Repeated for each task T_k }

1. Use any predictor to predict future work $\hat{w}_k, \ldots, \hat{w}_N$.
2. Calculate s_k by solving Optimisation Problem 4.3 using Algorithm 4.1.
3. Start the execution of task T_k at speed s_k.
4. When $\hat{w}_k$ work has been executed and the task is not finished, switch to the speed $s^{\max}$.

RA-SS has a good performance when the predictions are higher than the actual work, because switching to $s^{\max}$ results in a performance penalty. If it can be guaranteed that $\hat{w}_n \ge w_n$ for all tasks, then the actual deadline can be used instead of the robust deadline.

RA-SS produces an optimal robust solution when the predictions are correct. In that case, it is not required to increase the speed during the execution of a task, and the task uses a constant speed (satisfying the conditions of Theorem 2.1).

4.4.2 PRA-SS

The RA-SS algorithm presented in the previous section requires predictions of all future tasks, which may imply that a significant computational overhead occurs for applications that have many tasks. For some types of applications, we can use the structure to reduce the complexity of the algorithm. In this section we consider applications have tasks with equidistantly spaced (i.e. periodic) arrival times and deadlines. Examples of such applications can for instance be found in specific multimedia and telecommunication applications.

More precisely, an application consists of N tasks, with an unknown workload for each task, and a fixed amount T of time between the arrival times and deadlines of

two consecutive tasks, i.e. for all tasks T_n:

$$a_n = a^0 + nT,$$
$$d_n = d^0 + nT,$$

where the constants a^0 and d^0 represent the phase of the arrival times and deadlines. Note, that it is possible that d^0 is greater than some of the first arrivals $a_1, a_2, \ldots,$ meaning that there can be several tasks that have already arrived, but for which the deadline has not yet passed.

In practice, only predictions of the work of a few future tasks are available. However, the evaluation (Section 4.5) shows that predictions of only a small number of future tasks are sufficient to obtain near-optimal results. We denote the maximum number of future tasks for which predictions are available by ρ. The tasks $T_k, \ldots, T_{k+\rho-1}$ together are called the *prediction window*. For the remaining tasks $T_n \in \{T_{k+\rho}, \ldots, T_N\}$, we use the mean of the work (w^{AVG}) as predictor for the work, i.e. $\hat{w}_n = w^{\text{AVG}}$. For this, we assume that either this mean is known or approximated using a moving average. In the evaluation in Section 4.5 we show that this is a good predictor for work outside the prediction window.

The following theorem proves that the optimal speed does not change during the aggregated tasks.

Theorem 4.2. *For the optimal robust solution of the considered problem it holds that* $s_{k+\rho+1} = \cdots = s_{N-1}$.

Proof. For this proof, we use the function OPTSPEED from Algorithm 4.1. Assume the theorem is not true and some task $T_y \in \{T_{k+\rho+1}, \ldots, T_{N-1}\}$ has a different speed than task T_{y-1} or task T_{y+1} from the same set. Then task T_y was selected by the second statement in the function OPTSPEED that operates on some tasks $\{T_x, \ldots, T_z\}$. The algorithm assigns speed F to this task T_y.

When task T_y was selected because it misses its deadline, it directly follows from the function OPTSPEED that

$$y = \underset{i \in \{x, \ldots, z\}}{\text{argmax}}\ t^B + \frac{\hat{w}_i}{F} - \hat{d}_i(\hat{w}_i). \tag{4.1}$$

Then

$$\begin{aligned}
& t^B + \frac{\sum_{i=x}^{y} \hat{w}_i}{F} - \hat{d}_y(\hat{w}_y) > t^B + \frac{\sum_{i=x}^{\rho+k} \hat{w}_i}{F} - \hat{d}_{k+\rho}(\hat{w}_{k+\rho}) \\
\Rightarrow & \frac{\sum_{i=k+\rho+1}^{y} w^{\text{AVG}}}{F} - Ty > -T(k+\rho) \\
\Rightarrow & \frac{(y-(k+\rho))w^{\text{AVG}}}{F} > T(y-(k+\rho)) \\
\Rightarrow & \frac{w^{\text{AVG}}}{F} - T > 0.
\end{aligned} \tag{4.2}$$

Using this inequality we can derive:

$$
\begin{aligned}
& t^B + \frac{\sum_{i=x}^{y} \hat{w}_i}{F} - \hat{d}_y(\hat{w}_y) \;>\; t^B + \frac{\sum_{i=x}^{N} \hat{w}_i}{F} - \hat{d}_N(\hat{w}_N) \\
\Rightarrow & \left(\frac{w^{\mathrm{AVG}}}{F} - T\right) y \;>\; \left(\frac{w^{\mathrm{AVG}}}{F} - T\right) N \\
\Rightarrow & \, y > N.
\end{aligned}
$$

The first inequality is true due to (4.1), since y is the result of the argmax operator, and the second inequality is true due to (4.2), leading to a contradiction.

When task T_y is selected because it is scheduled to start before it arrives, a contradiction can be derived in a similar way. Since this contradicts the assumption $T_y \in \{T_{k+\rho+1}, \dots, T_{N-1}\}$, the theorem holds. □

Based on this the tasks $T_{k+\rho+1}, \dots, T_{N-1}$ can be aggregated to a single task. More precisely, we aggregate the tasks $T_{k+\rho+1}, \dots, T_{N-1}$ outside the prediction window, to a new task that has the work of tasks $T_{k+\rho}, \dots, T_{N-1}$ together and has the arrival time of task $T_{k+\rho+1}$ and deadline of task T_{N-1}. Aggregating these tasks does not influence the solution since the optimal speed does not change during any of these tasks. Within our revised algorithm, called PRA-SS, tasks are aggregated and RA-SS is executed to find the optimal solution of the resulting set of $\rho + 1$ tasks. Due to the aggregation of tasks, PRA-SS is not quadratic in the number of tasks N, but in the window length ρ. For example, with $\rho = 1$, PRA-SS is executed in constant time, and only a few operations are required.

The following example demonstrates the solution produced by RA-SS and PRA-SS.

Example 4.3 ((P)RA-SS). *We again use the application from Example 4.1. Figure 4.4 shows both optimal solutions, namely the optimal robust solution obtained by RA-SS, and the solution found by PRA-SS. For PRA-SS, we use a prediction window of $\rho = 1$. The figure shows that (P)RA-SS follows the optimal solution closely, but there are slight deviations which is the price for robustness. As RA-SS takes more information into account, it has a better performance than PRA-SS.*

4.5 Evaluation

4.5.1 Application for Evaluation

For the evaluation, we use an MPEG-2 decoder (*libmpeg2* [40]) and consider decoding each video frame to be a separate task. To get sample data that is used to evaluate all considered algorithms, we measured the work of decoding each video frame of 34 different video sequences on a 3GHz Intel Core Duo processor.

Although video decoding is not a hard real-time application, we show that we can add robustness at no significant cost. More precisely, we show that while we maximise the QoS (Quality of Service), the energy consumption is significantly lower

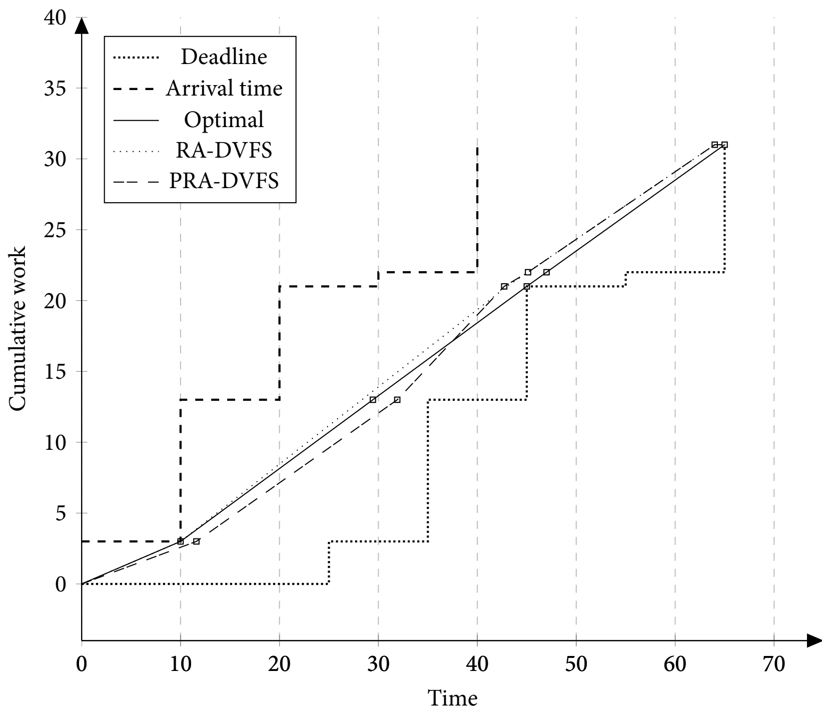


Figure 4.4 – The online solution from Example 4.3.

than that of approaches that do not guarantee that deadlines are met. Because (i) MPEG-2 video decoding is a well-known application, (ii) MPEG-2 video decoding has a fluctuating workload (due to I/P/B-frames), (iii) many standard video sequences are available and (iv) our results are easy to reproduce, this is an excellent application for evaluating our approaches. However, it should be noted that this evaluation using MPEG-2 is only an example, and that the (P)RA-SS approach is very generic and works for many different types of applications.

We have used the CIF and 4SIF videos from [1] and encoded these sequences to MPEG-2. These sequences have a significant workload on a modern computer, but still offer enough opportunities for speed scaling. Because the work for the first few frames cannot be accurately measured, we skip the first Group Of Pictures (GOP) of 12 frames in all video sequences.

For the evaluation we must choose a size of the playback buffer. Video decoders use a buffer for playback that can contain at least the frames within a single GOP (i.e. at least 0.5s for our sequences), for both reordering of out-of-order frames and for flexibility. We set this buffer to one second and adapt the deadline of the first frame accordingly, i.e. $a^0 = 0$ and $d^0 = 1$.

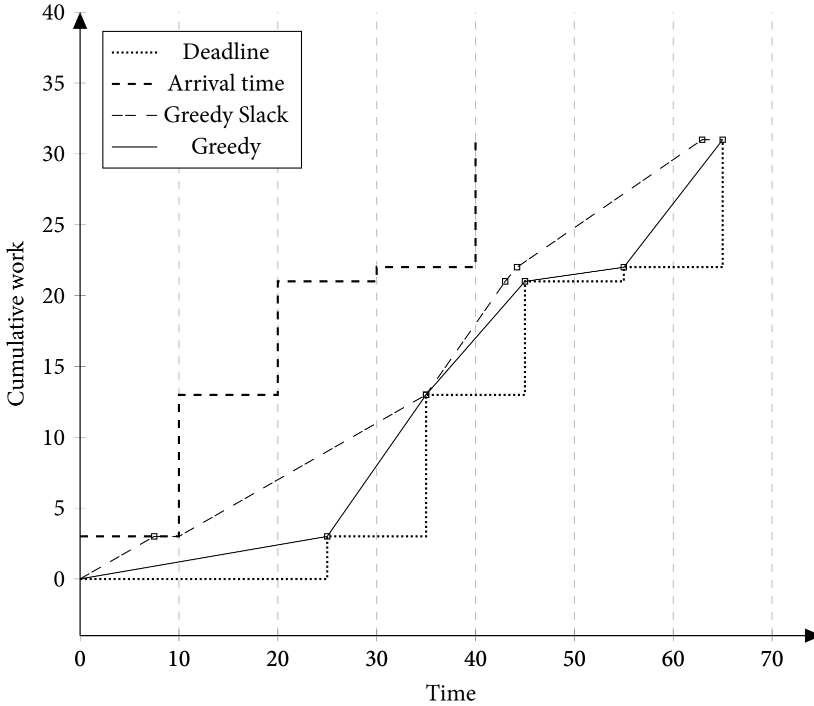


Figure 4.5 – The greedy solution from Example 4.4.

4.5.2 greedy algorithms

To compare our algorithms with related work, we consider two popular classes of approaches to speed scaling, which we refer to as GreedySlack and Greedy .

Many papers (e.g., [67, 93]) use the slack time of a task to decrease the speeds of subsequent tasks by greedily choosing the local optimal speed. The GreedySlack approach is straightforward: all slack time that is created by task T_{k-1} is used to decrease the speed of a task T_k. It calculates the speed for task T_k by

$$s_k = \frac{w^{\max}}{d_k - b_k}.$$

Other papers (e.g., [27, 69, 71]) use predictions of the work to decrease the speed in a greedy fashion, by choosing the lowest speed s_k that is allowed by the deadline of task T_k. We do not compare our approach with individual papers, but compare it to the entire class of such greedy approaches. For fairness, we assume that the predictor is perfect (i.e. $\hat{w}_k = w_k$), which is the goal of the predictors in all these papers. We refer to this approach as Greedy. For Greedy, the speed for a task T_k

is calculated by

$$s_k = \frac{\hat{w}_k}{d_k - b_k}.$$

The results of these techniques are often far from the theoretical optimum given in Section 4.3. The following example illustrates this.

Example 4.4 (Greedy speed scaling approaches)**.** *We again use the application from Example 4.1. When GreedySlack is used, the speeds are initially high because there is no slack, but this is only a start-up effect. Figure 4.5 demonstrates GreedySlack.*

We compare this approach with Greedy, for which we use a perfect predictor. From Figure 4.5 it is clear that both greedy approaches result in many speed fluctuations and are not optimal, although GreedySlack has a reasonable performance for this specific example and performs better than Greedy.

This difference between the two approaches is remarkable, since in many papers the focus is on predicting the work and using an approach similar to Greedy, while a simple approach as GreedySlack attains a much lower energy consumption without using any predictor. Still, both greedy speed scaling approaches—despite their popularity [27, 69, 71]—require significantly more energy than the optimal energy consumption, because the speed fluctuates a lot. This is contrary to what is accomplished by the optimal offline algorithm from Section 4.3.

4.5.3 Evaluation of online algorithms

In our evaluation, Greedy with a perfect predictor is used as a baseline. GreedySlack is less greedy than Greedy, and guarantees that deadlines are always met and does not require any predictor. In addition to this robustness, GreedySlack requires less energy for the tested video sequences, as Table 4.3 shows in the column GreedySlack. Because it is less greedy, the slack time is saved for future tasks and this leads to less fluctuations of the speed, which explains why less energy is consumed. Note that the greedy approaches work reasonably well for the sequences "bridge_close_cif" and "bridge_far_cif". Because these video sequences do not contain a lot of motion, these sequences are not very representative.

The theoretical lower bound for the energy consumption can be found by offline speed scaling. Hereby perfect knowledge of the entire future is used to calculate the optimal speeds. Table 4.3 shows the energy consumption of the optimal solution (offline speed scaling) as percentage of the energy consumption of Greedy in the column "Optimal", which shows that in theory, the energy can be reduced by up to 55% for these sequences. This number increases significantly by enlarging d^0 (meaning that the buffer size is increased). Table 4.3 shows that the optimal solutions require almost no speed changes (about 1 for every 300 frames), while the greedy approaches require that the speed is changed for every one or two frames.

Table 4.3 – Evaluation: Energy consumption % of Greedy (% of frames that change the speed). Lower is better.

Filename	Optimal		PRA-SS (perfect)		PRA-SS (WCW)		Greedy-Slack (perfect)	
akiyo_cif	77	(0.000)	78	(11)	78	(9)	93	(53)
bowing_cif	76	(0.000)	77	(10)	78	(10)	91	(55)
bridge_close_cif	91	(0.001)	91	(4)	91	(4)	95	(47)
bridge_far_cif	91	(0.001)	91	(5)	91	(4)	96	(50)
bus_cif	45	(0.000)	46	(11)	49	(10)	69	(50)
city_4cif	88	(0.000)	89	(22)	89	(21)	97	(80)
city_cif	80	(0.000)	80	(11)	81	(11)	95	(53)
coastguard_cif	81	(0.000)	81	(7)	81	(8)	97	(65)
container_cif	78	(0.000)	78	(7)	78	(8)	94	(55)
crew_4cif	88	(0.000)	89	(45)	89	(46)	97	(87)
flower_cif	77	(0.000)	77	(13)	78	(12)	96	(71)
football_422_4sif	83	(0.003)	83	(34)	84	(32)	96	(86)
football_422_cif	83	(0.000)	83	(7)	84	(8)	96	(69)
foreman_cif	81	(0.000)	81	(8)	81	(8)	97	(62)
galleon_422_4sif	81	(0.000)	82	(17)	82	(17)	97	(75)
galleon_422_cif	77	(0.003)	78	(11)	78	(11)	93	(55)
garden_sif	62	(0.000)	62	(17)	63	(17)	96	(82)
hall_monitor_cif	79	(0.000)	79	(6)	79	(5)	96	(53)
harbour_4cif	89	(0.000)	89	(21)	89	(20)	98	(83)
highway_cif	92	(0.000)	92	(7)	92	(7)	95	(46)
ice_4cif	87	(0.002)	87	(18)	87	(16)	98	(84)
ice_cif	79	(0.000)	79	(5)	79	(5)	97	(61)
intros_422_4sif	83	(0.000)	84	(25)	84	(26)	98	(74)
intros_422_cif	80	(0.003)	81	(11)	81	(11)	95	(60)
mad900_cif	88	(0.002)	89	(6)	89	(6)	96	(54)
mobile_cif	79	(0.004)	80	(14)	82	(15)	93	(63)
news_cif	77	(0.000)	78	(8)	78	(8)	93	(60)
paris_cif	87	(0.002)	87	(7)	88	(7)	94	(50)
sign_irene_cif	86	(0.004)	87	(8)	87	(7)	97	(60)
silent_cif	79	(0.000)	79	(8)	79	(8)	96	(60)
stefan_sif	80	(0.000)	82	(6)	80	(6)	93	(63)
students_cif	86	(0.001)	87	(6)	87	(6)	94	(51)
tempete_cif	77	(0.000)	78	(11)	78	(11)	94	(61)
vtc1nw_422_4sif	81	(0.000)	82	(21)	82	(18)	96	(71)

In the previous section, we mentioned that a single prediction (ρ=1) is often sufficient for a good performance. In our evaluation, we use the moving average of the last twelve frames (the GOP size) for w^{AVG} and we use a perfect predictor $\hat{w}_k$. The result is shown in the column "PRA-SS (perfect)", which shows that with a good prediction for the work of the next frame, PRA-SS obtains results that are close to the theoretical optimum found using the results from Section 4.3 (approximately one percentage point difference). This shows that the additional robustness of PRA-SS does not decrease the performance significantly. The difference is this small, because the cost of robustness is only due to a start-up effect. Only the first few tasks may execute at a too high speed because no slack is available. For later tasks, slack has been built up. Using only a single prediction does not have a negative effect in our evaluation, because for the considered application the average work provides enough information to determine the amount of slack time that has to be kept back for the *entire* future; fine grained predictions for tasks further in the future are not required.

Hence, with a perfect prediction (P)RA-SS has a near optimal performance. However, a perfect prediction is not realistic in practice, hence we want to known what happens when an extremely imperfect and conservative predictor is used. We evaluate this by using the Worst Case Work (WCW) as a prediction (i.e. $\hat{w}_k = w^{\max}$), which is shown in the column "PRA-SS (WCW)" of Table 4.3. The table shows that using $\hat{w}_k = w^{\max}$ works surprisingly well. The main reason for this is that this approach is very conservative with using slack. Slack time is not used at first, but after a short while, there is plenty of slack time available. Hence overpredicting the work decreases the performance initially, but this is again only a start-up effect. We noticed that when d^0 is much smaller, using the WCW as a prediction gives a decreased performance, but remains competitive with GreedySlack and Greedy (with optimal predictions). Furthermore, we used a prediction window of size one (ρ=1), hence PRA-SS has a constant time complexity, and the number of steps the algorithm requires is very small.

Figure 4.6 depicts the evaluation results from Table 4.3. Since the energy consumption is normalised to Greedy, Greedy is shown as 100%. In the optimal solution, the speed is rarely changed (for less than 1% of the frames), and, therefore, these solutions are on the bottom of this plot. In the top right area of this plot, the results for Greedy and GreedySlack are clustered, which means that such approaches require many speed changes and have a relatively high energy consumption.

Our PRA-SS algorithm with the WCW as predictor has a performance that is nearly identical to PRA-SS with a perfect predictor for the work. With a perfect predictor the energy consumption is slightly lower, while the speed is sometimes changed less often than when the WCW is used to predict the work. PRA-SS has an excellent performance for both predictors, compared to Greedy and GreedySlack. The energy consumption is near the theoretical optimum, while for most video sequences the speed is changed for every 5-10 frames. This shows that when RA-SS or PRA-SS are used, the overhead due to the number of speed changes is negligible.

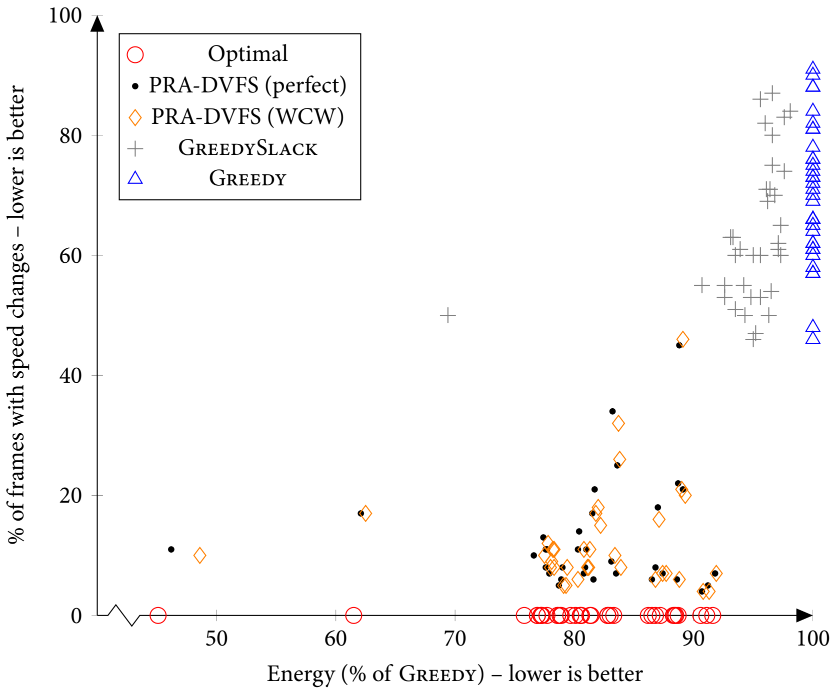


FIGURE 4.6 – Scatter plot with the energy consumption and speed changes for the sequences from Table 4.3.

This evaluation demonstrates that little can be gained by spending a lot of effort at predicting the work of tasks for video decoding applications. It is more important to have a good estimate of the average work of future tasks—which is in practice easy to approximate by using a moving average—and to use this information to decrease the number of speed changes.

4.5.4 SIMPLIFYING ASSUMPTIONS

In Section 4.2.2, we gave some simplifying assumptions that we used to ease the notation and discussion. We assumed for instance that the speed scales linearly with the clock frequency, and that the speed is decreased with respect to the maximal speed. Because of this assumption, the actual execution time is lower than is predicted by the model. However, this is the same as using a work prediction that is too high. But as the evaluation shows, overpredictions of the work have almost no negative effects on the (P)RA-SS algorithm.

Furthermore, we assumed that continuous speeds are available. However, in practice many processors have only a few (discrete) speeds. When we use the nearest discrete speed that is higher than the optimal (continuous) speed, the task finishes

earlier than our algorithm predicts. Again, this is in effect similar to an overprediction of the work, and we have shown that the negative effect of overpredicting the work is relatively small.

In Section 4.2.2, we assumed that the overhead of changing the speed is negligible. Our evaluation (see Table 4.3 and Figure 4.6) shows that the number of speed changes for (P)RA-SS is very low (typically every 5–10 frames). For video, tasks are large (in the evaluation 40ms), and the speed is not changed for every task. The transition delay overhead is at most 62.68μ*s* [68] on an Intel Core2 Duo E6850. Because the transition delay overhead is low and speed changes by our algorithms are rare (approximately every 0.25s–0.5s), the overhead is negligible.

4.6 Conclusions

This chapter discusses both offline and online minimisation of the energy consumption of uniprocessor systems executing tasks with agreeable deadlines. The power consumption consists of both static and dynamic power. For the static power, we considered two different scenarios in Section 4.3: (i) static power is consumed until the deadline of the last task, (ii) static power is consumed until the last task has finished. For uniprocessor systems, the first problem was already solved in the literature, while for the second problem we gave an optimal algorithm in Section 4.3.2. This algorithm is a contribution to the theory of single core speed scaling.

A second aspect considered in this chapter concerns the knowledge about the work of future tasks. Many speed scaling techniques use either a feedback controller to decrease the speed, or use predictions of the work of a task to greedily adjust the speed. The RA-SS and PRA-SS algorithms presented in this chapter generalise both techniques mentioned above: we use predictions of the work, calculate the optimal speed, and use the available slack time as feedback for the next control step. Our algorithms accept predictions for an arbitrary number of tasks, but we have shown that and argued why using only a single prediction is often sufficient for near-optimal energy consumption.

Many papers use predictions in a greedy fashion: each task is executed at a speed that is only locally optimal. Not all greedy approaches are robust, and they require significantly more energy than our algorithms, because the speed is changed very often. We show that greedy approaches—often used in the literature on work prediction for speed scaling—require up to twice the amount of energy required when RA-SS is used. Surprisingly, in the evaluation our approach with a very inaccurate and conservative predictor (namely, the WCW) works significantly better than the greedy approach with a perfect predictor. This demonstrates that it is not always important to have good predictions for speed scaling to work well. Instead, it is crucial to have a good estimation of the *average* work, which is in practice easier to obtain, and to avoid speed fluctuations by distributing slack among future tasks.

We have evaluated our algorithms using video decoder work, and compared the energy consumption and the number of speed changes of our approaches to the

most common speed scaling approaches from the literature. Compared to the approaches from the literature, our approach reduces the energy consumption and the number of speed changes significantly. Our evaluation shows that with PRA-SS, the speed is (on average) changed for every 5-10 frames (i.e. tasks), while for the greedy approaches from the literature it is changed every one to two frames.

Finally, in case the predicted work is always higher than the actual work, the robustness constraints that we introduce can be relaxed (i.e. use d_n in RA-SS instead of $\hat{d}_n(\hat{w}_n)$), and the energy consumption can be further reduced.

Chapter **5**

Scheduling for Global Speed Scaling

Abstract – *While a large part of the theory oriented literature focuses on local speed scaling, where every core's speed can be set separately, this chapter presents a study of the theoretical aspects of the in practice more common global speed scaling, that makes speed changes for the entire chip.*

This chapter shows how to choose the optimal speeds, which minimise the energy for global speed scaling, and it discusses the relationship between scheduling and optimal global speed scaling. Formulas are given to find this optimum under time constraints, including proofs thereof. The NP-hard problem of simultaneously choosing speeds, and a schedule that together minimise the energy consumption, is discussed. A scheduling criterion is derived that implicitly assigns speeds and minimises energy consumption.

Furthermore, this chapter studies the effectiveness of a large class of scheduling algorithms with regard to the derived criterion, and a bound on the maximal relative deviation from the optimum is given. Simulations show that with our techniques an energy reduction of up to 30% can be achieved with respect to state-of-the-art methods.

5.1 Introduction

The previous chapter deals with optimal speed scaling for uniprocessors. Lately, multicore processors have become popular, but theory on optimal speed scaling for real-time applications that are executed on such processors is still missing. There are two important flavours of multicore speed scaling, namely local speed scaling and global speed scaling. Where local speed scaling can set the speed for each

Major parts of this chapter have been submitted for publication [MG:2].

core separately, global speed scaling sets the speed for the entire chip [66]. Global speed scaling occurs more often in practice, because global speed scaling hardware is easier and cheaper (see, e.g., [24, 66]) to implement in a microprocessor than local speed scaling. However, local speed scaling has more freedom in choosing speeds, and therefore, it is often capable of saving more energy than global speed scaling. Furthermore, it requires completely different algorithms to find the optimal schedules and speeds.

In this chapter, we focus on global speed scaling, since it is the most popular technique in practical systems nowadays. Examples of modern processors and systems that use global speed scaling are the Intel Itanium, the PandaBoard (dual-core ARM Cortex A9), IBM Power7 and the NVIDIA Tegra 2 [50, 51, 66].

A popular approach to program multicore systems divides the application into several sequential tasks such that the concurrency becomes explicit. Since these tasks cannot be executed in an arbitrary order, the ordering relations of tasks have to be specified. This is done via a so-called *task graph* in which the tasks are represented by vertices and dependencies between tasks are represented by edges. Furthermore, the applications that we consider have a global time constraint on the completion of the entire application. To be able to execute an application on a multicore system, a schedule for its tasks needs to be specified.

While several important theoretic results on *local* speed scaling and scheduling are given in recent publications [26, 58, 72], to the best of our knowledge, no theoretic results on the interplay of scheduling and global speed scaling are given in the literature. The problem that we study in this chapter is energy minimisation of a global speed scaling system that executes tasks with precedence constraints and a common deadline (P_M; global | $a_n = a; d_n = d$; prec | E). A simple and practice oriented approach would be to use a single speed for the entire application, for example by using the state-of-the-art work by Li [59]. However, for most applications on a global speed scaling system, such an approach does not lead to an optimal solution. We prove that the approach that we present gives an optimal solution. We use the single speed approach as a reference to compare against, to show the significance of the energy gains. In contrast to other approaches, our approach takes parallelism into account.

We furthermore show that to determine both optimal speeds *and* an optimal schedule, the problems should not be considered separately, but as a single optimisation problem. For a given application, modelled as a task graph, we determine a criterion for an optimal schedule and show how to calculate the speeds that minimise energy consumption, while still meeting the application deadline. This scheduling criterion, in contrast to other scheduling criteria for energy minimisation, takes this interplay between scheduling and speed selection into account. It implicitly assigns optimal speeds, and minimises the energy consumption. As many of the well-known scheduling algorithms aim at minimising the makespan (schedule length) of an application, we investigate how well these algorithms perform at minimising our scheduling criterion (which minimises the energy consumption).

To derive our results, we characterise schedules of applications in terms of *parallelism*, which gives for each number of cores the work (e.g., in number of clock cycles) for which exactly that many cores are active. This model abstracts from the actual tasks and their precedence constraints, but still allows for the required analysis. For a given schedule, we calculate the speed for each "number of active cores" and thereby obtain an abstract expression in terms of the parallelism. This expression is substituted into the cost function, to obtain the costs of a schedule with optimal clock frequencies as a function of the *weighted makespan* (to be discussed), which we use as scheduling criterion. For a given makespan and total amount of work of all tasks, we use the weighted makespan to determine the best and worst possible parallelism (i.e. distribution of work over the cores). Using these bounds we derive an approximation ratio for the weighted makespan (that minimises total energy) in terms of the makespan (schedule length). With these results, we derive theoretical results on energy optimal scheduling and we study the energy reduction resulting from scheduling algorithms that were designed to minimise the makespan.

Summarising, this chapter fills a gap in the literature by answering the following research questions:

» Given a schedule, how can we determine the speeds that minimise the energy consumption by using global speed scaling?

» What is the relation between scheduling and optimal global speed scaling?

» How does the presented approach compare against an approach that uses a single speed for the entire application?

The remainder of this chapter is structured as follows. In Section 5.2, mathematical models of the power consumption and applications are given. To be able to derive analytical results on speed selection and scheduling, modelling assumptions must be made, such as neglecting speed scaling overhead and the influence of shared caches. The implications of these assumptions are also discussed in Section 5.2. Using the presented model, Section 5.3 gives an algorithm to calculate the globally optimal speeds and shows that the optimal speeds depend on the amount of parallelism. Whereas Section 5.3 gives optimal speeds for a given schedule, Section 5.4 discusses the theoretic relation between scheduling and optimal speed scaling. This section proves that for two cores an energy minimal solution can be achieved by minimising the makespan. For three or more cores, we show that minimising the makespan does not necessarily minimise the energy consumption. Furthermore, we give a scheduling criterion that, when minimised, *does* minimise the energy consumption. Since many popular scheduling algorithms aim at minimising the makespan, we give an approximation ratio that shows how efficient these algorithms are at minimising the energy consumption. The evaluation in Section 5.5 compares our work to the state of the art (that uses one speed for the entire application), and shows that in theory, for 16 cores the energy consumption can be reduced by up to 44%.

5.2 Model

The application model specific to this chapter is given in Section 5.2.1, while the power model used in this chapter is given in Section 5.2.2. These sections recap the relevant aspects already given in Chapter 2. Since for global speed scaling the model can be simplified after scheduling, we discuss a model that uses the amount of parallelism in Section 5.2.3.

5.2.1 application model

We consider an application running on a Chip Multi Processor (CMP) system with $M > 1$ homogeneous processor cores (the single core case is trivial). We assume that the cores are coupled with highly efficient communication mechanisms (e.g., shared memory) as in [59]. The application consists of N tasks, denoted by $T_1, \dots, T_N$, for which we use the assumption that all tasks arrive at time 0 (i.e. $a_n = 0$ for all n). These tasks are executed without preemption or migration. For each pair of tasks T_i and T_j there may be a precedence constraint denoted $T_i \prec T_j$, meaning that task T_i should be executed before task T_j. In the context of this chapter, we use a single deadline d for the entire application (i.e. $d_n = d$ for all n), or equivalently, for the last task that finishes.

Each task T_n has w_n work (measured, e.g., in clock cycles), leading to a *total work* for the application $W = \sum_{n=1}^{N} w_n$. The length of the schedule in terms of work, called the *makespan*, is denoted by S. To be independent of the speed the makespan is measured in terms of *work*, in contrast to a makespan in seconds that does depend on the speed.

We assume that the speed can be changed at any time, also during execution of a task. As global speed scaling is used, this speed is used for the entire chip. In this way, the speed can be given by a function $s : \mathbb{R}^+ \to \mathbb{R}^+$ that maps a moment in time to a normalised speed.

For our study, some assumptions are made, to be able to calculate optimal speeds and determine a criterion for optimal scheduling. We assume that the speed of the cores scale linearly with the clock frequency, and we neglect the influence of caching and shared resources. If we take as base speed a speed which is larger than or equal to all used speeds, this assumptions does not lead to a violation of deadlines since decreasing speed does not decrease the speed of the memory access. Furthermore, as is common in the literature [26, 32, 38, 75, 89, 90], we do not consider the effect of caches because it is very application specific, and it makes deriving an optimal scheduling criterion infeasible.

If for a given application the speeds are specified, then the execution times of the tasks are known and it can easily be checked whether the application meets its deadline. Since the completion time for the last task depends on the schedule (i.e. the makespan) and on the chosen speeds, the completion time of the last task can be considered to be a speed dependent makespan.

5.2.2 POWER MODEL

We consider a homogeneous CMP system that employs *global speed scaling*, which means that at any time t all M cores use the same speed. The power consumption consists of dynamic power and static power. For a uniprocessor system, the power consumption is given by (see Section 2.3.1):

$$p(s) = \gamma_1 s^\alpha + \gamma_2 + \gamma_3 s.$$

As is argued in Section 2.3.1, we may assume without loss of generality that $\gamma_3 = 0$. We assume that the processor is powered down after the last task is finished.

The power consumption of a multicore system depends on how many cores are *active*, i.e. how many cores have tasks scheduled on them. By using clock gating, the speed (e.g., clock frequency) of an inactive core can be set to zero with little overhead. The *power function* p_m now gives for a given number m of active cores the total power as a function of the speed, which is the dynamic power times the number of active cores m, plus the static power:

$$p_m(s) = \gamma_1 m s^\alpha + \gamma_2,$$

and the respective energy-per-work function is given by:

$$\bar{p}_m(s) = \frac{p_m(s)}{s}.$$

Since $s(t)$ gives the speed for a certain time, $p_m(s(t))$ gives the power consumed by m active cores operating with speed $s(t)$ at time t. Thus, the energy consumption during a time interval $[t_1, t_2]$ in which always m cores are active can be calculated by integrating power over time:

$$\int_{t_1}^{t_2} p_m(s(\tau))d\tau.$$

Note that the power function is strictly convex. We use this fact to prove that, when the number of active cores is constant for some interval, it is optimal to use a constant speed for this interval (equivalent to the uniprocessor case):

Lemma 5.1. *If the number of active cores remains constant (equal to m) during the entire time interval $[t_1, t_2]$, there is a constant speed that is optimal for this interval.*

Proof. The proof of Theorem 2.1 also works for this lemma, because m is constant on the interval $[t_1, t_2]$. □

Following this lemma, we assume without loss of generality in the remainder of this chapter that the speed function is constant on intervals where the number of active cores is constant. This avoids unnecessary switching of the speed and makes the analysis of the problem easier, while the solution remains optimal. This lemma only shows that for the periods where the number of active cores does not change, the speed should be constant, the actual calculation of the optimal speeds is discussed in Section 5.3.

5.2.3 Parallelism based model

The model given above uses the perspective of tasks. In the following, we use an alternative formulation from the viewpoint of parallelism, since this simplifies the analysis in the remainder of this chapter. Furthermore, task-centric reasoning does not make sense for speed selection in a global speed scaling system, since an individual task cannot be assigned its own speed.

In total, there are M cores on which N tasks have to be scheduled, while respecting the precedence constraints. For a given schedule, the relative order in which tasks are executed is unaffected by the speeds that are used, since all cores run at the same speed. In other words, when tasks T_i and T_j finish at the same time for some speed assignment, they finish at the same time for all other speed assignments (i.e. where a higher/lower speed is used, both tasks run equally faster/slower). This is in contrast to local speed scaling, where the relative ordering of tasks can change if not all cores run at the same speed. We use this property to decrease the complexity of calculating the optimal speeds: only the number of active cores at any moment has to be considered. The following corollary is used to simplify the speed scaling problem.

Corollary 5.1. *Consider two time intervals* $[t_1, t_2]$ *and* $[t_3, t_4]$, *during which exactly* m *cores are active. If in an optimal solution both intervals are assigned constant speeds, then these speeds are the same for both intervals.*

Proof. The proof of Theorem 2.1 also works for this corollary, because m is constant on the intervals $[t_1, t_2]$ and $[t_3, t_4]$. □

This corollary implies that the amount of work of a task is no longer relevant after scheduling to determine the optimal speeds, only the number of cores that are active at a certain interval is. For this, let ω_m denote the total duration of work (measured again, e.g., in clock cycles) for which exactly m cores are active. Hence, the *total work* is given by $W = \sum_{m=1}^{M} m\omega_m = \sum_{n=1}^{N} w_n$. Thus, whenever the schedule is known, we only have to consider the values $\omega_1, \dots, \omega_M$, since these contain all the relevant information for solving the problems under consideration. The values $\omega_1, \dots, \omega_M$ together are referred to as the *amount of parallelism* of the schedule of an application. For example, the amount of parallelism if all work W is done by always using all M cores is given by $\omega_M = W/M$ and $\omega_m = 0$ for $m \neq M$, while the amount of parallelism for all work being done on one core is given by $\omega_1 = W$ and $\omega_m = 0$ for $m \neq 1$. The variables $\omega_1, \dots, \omega_M$ fully describe the relevant structure of a schedule that we need for analysis, meaning that we no longer need the individual tasks and their precedence constraints.

Since the optimal speeds only depend on the number of active cores, we use s_m to denote the *speed* that is used when exactly m cores are active. It is important to note, that in contrast with the previous chapter, where s_m denotes the speed of a task, now s_m denotes the speed for *all* intervals with m active cores. Now the

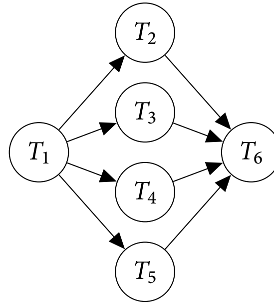


Figure 5.1 – Graph for precedence constraints in Example 5.1.

total energy consumption can be calculated by multiplying the energy-per-work function by the work:

$$E(s_1,\ldots,s_M) = \sum_{m=1}^{M} \bar{p}_m(s_m)\omega_m \tag{5.1}$$

$$= \sum_{m=1}^{M} \left[\gamma_1 m s_m^{\alpha-1}\omega_m + \gamma_2 \frac{\omega_m}{s_m}\right]. \tag{5.2}$$

Since $\frac{\omega_m}{s_m}$ gives the execution time of all the parts of the application that require exactly m active cores, the total execution time of the application is given by:

$$\sum_{m=1}^{M} \frac{\omega_m}{s_m}.$$

The notation and ideas from this section are illustrated in the following example.

Example 5.1. *Consider an application that consists of six tasks ($N = 6$) and with a deadline $d = 100$, that has to be scheduled on a multicore system with $M = 3$ cores. The precedence constraints are given by $T_1 \prec T_i$ and $T_i \prec T_6$ for $i \in \{2,\ldots,5\}$ as depicted in Figure 5.1. The work of the tasks is given by $w = (10,20,15,40,15,10)$, and the total work is given by $W = w_1 + \cdots + w_6 = 110$.*

Figure 5.2 gives a schedule that minimises the makespan. The relevant intervals of the schedule with length $I_1,\ldots,I_6$ (in workload) are presented horizontally, where a new interval begins at the times where a task starts or stops. During intervals 1, 5 and 6, exactly one core is active, hence $\omega_1 = I_1 + I_5 + I_6 = 30$. Similarly it holds that $\omega_2 = I_4 = 10$ and $\omega_3 = I_2 + I_3 = 20$. The makespan is given by the work duration between starting task T_1 and completing task T_6, hence $S = I_1+I_2+I_3+I_4+I_5+I_6 = 60$ or alternatively $S = \omega_1 + \omega_2 + \omega_3 = 60$. The total work can be calculated in terms of ω_m, $W = \omega_1 + 2\omega_2 + 3\omega_3 + = 110$.

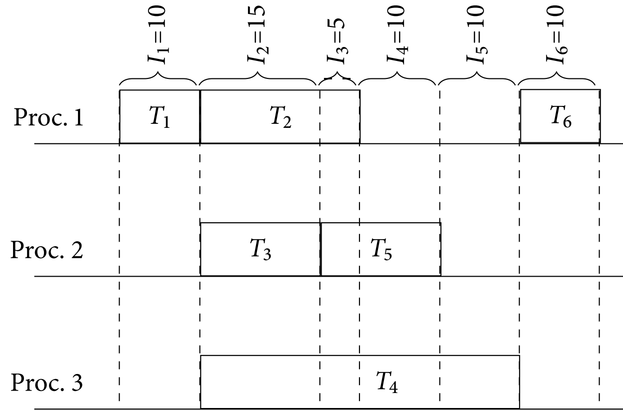


Figure 5.2 – Schedule from Example 5.1.

Corollary 5.1 states that intervals with the same amount of parallelism should receive the same constant speeds in an optimal assignment. Hence, intervals 1, 5 and 6 receive the same speed (namely s_1) since exactly one core is active during these intervals. In contrast, intervals 3 and 4 have respectively two and three active cores and may require different speeds (respectively s_2 and s_3). Although the speed may change while a task is active, changing the speed always coincides with the begin or completion time of some task, which makes it easier to implement in a scheduler. For example, during the execution of task T_4 the speed could be changed three times: after completing tasks T_3, T_2 and T_5.

For illustration purposes, we use (normalised) speeds from the interval $[0,1]$ and use the power function $p_m(s) = ms^3$. This leads to an energy consumption of $E = s_1^2 30 + 2s_2^2 10 + 3s_3^2 20$.

The optimal speeds (as will be shown in Section 5.3) are $s_1 \approx 0.714$, $s_2 \approx 0.567$ and $s_3 \approx 0.495$, with an optimal energy consumption of 36.47. When a single speed is used for the entire application ($s_1 = s_2 = s_3$), the optimal speed is $\frac{S}{d} = 0.6$ (where S is the makespan of 60 and d is the deadline of 100). In that case, the energy consumption is 39.60. The formulas used to calculate the optimal speeds are given in the following section.

5.3 Optimal speeds

In the previous section we presented a model that gives the energy in terms of the amount of parallelism. In this section, we use this model and show how to minimise the total energy consumption for a *given* schedule under the constraint that the single deadline d for the entire application is met.

For now (until Section 5.4), we assume that a schedule is given, hence $\omega_1, \ldots, \omega_M$ are known. The energy consumption for this schedule can be calculated using (5.2), meaning that this is the cost function we have to minimise. The constraint for the minimisation is that the deadline d has to be met, i.e., $\sum_{m=1}^{M} \frac{\omega_m}{s_m} \leq d$. This leads to the following convex optimisation problem.

Optimisation Problem 5.1.

$$\min_{s_1,\ldots,s_M} \sum_{m=1}^{M} \left[\gamma_1 m s_m^{\alpha-1} \omega_m + \gamma_2 \frac{\omega_m}{s_m} \right],$$
$$s.t. \quad \sum_{m=1}^{M} \frac{\omega_m}{s_m} \leq d.$$

Before we solve this problem, we discuss a necessary property of its optimal solution. Assume, that we use a single speed for the entire application, i.e. $s_1 = \cdots = s_M$. Since this solution does not take the amount of parallelism into account, we can improve it by slightly increasing s_1 and slightly decreasing s_m (for some $m > 1$), while keeping the total execution time the same. This implies that for one core the energy consumption is increased, while for m cores the energy consumption is decreased. Due to the superlinear relation (depending on α) between the speed and the energy consumption, there is a bound (depending on α) on how far s_1 should be increased and s_m should be decreased. The following lemma formalises this aspect and shows that there is a fixed factor between the optimal values for s_n and s_m that depends on α, m and n:

Lemma 5.2. *For the optimal solution $s_1, \ldots, s_M$ to Optimisation Problem 5.1, it holds for every pair $n, m \in \{1, \ldots, M\}$, with $\omega_n, \omega_m > 0$ that:*

$$s_n \sqrt[\alpha]{n} = s_m \sqrt[\alpha]{m}. \tag{5.3}$$

Proof. Since s_n and s_m are positive real numbers, there exists an $x > 0$ such that:

$$s_m = s_n x. \tag{5.4}$$

What remains to be proven is that $x = \sqrt[\alpha]{\frac{n}{m}}$. We show that when the sum of the execution times of the work on n and m cores remains fixed, the energy consumption is minimised when $x = \sqrt[\alpha]{\frac{n}{m}}$.

Assume that the sum of the time during which either m or n cores are active is given by the constant $t_{n,m}$. Using (5.4) this term can be expressed by:

$$\begin{aligned} t_{n,m} &= \frac{\omega_n}{s_n} + \frac{\omega_m}{s_m} \\ &= \frac{\omega_n + \frac{\omega_m}{x}}{s_n}. \end{aligned}$$

Now s_n and s_m can be written as a function of x:

$$s_n(x) = \frac{\omega_n + \frac{\omega_m}{x}}{t_{n,m}},$$
$$s_m(x) = \frac{x\omega_n + \omega_m}{t_{n,m}}.$$

Using this, the energy consumption $E_{n,m}$ for the terms that belong to n and m can be written as a function of x:

$$\begin{aligned} E_{n,m}(x) &= \gamma_1 n\left(s_n(x)\right)^{\alpha-1}\omega_n + \gamma_2\frac{\omega_n}{s_n(x)} \\ &\quad + \gamma_1 m\left(s_m(x)\right)^{\alpha-1}\omega_m + \gamma_2\frac{\omega_m}{s_m(x)} \\ &= \gamma_1 n\left(s_n(x)\right)^{\alpha-1}\omega_n + \gamma_1 m\left(s_m(x)\right)^{\alpha-1}\omega_m + \gamma_2 t_{n,m}. \end{aligned}$$

As we consider an optimal solution, the factor x has to minimise $E_{n,m}(x)$ for the given execution time $t_{n,m}$. Because the function $E_{n,m}(x)$ is strictly convex, the critical point of this function is a global minimiser. Hence, the value x that minimises $E_{n,m}$ can be calculated by solving:

$$\begin{aligned} \frac{d}{dx}E_{n,m}(x) &= \gamma_1 n(\alpha-1)\left(s_n(x)\right)^{\alpha-2}\left(-\frac{\omega_m}{x^2 t_{n,m}}\right)\omega_n \\ &\quad + \gamma_1 m(\alpha-1)\left(s_m(x)\right)^{\alpha-2}\frac{\omega_n}{t_{n,m}}\omega_m \\ &= 0. \end{aligned}$$

This gives the minimiser $x_{\min} = \sqrt[\alpha]{\frac{n}{m}}$, and the lemma is proven. □

We use Lemma 5.2 to prove the following theorem.

Theorem 5.1. *The optimal speeds for Optimisation Problem 5.1 are given by:*

$$\mathring{s} = \max\left\{\sqrt[\alpha]{\frac{\gamma_2}{\gamma_1(\alpha-1)}}, \frac{\sum_{m=1}^{M}\omega_m\sqrt[\alpha]{m}}{d}\right\}, \tag{5.5}$$

$$s_m = \mathring{s}\sqrt[\alpha]{\frac{1}{m}}, \quad \text{for } m \in \{1,\ldots,M\}. \tag{5.6}$$

Proof. By Lemma 5.2, any two speeds s_n and s_m are related by a factor that only depends on n and m. We exploit this idea by defining a new variable $\mathring{s} = s_n\sqrt[\alpha]{n}$, for some n with $\omega_n > 0$. Because of (5.6), this implies that $\mathring{s} = s_m\sqrt[\alpha]{m}$, for all $m \in \{1,\ldots,M\}$ with $\omega_m > 0$. Substitution of $s_m = \frac{\mathring{s}}{\sqrt[\alpha]{m}}$ into Optimisation Problem 5.1 gives:

$$\min_{\mathring{s}} \quad \left(\gamma_1 \mathring{s}^{\alpha-1} + \frac{\gamma_2}{\mathring{s}}\right)\left[\sum_{m=1}^{M} \omega_m \sqrt[\alpha]{m}\right],$$
$$\text{s.t.} \quad \frac{\sum_{m=1}^{M} \omega_m \sqrt[\alpha]{m}}{\mathring{s}} \leq d.$$

This is a strictly convex problem in a single variable ($\mathring{s}$), similar to the problem for uniprocessor systems (see Section 2.6.1). The solution is either

$$\mathring{s} = \frac{\sum_{m=1}^{M} \omega_m \sqrt[\alpha]{m}}{d},$$

which is the lowest speed that is allowed by the deadline, or

$$\mathring{s} = \sqrt[\alpha]{\frac{\gamma_2}{\gamma_1(\alpha - 1)}},$$

which is the (unconstrained) minimiser of the cost function, i.e. the "generalised" critical speed $\mathring{s}$ (see Section 2.6.3). Equation (5.5) chooses the highest of the two, to ensure that the deadline is met and the speeds are at least the critical speed. □

The proof of this theorem shows that the critical speed (the speed below which all speeds are energy inefficient, see Section 2.6.3) for m active cores is given by:

$$s_m^{\text{crit}} = \sqrt[\alpha]{\frac{\gamma_2}{m\gamma_1(\alpha - 1)}}. \tag{5.7}$$

5.4 Scheduling and speed scaling

In the previous section, we assumed that a schedule was *given*. This section studies the problem of determining a schedule and a set of speeds that *together* minimise the energy consumption while still meeting the deadline. Section 5.4.1 gives a scheduling criterion for energy optimal scheduling, followed by the relation between this scheduling criterion and the makespan in Section 5.4.2. For the specific situation $M = 2$, Section 5.4.3 shows that minimising the makespan also minimises the energy consumption.

5.4.1 scheduling criterion

It is appealing to first determine a schedule that minimises the makespan (e.g., number of clock cycles) and calculate the optimal speeds for this schedule to solve the overall problem afterwards. This can lead to a suboptimal solution as the following example demonstrates.

Example 5.2. *Consider an application with $T_1, \ldots, T_7$, precedence constraints as depicted in Figure 5.3 and workloads $w = (5.25, 5, 5, 5, 5, 5, 5.25)$. Assume that the*

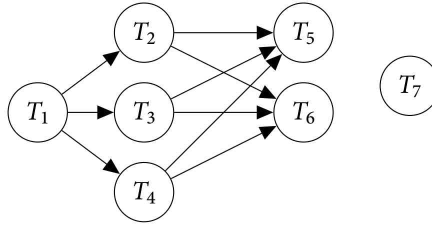


Figure 5.3 – Example task graph.

deadline of this application is $d = 10$ *(in wall-clock time). The unique schedule (up to some reassignments to different cores without changing the timing) that minimises the makespan (to* $S = 15.25$*, in work) for* $M = 3$ *is given in Figure 5.4a. For this schedule,* $\omega_1 = 0$*,* $\omega_2 = 10.25$ *and* $\omega_3 = 5$*. All other schedules with different values for* $\omega_1, \dots, \omega_3$ *have a longer makespan (e.g., require more clock cycles). For illustration purposes we use the dynamic power function* $p_m(s) = ms^3$ *with* $s \in \mathbb{R}^+$*. For this power function, using the optimal speeds leads to an energy consumption of* 81.515.

Figure 5.4b shows an alternative schedule with makespan $S = 15.5$*. For this schedule,* $\omega_1 = 5.25$*,* $\omega_2 = 0$ *and* $\omega_3 = 10.25$ *and the minimal energy consumption is* 80.397*. This shows that the alternative schedule, although it has a longer makespan (e.g., requires more clock cycles), requires less energy. An important reason for this is that in the first schedule the work is distributed over either two or three cores, while in the second schedule the work is distributed over one or three cores.*

This example shows that minimising the makespan does not necessarily minimise the energy consumption. Additional properties, like the amount of parallelism, have influence on the optimal energy consumption. Therefore, in the following, the influence of schedule properties like the amount of parallelism is thoroughly discussed.

A straightforward implication of Example 5.2 is that the problems of scheduling and speed selection should be considered simultaneously. The approach in this section is as follows: first we determine a scheduling criterion, such that when a schedule that minimises this criterion is combined with the optimal speeds for this schedule, the energy consumption is globally minimised. Our criterion (implicitly) takes the optimal speeds into account to break the bi-directional dependence between speed selection and scheduling. Next, we relate this criterion to the makespan, to determine the impact of (minimising) the makespan on the energy consumption.

Consider the energy function given by (5.2). If we substitute (5.6), (5.5), and (5.7), we get:

$$E(\bar{S}) = \begin{cases} \left[\gamma_1 \left(s_1^{\text{crit}}\right)^{\alpha-1} + \frac{\gamma_2}{s_1^{\text{crit}}}\right] \bar{S}, & \text{if } \frac{\bar{S}}{d} \leq s_1^{\text{crit}}; \\ \frac{\gamma_1}{d^{\alpha-1}} \bar{S}^{\alpha} + \gamma_2 d, & \text{otherwise,} \end{cases}$$

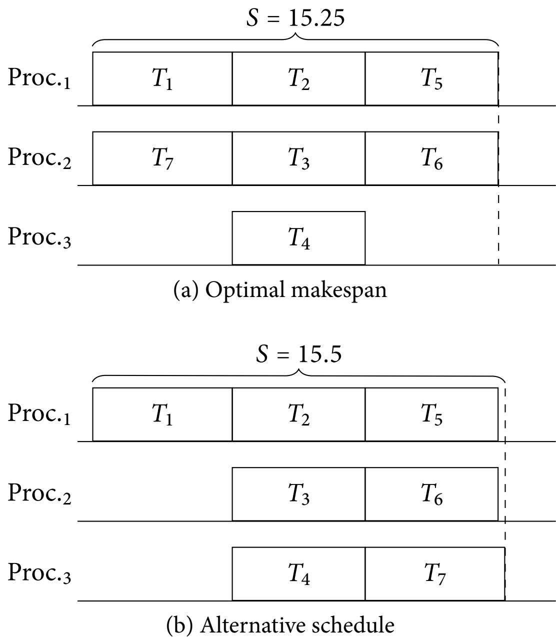


Figure 5.4 – Two schedules for $M = 3$.

where

$$\bar{S} = \sum_{m=1}^{M} \omega_m \sqrt[\alpha]{m}.$$

Here the variable $\bar{S}$, called the *weighted makespan*, is not only used to simplify the notation, but mainly because this is a useful quantity that is important in the remainder of this chapter. The cost function $E(\bar{S})$ is continuous and strictly increasing in $\bar{S}$. Hence, a schedule that minimises $\bar{S}$, also minimises the energy consumption and vice versa. This way, the minimal energy scheduling problem reduces to the problem of finding a schedule that minimises $\bar{S}$. The value $\bar{S}$ is a weighted version of the makespan, where the weights $\sqrt[\alpha]{m}$ are often small and do not differ a lot for different values of m, since in practice α is often close to 3. For this reason, a small makespan S often results in a small weighted makespan $\bar{S}$. We make the relation between S and $\bar{S}$ more precise in the next section.

5.4.2 Using the makespan

Decades of research have been spent on finding scheduling algorithms for minimising the makespan (for a survey, see [53]). However, for global energy minimisation, a scheduling algorithm should minimise the weighted makespan $\bar{S}$ as shown in the previous section.

Repeating and improving on all the known results on the common makespan for the slightly different criterion $\bar{S}$ is not in the scope of this research. Instead, we study how good existing scheduling algorithms are at minimising our criterion $\bar{S}$ and thus the energy consumption.

For an application with total work W, we would like to know how energy efficient a schedule with makespan S can be. Recall that the value of $\bar{S}$ (expressing energy efficiency) solely depends on the amount of parallelism of the schedule described by $\omega_1, \ldots, \omega_M$. Note that also the parallelism, which may be realised for a given application, depends on the precedence constraints of the application. For a given amount of total work W and a makespan S, we are thus interested in the best and worst possible distribution of this work over all cores w.r.t. energy consumption, i.e. the parallelism $\omega_1, \ldots, \omega_M$. This best and worst case distribution then bound the weighted makespan $\bar{S}$ (and thus the energy) of a given set of tasks with total work W that has a schedule of length S. To determine the energy efficiency of arbitrary scheduling algorithms that minimise the makespan, we use these bounds to obtain an approximation ratio for $\bar{S}$ (i.e. energy efficiency) in terms of the makespan.

To determine the best possible $\bar{S}$ for a given schedule with makespan S and total work W we get:

Optimisation Problem 5.2.

$$\begin{aligned}
\min_{\omega_1,\ldots,\omega_M} \quad & \sum_{m=1}^{M} \omega_m \sqrt[\alpha]{m}, \\
s.t. \quad & \sum_{m=1}^{M} m\omega_m = W, \\
& \sum_{m=1}^{M} \omega_m = S, \\
& \omega_m \geq 0.
\end{aligned}$$

Using the concavity (see Section A.1) of $\sqrt[\alpha]{\cdot}$, the optimal solution of this optimisation problem is given by the following lemma.

Lemma 5.3. *The optimal solution to Optimisation Problem 5.2 is given by*

$$\omega_m = \begin{cases} \frac{MS-W}{M-1}, & \text{for } m = 1; \\ \frac{W-S}{M-1}, & \text{for } m = M; \\ 0, & \text{otherwise.} \end{cases} \tag{5.8}$$

Proof. Using elementary algebra, it can be verified that the solution given by (5.8)

is feasible, i.e.:

$$\begin{aligned} \omega_1 + \omega_M &= S, \\ \omega_1 + M\omega_M &= W, \\ \omega_m &\geq 0, \quad \text{for all } m. \end{aligned}$$

Now it remains to show that any other feasible solution leads to higher costs and, thus, cannot be optimal. Consider any other feasible solution $\tilde{\omega}_1, \ldots, \tilde{\omega}_M$. We have to show that:

$$\omega_1 \sqrt[\alpha]{1} + \omega_M \sqrt[\alpha]{M} < \sum_{m=1}^{M} \tilde{\omega}_m \sqrt[\alpha]{m}.$$

For every $m \in \{1, \ldots, M\}$, we define:

$$\lambda_m = \frac{M-m}{M-1} \in [0,1].$$

Because $\sqrt[\alpha]{\cdot}$ is strictly concave and by using the definition of λ_m, we get:

$$\begin{aligned} \lambda_m \sqrt[\alpha]{1} + (1-\lambda_m)\sqrt[\alpha]{M} &< \sqrt[\alpha]{\lambda_m + (1-\lambda_m)M} \\ &= \sqrt[\alpha]{m}. \end{aligned}$$

Using the definition of λ_m, $S = \sum_{m=1}^{M} \tilde{\omega}_m$ and $W = \sum_{m=1}^{M} m\tilde{\omega}_m$ we get by simple algebraic manipulations:

$$\begin{aligned} \sum_{m=1}^{M} \lambda_m \tilde{\omega}_m &= \frac{MS - W}{M-1} = \omega_1, \\ \sum_{m=1}^{M} (1-\lambda_m)\tilde{\omega}_m &= \frac{W-S}{M-1} = \omega_M. \end{aligned}$$

Using this and the strict concavity of $\sqrt[\alpha]{\cdot}$ leads to:

$$\begin{aligned} \omega_1 \sqrt[\alpha]{1} + \omega_M \sqrt[\alpha]{M} &= \sum_{m=1}^{M} \left[\lambda_m \tilde{\omega}_m \sqrt[\alpha]{1} + (1-\lambda_m)\tilde{\omega}_m \sqrt[\alpha]{M}\right] \\ &< \sum_{m=1}^{M} \tilde{\omega}_m \sqrt[\alpha]{m}. \end{aligned}$$

Hence, the choice for $\omega_1, \ldots, \omega_M$ as given by (5.8) minimises the energy consumption and the lemma is proven. □

The lemma shows that the energy consumption for a given workload and makespan is minimised when the maximal allowed work is assigned to M cores and the remainder of the work on a single core. In a similar fashion, we can determine the worst possible values for $\omega_1, \ldots, \omega_M$ (maximising $\bar{S}$) for the situation with makespan S and total work W by solving:

Optimisation Problem 5.3.

$$\begin{aligned}\max_{\omega_1,\ldots,\omega_M} \quad & \sum_{m=1}^{M} \omega_m \sqrt[\alpha]{m},\\ s.t. \quad & \sum_{m=1}^{M} m\omega_m = W,\\ & \sum_{m=1}^{M} \omega_m = S,\\ & \omega_m \geq 0.\end{aligned}$$

The following lemma gives the worst possible values for $\omega_1, \ldots, \omega_M$ for the energy consumption.

Lemma 5.4. *Defining* $k = \lceil \frac{W}{S} - 1 \rceil$, *the optimal solution to Optimisation Problem 5.3 is given by*

$$\omega_m = \begin{cases} S(k+1) - W, & \textit{for } m = k; \\ W - kS, & \textit{for } m = k+1; \\ 0, & \textit{otherwise.} \end{cases} \tag{5.9}$$

Proof. The first part of the proof shows that for an optimal solution, it must hold that $\omega_m = 0$ when $m \neq k$ and $m \neq k+1$. The second part shows that the given solution is the only feasible solution with this property.

Assume that there is a feasible solution $\tilde{\omega}_1, \ldots, \tilde{\omega}_M$ for which $\tilde{\omega}_m > 0$ for some $m \in \{1, \ldots, k-1, k+2, \ldots, M\}$. We show that this solution is not optimal. First we define:

$$\begin{aligned}\mathring{\omega}_k &= \min\left\{\tilde{\omega}_a \frac{k-b}{a-b}, \tilde{\omega}_b \frac{a-k}{a-b}\right\},\\ \mathring{\omega}_a &= \mathring{\omega}_k \frac{k-b}{a-b},\\ \mathring{\omega}_b &= \mathring{\omega}_k \frac{a-k}{a-b}.\end{aligned}$$

for some $a \leq k < k+1 \leq b$, with either $a = m$ or $b = m$.

If we can demonstrate that decreasing $\tilde{\omega}_a$ by $\mathring{\omega}_a$ and decreasing $\tilde{\omega}_b$ by $\mathring{\omega}_b$, while increasing $\tilde{\omega}_k$ by $\mathring{\omega}_k$ improves the solution while keeping it feasible, we prove that the solution $\tilde{\omega}_1, \ldots, \tilde{\omega}_M$ is not optimal.

Using simple algebra, it can be readily checked that $\mathring{\omega}_k = \mathring{\omega}_a + \mathring{\omega}_b$, $k\mathring{\omega}_k = a\mathring{\omega}_a + b\mathring{\omega}_b$, $\mathring{\omega}_k \geq 0$, $\mathring{\omega}_a \leq \tilde{\omega}_a$ and $\mathring{\omega}_b \leq \omega_b$. Then:

$$\begin{aligned}\sqrt[\alpha]{k} &= \sqrt[\alpha]{\frac{k\mathring{\omega}_k}{\mathring{\omega}_k}} = \sqrt[\alpha]{\frac{a\mathring{\omega}_a + b\mathring{\omega}_b}{\mathring{\omega}_a + \mathring{\omega}_b}}\\ &> \frac{\mathring{\omega}_a \sqrt[\alpha]{a} + \mathring{\omega}_b \sqrt[\alpha]{b}}{\mathring{\omega}_a + \mathring{\omega}_b} = \frac{\mathring{\omega}_a \sqrt[\alpha]{a} + \mathring{\omega}_b \sqrt[\alpha]{b}}{\mathring{\omega}_k},\end{aligned}$$

where the inequality is due to the strict concavity of $\sqrt[\alpha]{\cdot}$ and the equalities use the relations given above. Hence,

$$\mathring{\omega}_k \sqrt[\alpha]{k} > \mathring{\omega}_a \sqrt[\alpha]{a} + \mathring{\omega}_b \sqrt[\alpha]{b}. \tag{5.10}$$

As a consequence, we can increase the costs, by decreasing $\tilde{\omega}_a$ by $\mathring{\omega}_a$, decreasing $\tilde{\omega}_b$ by $\mathring{\omega}_b$ and increasing $\tilde{\omega}_k$ by $\mathring{\omega}_k$. It is straightforward to check (using the equalities above) that this new solution still satisfies the given constraints.

Based on the above, we may set $\omega_m = 0$ for $m \neq k$ and $m \neq k{+}1$. Now, for a feasible solution the following equations have to hold:

$$\begin{aligned} \omega_k + \omega_{k+1} &= S \\ k\omega_k + (k+1)\omega_{k+1} &= W. \end{aligned}$$

This system of equations has a unique solution given by (5.9), thus this is the optimal solution and the lemma is proven. □

Lemma 5.3 and Lemma 5.4 give values $\omega_1, \ldots, \omega_M$ that respectively minimise or maximise the energy consumption (i.e. $\bar{S}$) for some fixed makespan S and work W. This gives quantitative bounds to the energy consumption for scheduling algorithms that aim at minimising the makespan S.

In the following, we investigate whether the parallelism $(\omega_1, \ldots, \omega_M)$ that minimises or maximises the energy consumption can be attained in practice. In fact, the next lemma is more general and shows that, for any given desired parallelism $\bar{\omega}_1, \ldots, \bar{\omega}_M$, there exists an application that has an optimal schedule (in terms of makespan) with this parallelism.

Lemma 5.5. *Given a desired parallelism $\bar{\omega}_1, \ldots, \bar{\omega}_M$, there exists an application (characterised by tasks with work $w_1, \ldots, w_N$ and precedence constraints) such that, for some schedule that minimises the makespan, we have $\omega_1 = \bar{\omega}_1, \ldots, \omega_M = \bar{\omega}_M$.*

Proof. We construct an application with $N = M$ tasks, no precedence constraints and work such that $\omega_m = \bar{\omega}_m$ for all m in an optimal solution.

We define the work w_n of all tasks T_n in terms of $\bar{\omega}_m$:

$$w_n = \sum_{m=n}^{M} \bar{\omega}_m.$$

The makespan for this task set is minimised by scheduling task T_i on core i. An example for $M = 3$ is given in Figure 5.5. It remains to show that for this schedule, it holds that $\omega_1 = \bar{\omega}_1, \ldots, \omega_M = \bar{\omega}_M$.

All cores are active for the duration of the task T_M, which gives the workload $w_M = \sum_{m=M}^{M} \bar{\omega}_m = \bar{\omega}_M$, hence $\omega_M = \bar{\omega}_M$. At least $M-1$ cores are active during execution of task T_{M-1} with work $w_{M-1} = \sum_{m=M-1}^{M} \bar{\omega}_m = \bar{\omega}_{M-1} + \omega_M$. After subtracting the part of the work done by M cores from w_{M-1}, we get $\bar{\omega}_{M-1} = \omega_{M-1}$. This argument can be repeated for all other cores $M{-}2, \ldots, 1$. □

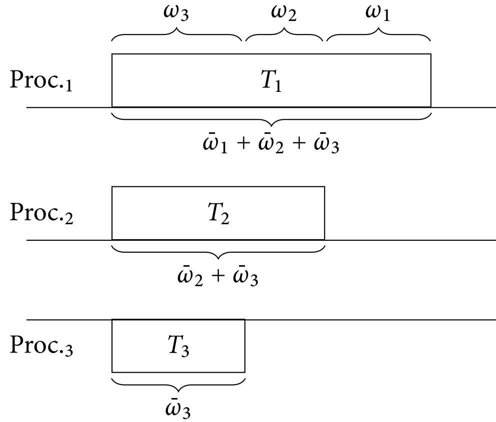


Figure 5.5 – Illustration of Lemma 5.5.

Note, that Lemma 5.5 indicates that for instances without precedence constraints, a lot of variability in the possible parallelism of feasible schedules can be achieved. Adding precedence constraints can only reduce the variability. While Lemma 5.3 and Lemma 5.4 give bounds for the best and worst possible weighted makespan ($\bar{S}$) for a given makespan (S), it is still unclear how well algorithms that aim at makespan minimisation perform at minimising the energy consumption. Given a scheduling algorithm with approximation ratio β (i.e. the algorithm finds schedules with makespan S, for which $S \leq \beta S^*$ with S^* the optimal makespan), we would like to know how well this algorithm performs in terms of energy. More precisely, we are interested in an approximation ratio for $\bar{S}$ of this algorithm, i.e. a value $\bar{\beta}$ such that $\bar{S} \leq \bar{\beta}\bar{S}^*$ where $\bar{S}^*$ is the weighted makespan of the energy optimal schedule. This ratio is given by the following theorem.

Theorem 5.2. *Let a scheduling algorithm A with the approximation ratio β for the makespan be given. This algorithm A has approximation ratio $\bar{\beta}$ for the weighted makespan, i.e. $\bar{S} \leq \bar{\beta}\bar{S}^*$, where $\bar{\beta}$ is given by:*

$$\bar{\beta} = \frac{(\alpha-1)\beta(M-1)}{\alpha\sqrt[\alpha]{(\alpha-1)\beta}(M-\sqrt[\alpha]{M})}\sqrt[\alpha]{\frac{M-\sqrt[\alpha]{M}}{\sqrt[\alpha]{M}-1}}.$$

Proof. We have $\bar{S} \leq S\sqrt[\alpha]{\frac{W}{S}}$, since:

$$\begin{aligned}\frac{\bar{S}}{S} &= \frac{\sum_{m=1}^{M}\omega_m\sqrt[\alpha]{m}}{\sum_{m=1}^{M}\omega_m} \\ &\leq \sqrt[\alpha]{\frac{\sum_{m=1}^{M}\omega_m m}{\sum_{m=1}^{M}\omega_m}}\end{aligned}$$

$$= \sqrt[\alpha]{\frac{W}{S}}.$$

The inequality is due to the finite form of Jensen's inequality (see Appendix A).

The optimal schedule has a value $\bar{S}^*$ for which by Lemma 5.3 holds that

$$\bar{S}^* \geq W(1 - \sqrt[\alpha]{M}) + S^* \frac{\sqrt[\alpha]{M} - M)}{1 - M}.$$

Lemma 5.5 shows that this value $\bar{S}^*$ can occur in practice.

Now the approximation ratio is given by:

$$\begin{aligned} \frac{\bar{S}}{\bar{S}^*} &\leq \frac{S\sqrt[\alpha]{W/S}}{(W(1 - \sqrt[\alpha]{M}) + S^*(\sqrt[\alpha]{M} - M))/(1 - M)} \\ &\leq \frac{\beta}{\sqrt[\alpha]{\beta}} \frac{S^*\sqrt[\alpha]{W/S^*}}{(W(1 - \sqrt[\alpha]{M}) + S^*(\sqrt[\alpha]{M} - M))/(1 - M)}. \end{aligned}$$

The right hand side is a strictly concave function of S^* with a global maximum that is found by taking the partial derivative with respect to S^*. The value $\hat{S}^*$ for which this derivative becomes zero, the maximum, is:

$$\hat{S}^* = (\alpha - 1) W \frac{1 - \sqrt[\alpha]{M}}{\sqrt[\alpha]{M} - M}.$$

Note that this maximum can be pessimistic, since it can be below $\frac{W}{M}$, i.e. outside the interval for S^*.

Now the approximation ratio can be determined by:

$$\begin{aligned} \frac{\bar{S}}{\bar{S}^*} &\leq \frac{\beta}{\sqrt[\alpha]{\beta}} \frac{S^*\sqrt[\alpha]{W/S^*}}{(W(1 - \sqrt[\alpha]{M}) + S^*(\sqrt[\alpha]{M} - M))/(1 - M)} \\ &\leq \frac{(\alpha - 1)\beta(M - 1)}{\alpha\sqrt[\alpha]{(\alpha - 1)\beta(M - \sqrt[\alpha]{M})}} \sqrt[\alpha]{\frac{M - \sqrt[\alpha]{M}}{\sqrt[\alpha]{M} - 1}}. \end{aligned}$$

□

It is straightforward to check that this is an $O\left(M^{1/\alpha^2 - 1/\alpha}\right)$-approximation (see Appendix A). Figure 5.6 shows for 2–10 cores and $\alpha = 3$ how close a makespan optimal schedule ($\beta = 1$) approximates the energy optimal schedule. This shows that when the makespan is minimised and up to six cores are used, $\bar{S}$ is at most 10% higher than its optimal value $\bar{S}^*$. Furthermore, when the makespan is not optimal but has an approximation ratio β, it accounts for a factor $\beta/\sqrt[\alpha]{\beta}$ in our approximation ratio $\bar{\beta}$ that slightly reduces the negative effect of a suboptimal makespan. Note, that Theorem 5.2 works both for the situation with and without precedence constraints (since the parallelism abstracts from this), while the precedence constraints are taken into account in the approximation ratio of the scheduling algorithm.

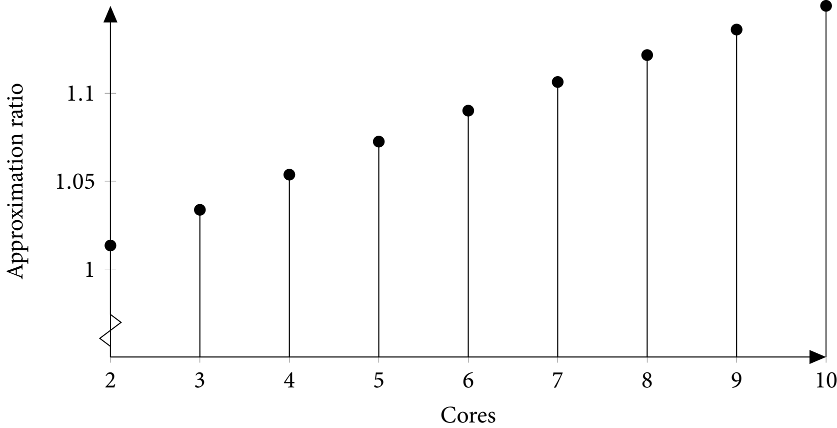


Figure 5.6 – Approximation ratio $\frac{\bar{S}}{\bar{S}^*}$ (for $\alpha = 3$, $\beta = 1$).

5.4.3 TWO CORES

In Section 5.4.1, we have shown that a schedule that minimises $\bar{S}$, also minimises the energy consumption. Furthermore, Example 5.2 shows an optimal schedule that minimises the makespan, but does not minimise $\bar{S}$. However, for two cores minimising the makespan also minimises $\bar{S}$ and thus also the energy consumption, as the following lemma shows.

Lemma 5.6. *When $M = 2$, the energy consumption is a strictly increasing function of the makespan.*

Proof. We have shown that the costs are strictly increasing with $\bar{S} = \sum_{m=1}^{M} \omega_m \sqrt[\alpha]{m}$. Now consider a schedule with some makespan S. Then $\omega_1 = 2S - W$ and $\omega_2 = W - S$, which is obtained by solving $\omega_1 + \omega_2 = S$ and $\omega_1 + 2\omega_2 = W$. Substitution into the definition for $\bar{S}$ gives:

$$\bar{S} = W\left(2 - \sqrt[3]{2}\right) + S\left(\sqrt[3]{2} - 1\right).$$

Since the energy consumption strictly increases with $\bar{S}$, and $\bar{S}$ increases strictly with S, the lemma holds. □

Note that, because of Lemma 5.6, the two-processor scheduling problem can be reduced to our global speed scaling problem in polynomial time; hence the global speed scaling problem is also NP-hard.

When all tasks require the same amount of work and $M = 2$, a schedule that minimises the makespan of an application with precedence constraints can be found in polynomial time [29]. Since the optimal speeds can also be found in polynomial time, the optimal schedule and speeds can be determined in polynomial time.

In case no precedence constraints are present, a Polynomial Time Approximation Scheme (PTAS)[1] exists for finding the minimal makespan [39], hence also for optimal global speed scaling with $M = 2$. We emphasise these results in the following proposition.

Proposition 5.1. *For $M = 2$, the following results hold:*

(i) When $w_i = \mathcal{W}$ for all i (for some constant $\mathcal{W}$), there is a polynomial time solution to the global speed scaling problem.

(ii) When there are no precedence constraints, there is a PTAS for the global speed scaling problem.

5.5 Evaluation

To the best of our knowledge there is no literature in which global speed scaling for tasks with precedence constraints is studied algorithmically. However, there exist several papers on local speed scaling that provide solutions that use a single speed for all tasks (e.g., some algorithms from [59], the state of the art on local speed scaling) and these algorithms can also be used on a system that supports global speed scaling. Therefore, we compare our work to the class of algorithms that use a single speed for the entire run time of the application. Section 5.5.1 compares the dynamic energy consumption of our approach to the single speed case analytically. For this, we use the bound that we obtained in Section 5.4. Then, in Section 5.5.2, we compare both approaches using extensive simulations and demonstrate the effect of static power.

5.5.1 analytic evaluation

First, we focus on the effect of using global speed scaling, and later, in the next subsection we study the influence of static power. In this subsection, we assume that $\gamma_1 = 1$ (normalised dynamic power) while we choose $\gamma_2 = \gamma_3 = 0$. While in Section 5.5.2 we make a comparison based on a set of applications, in this section we compare both approaches analytically.

For our analytic evaluation, we normalise the total work ($W = 1$). When we use a single speed for the entire run time of the application, the optimal speed is $s = \frac{S}{d}$. The corresponding energy consumption is given by:

$$E^{\text{single}} = \frac{S^{\alpha-1}}{d^{\alpha-1}} W.$$

For the best distribution of tasks over cores and when global speed scaling is used optimally, the bound on the energy consumption is given by (see Lemma 5.3):

$$E^{\text{best}} = \frac{\left(W(\sqrt[\alpha]{M}-1) + S^*(M - \sqrt[\alpha]{M})/(M-1)\right)^{\alpha}}{d^{\alpha-1}}.$$

[1]Recall that a PTAS is a polynomial approximation algorithm that can find an approximation that is arbitrarily close to the optimum.

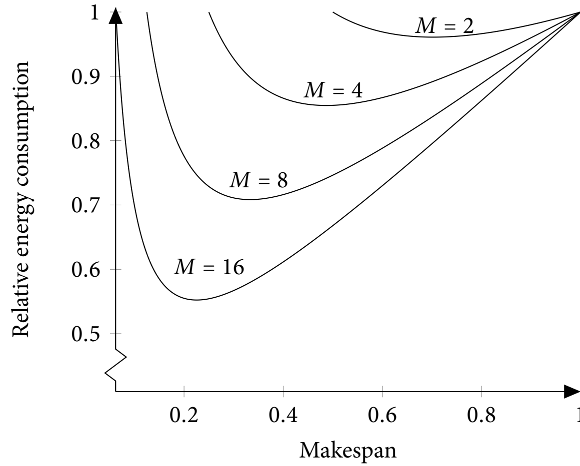


Figure 5.7 – Energy reduction optimal global speed scaling.

The graphs in Figure 5.7 show the ratio ($E^{\text{best}}/E^{\text{single}}$), which is the potential energy saving of optimal global speed scaling. This figure shows that, when optimal global speed scaling is used for 16 cores, up to 44% energy might be saved.

5.5.2 simulations

The above comparison gives theoretic bounds on the possible performance improvements, but it does not take actual schedules into account. We use the Standard Task Graph (STG) set that was created by Tobita and Kasahara [82], to compare our approach—where the speed is varied over time—with an approach where a single speed is used for the entire application.

From the STG set, we use the set of 180 graphs with each 50 tasks with precedence constraints, and schedule these tasks using the Longest Processing Time (LPT) algorithm [34], since this algorithm has a good approximation ratio for makespan minimisation. This good approximation ratio implies that LPT (due to Theorem 5.2) also works well for energy efficient scheduling. Another reason to use LPT is that it aims at maximising the workload assigned to M cores, which is a good strategy as Lemma 5.3 suggests. In our evaluation we compare our global speed scaling approach (aware of static power) to the approach that uses a single speed for the entire application as is used by Li [59] (not aware of static power). We use LPT to schedule 180 STG instances on 2 to 12 cores, use the deadline $d = 2W$, determine the optimal speeds (Lemma 5.1), and calculate the optimal energy consumption for both approaches. For the energy consumption, we normalise γ_1, choose $\gamma_3 = 0$ and consider the cases $\gamma_2 = 0$ (no static power), $\gamma_2 = 0.2$ and $\gamma_2 = 0.4$ to evaluate the effects of taking the static power consumption into account. When static power

Table 5.1 – Ratio (avg/min/max) $E^{\text{global}} / E^{\text{single}}$.

M	Static power (γ_2)		
	0	0.2	0.4
2	0.987 / 0.961 / 0.999	0.982 / 0.978 / 0.988	0.781 / 0.751 / 0.940
3	0.968 / 0.919 / 0.996	0.902 / 0.862 / 0.988	0.664 / 0.599 / 0.940
4	0.948 / 0.887 / 0.994	0.835 / 0.755 / 0.988	0.600 / 0.504 / 0.940
5	0.928 / 0.842 / 0.982	0.790 / 0.673 / 0.988	0.563 / 0.441 / 0.940
6	0.913 / 0.822 / 0.975	0.759 / 0.617 / 0.988	0.539 / 0.400 / 0.940
7	0.900 / 0.795 / 0.975	0.739 / 0.558 / 0.988	0.525 / 0.359 / 0.940
8	0.890 / 0.763 / 0.975	0.726 / 0.541 / 0.988	0.515 / 0.348 / 0.940
9	0.882 / 0.735 / 0.975	0.717 / 0.509 / 0.988	0.509 / 0.326 / 0.940
10	0.877 / 0.722 / 0.975	0.711 / 0.480 / 0.988	0.505 / 0.307 / 0.940
11	0.872 / 0.718 / 0.975	0.707 / 0.468 / 0.988	0.502 / 0.299 / 0.940
12	0.869 / 0.715 / 0.975	0.704 / 0.452 / 0.988	0.500 / 0.289 / 0.940

is present ($\gamma_2 > 0$), the static energy consumption is added to both the energy consumption for global speed scaling (E^{global}), and to the energy consumption for using a single speed (E^{single}).

Table 5.1 shows the average, minimal and maximal ratio $E^{\text{global}}/E^{\text{single}}$. The first column shows that up to 28% energy can be saved when optimal global speed scaling is used instead of a single speed. The other columns show that, when static power is present, the ratio between global speed scaling (aware of static power) and single speed (not aware of static power) gets smaller; up to 72% of energy can be saved by using global speed scaling *and* by taking static power into account.

In a next step we compare the two approaches using the three application graphs from the STG set that are based on real applications, namely, a robotic control application, the FPPPP SPEC benchmark and a sparse matrix solver. In Table 5.2 we present the dynamic energy consumption for these three task graphs by using local speed scaling, global speed scaling and a single speed respectively. In all three cases, we use the LPT schedule. For local speed scaling we obtained numerical solutions using CVX [36] (which required a significant amount of computational time). Table 5.2 shows that by using global speed scaling instead of a single speed, energy savings of more than 30% can be achieved for actual applications (FPPPP, 12 cores). In case a system has support for local speed scaling, even more energy can be saved by using local speed scaling instead of global speed scaling, as should be expected. Note, that while local speed scaling allows for higher energy savings, global speed scaling hardware is cheaper to implement and the most popular of the two approaches in practice.

Table 5.2 – Optimal energy consumption for several applications (single/global/local speed scaling).

M	Application		
	Robot	FPPPP	Sparse
2	684 / 677 / 664	2054 / 2024 / 1972	498 / 496 / 495
3	419 / 399 / 335	1045 / 999 / 953	249 / 243 / 228
4	303 / 279 / 237	671 / 619 / 584	154 / 147 / 139
5	219 / 198 / 163	492 / 435 / 410	113 / 105 / 93
6	170 / 150 / 124	387 / 330 / 306	87 / 78 / 69
7	162 / 137 / 119	324 / 266 / 245	71 / 62 / 52
8	162 / 136 / 101	275 / 217 / 196	60 / 51 / 41
9	162 / 135 / 96	240 / 184 / 165	52 / 43 / 35
10	162 / 135 / 95	215 / 159 / 136	45 / 36 / 29
11	162 / 135 / 95	196 / 140 / 123	40 / 32 / 26
12	162 / 135 / 95	179 / 124 / 107	37 / 28 / 22

5.6 Conclusions

This chapter discussed the minimisation of the energy consumption of multicore processors with global speed scaling capabilities, where the tasks to be scheduled are restricted by precedence constraints. We presented the theoretical relation between scheduling and speed selection, and have shown how a combination of scheduling and speed scaling minimises the energy consumption under a time constraint. The considered problem is a difficult problem, since both scheduling and speed scaling should be taken into account *simultaneously*.

We have shown that the optimal speeds depend on the number of active cores. The optimal speeds for the time periods when n cores are active and the time periods when m cores are active are related by $s_m = s_n \sqrt[\alpha]{\frac{n}{m}}$. We presented formulas that determine the optimal speeds for a given schedule.

As scheduling has a significant influence on the optimal speeds, it also influences the energy consumption. We have shown that for two cores, first determining a schedule that minimises the makespan and then determining the speeds that minimise energy, globally minimises the energy consumption. This result does not hold in general for systems with more than two cores as shown by a counterexample. To deal with this property, we presented a single scheduling criterion (the weighted makespan, $\bar{S}$) that can be used to find an energy optimal schedule.

Computational tests show that by using optimal speeds, up to 30% energy can be saved compared to the state-of-the-art approaches, while the theory shows that even larger improvements are possible (see Figure 5.7). As the number of cores increases, the potential reduction increases significantly, up to 44% for 16 cores.

Chapter **6**

Speed Selection for Global Speed Scaling

Abstract – *In this chapter a transformation from a global speed scaling (multicore) problem to a uniprocessor problem is presented, which uses the amount of parallelism of an application. This reduction allows us to use existing single core algorithms to find the optimal solution for the multicore case. More precisely, and in contrast to the previous chapter, the considered problem has arrival times and deadlines for individual tasks, and it is assumed that a schedule is given.*

6.1 Introduction

In the previous chapter, we studied optimal scheduling and speed scaling of precedence constrained tasks with a common deadline on a global speed scaling system. In contrast, in this chapter each task has an individual arrival time and deadline. The presented solution method does not impose any restrictions on these characteristics of the tasks.

We focus on determining optimal speeds for the entire run-time of an application. Hereby, we assume that a feasible schedule for the tasks of the application is *given*. Hence, in this chapter we study the global speed scaling problem that is formally given by P_M; global | a_n; d_n; sched; prec | E. Since we consider global speed scaling, the concrete assignment of speeds does not influence the relative order of the execution of tasks and therefore does not influence the feasibility of a solution with respect to precedence constraints.

Major parts of this chapter have been presented in [MG:4].

Optimal speed selection is a nontrivial problem, since it may be efficient to finish some tasks early, such that later tasks can run on a lower speed as several authors have demonstrated [43, 90]. This property makes the problem a global problem. Furthermore, the amount of *parallelism* (i.e. number of active cores at a given time) is important when calculating the optimal speeds [26].

In the previous chapter we have shown that it is better to increase the speed when only a few cores are active (increasing the energy only for few cores) and to decrease the speed when more cores are active (decreasing the energy for more cores). While this trade-off has been formally studied in a simplified setting [26], in this chapter we study it in the context of a nontrivial application with arbitrary arrival time and deadline restrictions. For this, we extend some of the results from the previous chapter and use the algorithms from Chapter 4.

The approach presented in this chapter solves the multicore problem by first transforming it to a single core problem (depending on the amount of parallelism), then solving this single core problem, and finally transforming the solution of the single core problem back to the original problem. We combine several existing techniques from the literature to realise our transformation and to solve our problem. Using this transformation and techniques from Chapter 4, a restricted form of the online version of the global speed scaling problem can be solved.

The remainder of this chapter is structured as follows. First, we describe the applications using so called *pieces*, which we introduce in Section 6.2. The reduction from the multicore problem (with a given schedule) to an easier to analyse single core problem is discussed in Section 6.3. We discuss the online global speed scaling problem in Section 6.4. Finally, Section 6.5 presents a summary and a short discussion.

6.2 Pieces

An application consists of multiple tasks, with a given schedule. Each task corresponds to a certain amount of work, can have precedence constraints with other tasks, and (in contrast with what we assumed in the previous chapter) also an individual arrival time and a deadline. We have assumed that a schedule is given, and since we do not need the tasks in the following considerations, but only the structure of the schedule, we do not introduce a formal notation for tasks. Instead, we start with an example of an application.

Example 6.1. *Consider an application that consists of N=8 tasks with precedence constraints. The workload of the tasks $T_1, \dots T_8$ is given by* 4, 2, 3, 6, 2, 2, 2 *and* 2 *respectively. Tasks T_3 and T_8 have the deadlines* 30 *and* 150 *respectively, tasks T_3, T_4 and T_8 have arrival times* 19, 5 *and* 140 *respectively, the other tasks have no deadline and are available from the beginning.*

In the context of this example, the exact precedence constraints are (as for this whole chapter) not relevant, only the fact that they create "gaps" in the schedule. A possible

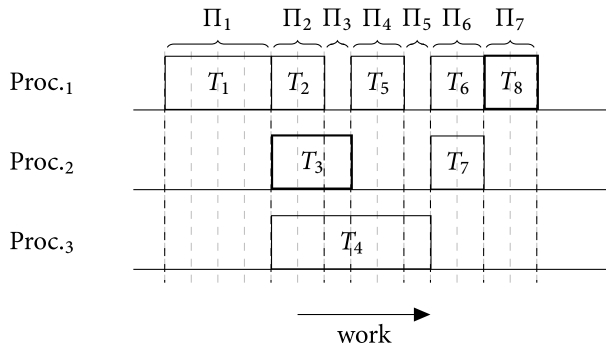


Figure 6.1 – A feasible schedule for Example 6.1.

feasible schedule for the application is depicted in Figure 6.1, in which the tasks with a deadline are highlighted.

In general, we assume (without loss of generality) that the given schedule does not contain any period where *all* cores are idle. Since only static power is consumed during these processor-wide idle periods, our assumption does not change the optimal solution.

Note, that only at the start or completion of the execution of a task, the number of active cores can change. Furthermore, timing constraints such as arrival times or deadlines are also related to these start and completion times. Therefore, we can focus on work intervals (a, b) where no tasks start or complete their execution.

The following corollary shows that it is optimal to use a single constant speed within each of these intervals.

Corollary 6.1. *Given work interval (a, b) during which no task starts or ends. Then there is an optimal speed function $s(t)$ that is constant during the execution of the work in $[a, b]$.*

Proof. This corollary is a direct consequence of Lemma 5.1. □

Because the optimal speed is constant on the interval $[a, b]$ as specified in Corollary 6.1, we subdivide the schedule into such intervals. We choose these intervals such that they are as large as possible and call these intervals *pieces*.

Definition 6.1 (Piece). *A piece is a maximal interval $[a, b]$ such that no task starts or finishes in (a, b).*

A given schedule uniquely subdivides into pieces. Let N be the number of these pieces and let the n-th piece be denoted by Π_n. In the following, we show that algorithmically tasks and pieces are very similar.

Let m_n denote the number of active cores within piece Π_n (this number is constant during Π_n since no task starts or ends its execution during this piece). Furthermore, let the workload of a single core within piece Π_n be denoted by w_n. Since all m_n cores have this amount of work w_n, in total $w_n m_n$ work is executed for this piece. To ease the discussion, we call w_n the work of piece Π_n. The total work W obtained by summing the work of all tasks, can now also be expressed in terms of pieces by $W = \sum_{n=1}^{N} w_n m_n$.

Let the actual begin time of piece Π_n be denoted by b_n and its completion time be denoted by c_n. The actual values for b_n and c_n depend on the chosen speeds of the pieces $\Pi_1, \ldots, \Pi_n$, which is discussed later. Furthermore, for each piece Π_n we can assign an arrival time a_n, which is the latest arrival time among all tasks that start at the beginning of this piece. This makes sure that the piece is not started before the arrival time of any task in this piece. Similarly, for piece Π_n a deadline, denoted by d_n, can be derived; this is the earliest deadline of all tasks that are completed at the end of this piece.

Example 6.2 (Continued from Example 6.1). *In Figure 6.1, also the subdivision of the schedule from Example 6.1 into seven pieces* ($\Pi_1, \ldots, \Pi_7$) *is given. A new piece starts whenever a task starts or finishes its execution. The work of the pieces is given by* $w = (4, 2, 1, 2, 1, 2, 2)$. *The number of active cores is given by* $m = (1, 3, 2, 2, 1, 2, 1)$. *Piece* Π_3 *has deadline* $d_3 = 30$ *(deadline of task* T_3*) and piece* Π_7 *has deadline* $d_7 = 150$ *(deadline of task* T_8*). The arrival time of piece* Π_2 *is* $a_2 = 19 = \max\{19, 5\}$. *For piece* Π_7, *the arrival time is* $a_7 = 140$.

During the execution of piece Π_n the power function p_{m_n} is used, since during the entire execution of piece Π_n exactly m_n cores are active. To determine the energy consumption, we consider the static and dynamic power separately.

Based on Corollary 6.1, we assign a constant speed s_n to each piece Π_n. For a given speed assignment s_n to piece Π_n ($n \in \{1, \ldots, N\}$), the dynamic energy consumption of piece Π_n is the power consumption ($m_n \gamma_1 s_n^\alpha$) times the duration of piece Π_n at the chosen speed $\left(\frac{w_n}{s_n}\right)$ leading to a power consumption of $m_n \gamma_1 s_n^{\alpha-1} w_n$. To obtain the total dynamic energy consumption we have to sum this over all pieces.

Now the total energy consumption can be expressed in terms of pieces and consists of the static and dynamic energy.

$$E = \gamma_2(t^C - t^B) + \sum_{n=1}^{N} m_n \gamma_1 s_n^{\alpha-1} w_n,$$

where t^B and t^C denote the times at which the static power consumption starts to be accounted for and stops to be accounted for respectively.

6.3 Optimisation model

Based on the discussion in the previous section, the problem considered in this chapter reduces to energy minimisation, under constraints such as ordering constraints (piece Π_n is executed before piece Π_{n+1}), arrival times and deadlines. More precisely, we get the following mathematical optimisation problem.

Optimisation Problem 6.1.

$$\min_{\substack{s_1,\dots,s_n \\ b_1,\dots,b_N \\ t^C}} \gamma_2(t^C - t^B) + \sum_{n=1}^{N} m_n \gamma_1 s_n^{\alpha-1} w_n, \tag{6.1}$$

$$\text{s.t.} \quad b_n + \frac{w_n}{s_n} \le d_n, \quad \text{for all } n \in \{1,\dots,N\}, \tag{6.2}$$

$$b_n \ge a_n, \quad \text{for all } n \in \{1,\dots,N\}, \tag{6.3}$$

$$b_n + \frac{w_n}{s_n} \le b_{n+1}, \quad \text{for all } n \in \{1,\dots,N-1\}, \tag{6.4}$$

$$b_N + \frac{w_N}{s_N} \le t^C. \tag{6.5}$$

The energy consumption is given as a cost function (6.1), which has to be minimised. The constraint (6.2) enforces that all pieces meet their deadline, (6.3) ensures that pieces do not begin before their arrival time, and (6.4) enforces that piece Π_n is finished before piece Π_{n+1} starts. The last constraint (6.5) makes sure that the application is not finished after the time for which the static energy consumption is accounted for.

For this problem, standard approaches from the literature cannot be used, due to the multicore aspect with weights m_n in the cost function that result from the number of active cores. We rewrite this problem to an easier to analyse problem. For this, we substitute the variables for speeds and work. This substitution cancels out some terms, and the values m_n—making the problem a multicore problem—disappear from the equations. The idea behind the substitution is based on the ratio $\sqrt[\alpha]{m_a/m_b}$ between optimal speeds of a multicore processor for pieces with different numbers of active cores, described in the previous chapter, and the transformation is inspired by several papers [54, 89].

The substitution of variables is as follows.

$$\mathring{s}_n = s_n \sqrt[\alpha]{m_n}, \tag{6.6}$$

$$\mathring{w}_n = w_n \sqrt[\alpha]{m_n}. \tag{6.7}$$

The variables $\mathring{s}_1,\dots,\mathring{s}_N$ can be interpreted as the speeds that take the optimal ratio $\sqrt[\alpha]{m_a/m_b}$ between optimal speeds (depending on parallelism) into account. The transformation of the work to the variables $\mathring{w}_1,\dots,\mathring{w}_N$ is chosen such that the execution time remains the same.

Substitution into the cost function (energy) gives:

$$E = \gamma_2(t^C - t^B) + \sum_{n=1}^{N} \gamma_1 \mathring{s}_n^{\alpha-1} \mathring{w}_n,$$

while substitution into the deadline constraint gives:

$$b_n + \frac{w_n}{s_n} = b_n + \frac{\mathring{w}_n}{\mathring{s}_n} \leq d_n.$$

The other constraints can be transformed similarly. This leads to the following optimisation problem:

Optimisation Problem 6.2.

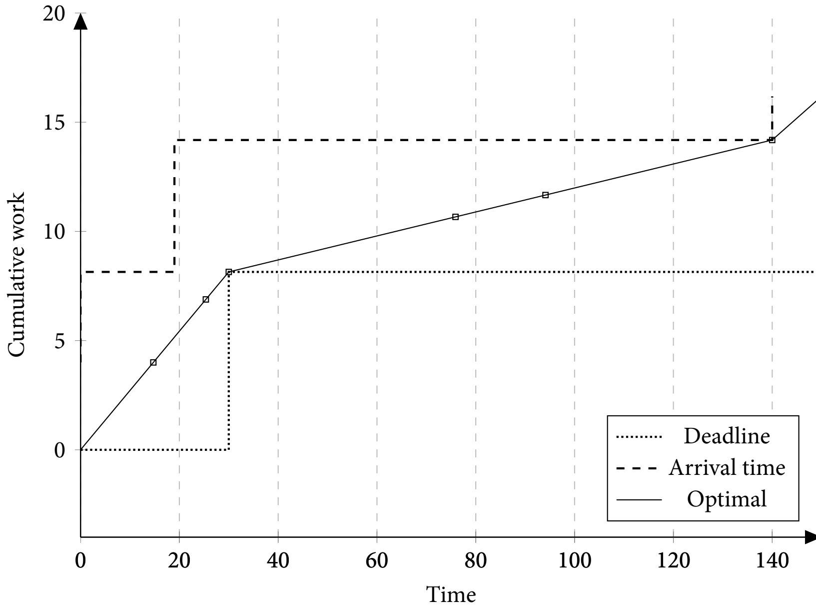
$$\min_{\substack{\mathring{s}_1,\ldots,\mathring{s}_n \\ b_1,\ldots,b_N \\ t^C}} \gamma_2(t^C - t^B) + \sum_{n=1}^{N} \gamma_1 \mathring{s}_n^{\alpha-1} \mathring{w}_n,$$

$$\begin{aligned}
\text{s.t.} \quad & b_n + \frac{\mathring{w}_n}{\mathring{s}_n} \leq d_n, && \textit{for all } n \in \{1,\ldots,N\}, \\
& b_n \geq a_n, && \textit{for all } n \in \{1,\ldots,N\}, \\
& b_n + \frac{\mathring{w}_n}{\mathring{s}_n} \leq b_{n+1}, && \textit{for all } n \in \{1,\ldots,N-1\}, \\
& b_N + \frac{\mathring{w}_N}{\mathring{s}_N} \leq t^C.
\end{aligned}$$

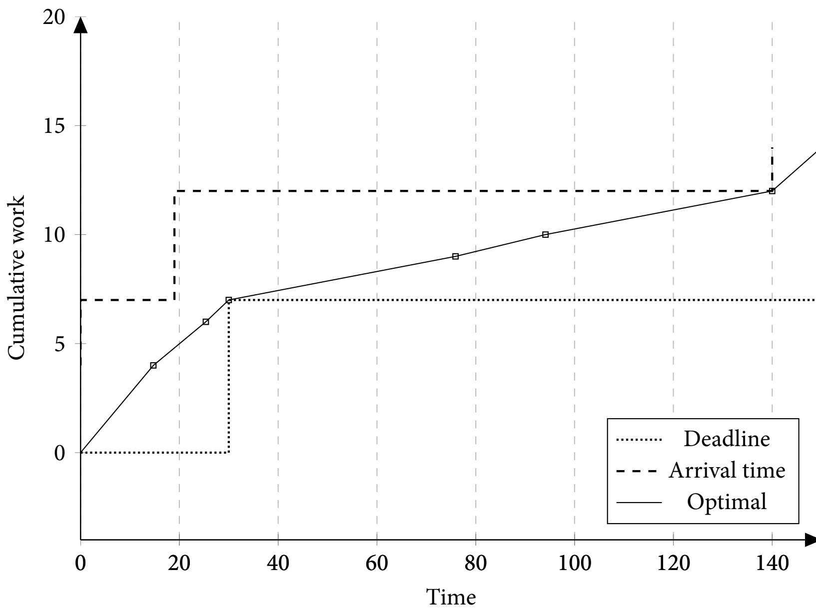
The nice property of Problem 6.2 is that it has the same form as the problem of energy minimisation for a real-time system with agreeable deadlines that was discussed in Chapter 4 (Optimisation Problem 4.1). In this uniprocessor problem, the variable $\mathring{w}_n$ is the work of task $\mathring{T}_n$, while $\mathring{s}_n$ is the optimal speed of task $\mathring{T}_n$. Furthermore, deadlines and arrival times are again given by a_n and d_n and the execution order is predetermined. As this relation is used throughout this chapter, we give a formal definition.

Definition 6.2 (Equivalent uniprocessor problem). *For a given schedule, the multicore global speed scaling problem with N pieces ($\Pi_1, \ldots, \Pi_n$) is equivalent to the uniprocessor problem with N tasks ($\mathring{T}_1, \ldots, \mathring{T}_n$), where each piece Π_n is represented by a task $\mathring{T}_n$ with work $\mathring{w}_n = w_n \sqrt[\alpha]{m_n}$, speed $\mathring{s}_n = s_n \sqrt[\alpha]{m_n}$, and the same completion times and begin times.*

Based on this, Problem 6.1, can be transformed to an equivalent uniprocessor problem. This problem is then solved and the resulting solution is transformed back to obtain the optimal speeds and begin times for the pieces. In the following, we give an example of this transformation and solution technique.



(a) Optimal speeds for the equivalent uniprocessor problem



(b) Optimal speeds for the multicore problem

Figure 6.2 – Optimal speeds for Example 6.3.

Example 6.3 (Continued from Example 6.2). *For illustration purposes, we use the power function* $p_m(s) = ms^3$. *The pieces with workload* $w_1 = 4$, $w_2 = 2$, $w_3 = 1$, $w_4 = 2$, $w_5 = 1$, $w_6 = 2$ *and* $w_7 = 2$, *together with the parallelism described by* $m_1 = 1$, $m_2 = 3$, $m_3 = 2$, $m_4 = 2$, $m_5 = 1$, $m_6 = 2$ *and* $m_7 = 1$ *are transformed to* $\mathring{w}_1 = 4$, $\mathring{w}_2 = 2\sqrt[3]{3}$, $\mathring{w}_3 = \sqrt[3]{2}$, $\mathring{w}_4 = 2\sqrt[3]{2}$, $\mathring{w}_5 = 1$, $\mathring{w}_6 = 2\sqrt[3]{2}$ *and* $\mathring{w}_7 = 2$. *Hence, the global speed scaling problem with 7 pieces is now transformed to the equivalent uniprocessor problem with 7 tasks.*

We use Algorithm 4.1 to obtain the optimal speeds $\mathring{s}_1 = \mathring{s}_2 = \mathring{s}_3 \approx 0.2715$, $\mathring{s}_4 = \mathring{s}_5 = \mathring{s}_6 \approx 0.0549$ *and* $\mathring{s}_7 = 0.2$. *Figure 6.2a shows the relation between the variables of the equivalent uniprocessor problem. The graph "Optimal" shows the cumulative workload that has been executed at a given time. The slope of this graph is the optimal speed. The completion times of the tasks are indicated using squares. This graph must stay above the graph "Deadline", otherwise a deadline is missed. For example, at time 30, exactly* $4 + 2\sqrt[3]{3} + \sqrt[3]{2} \approx 8.1444$ *work must have been done. Similarly, the graph "Optimal" has to stay below the graph "Arrival time", which depicts the arrival times of the equivalent uniprocessor problem. Recall that a well-known property of the solution of the uniprocessor problem is that—due to convexity of the power function* p_1*—the number of speed changes is kept to a minimum. The solution* $\mathring{s}_1, \dots, \mathring{s}_n$, *as shown in Figure 6.2a, meets this property: if other speeds would be used they are either too high, too low, or would imply unnecessary changes of the speed.*

The optimal speeds for the original multicore global speed scaling problem are obtained by transforming (using (6.6)*) the optimal solution for the equivalent uniprocessor problem* $\mathring{s}_1, \dots, \mathring{s}_n$ *back to* $s_1, \dots, s_n$. *After this transformation, the optimal speeds for the original global speed scaling problem are given by* $s_1 \approx 0.2715$, $s_2 \approx 0.1882$, $s_3 \approx 0.2155$, $s_4 \approx 0.0436$, $s_5 \approx 0.0549$, $s_6 \approx 0.0436$ *and* $s_7 = 0.2000$. *During the execution of task* T_4 *the speed changes four times.*

6.4 Online speed scaling

The theory from the previous section can be applied when the work is known on beforehand. In Chapter 4 we studied the online situation where the work of tasks that are executed on a uniprocessor system is not known on beforehand. When we assume that a schedule (processor assignments and ordering of tasks) is given, and predictions of the future workload are available, we can use the transformation from Section 6.3 to solve the online global speed scaling problem. For this, multicore tasks are transformed to pieces, and the RA-SS algorithm (Section 4.4) are applied. When the arrival times and deadlines have a periodic behaviour, the PRA-SS algorithm can be used to solve the problem.

Online global speed scaling was not evaluated, and we did not study the case where a schedule was not given. The evaluation from Chapter 5 suggests that a scheduling algorithm inspired by LPT may work well in practice. We leave a further discussion of this topic for future work and come back to this in Section 8.3.

6.5 Conclusions

We have translated the multicore global speed scaling problem to an equivalent uniprocessor problem. For this, we subdivide a given schedule into so-called *pieces*. The work of a piece is multiplied by $\sqrt[\alpha]{m}$, where m is the number of active cores of the piece. After this transformation, all references to the amount of parallelism disappear from the problem. We can consider these transformed pieces as tasks in an equivalent uniprocessor problem. With this transformation we can use uniprocessor speed scaling techniques for multicore global speed scaling systems.

Chapter 7

Sleep Modes and Speed Scaling for Frame-Based Systems

Abstract – *Sleep modes and speed scaling are popular techniques for reducing the energy consumption. Algorithms for speed scaling do exist. However, determining optimal sleep modes, and the optimal combination of speed scaling and sleep modes, is still an open problem for many real-time systems.*

In this chapter well established models for sleep modes and speed scaling for frame-based systems are considered. We show that it is not sufficient—as some authors argue—to consider only individual frames. Instead, we define a schedule that also takes interactions over frames into account and prove—in a theoretical fashion—that this schedule is optimal.

7.1 Introduction

In previous chapters, we focused on speed scaling. In contrast, in this chapter the focus is first on scheduling methods which make optimal use of sleep modes. Later in this chapter, we combine speed scaling and sleep modes to make further energy reductions possible.

Sleep modes are widely used since many computers support the Advanced Configuration and Power Interface (ACPI) [2]. With sleep modes, devices and/or the microprocessor are switched to a low power sleep mode when they are not used, resulting in a decreased energy consumption. Switching to a low power sleep mode has non-negligible time and energy overheads, hence this switching only takes place when the idle time of the processor is at least the *break-even time*. To calculate this break-even time a trade-off has to be considered, and in many cases minimising

Major parts of this chapter have been presented in [MG:1].

the energy consumption for a real-time system by using sleep modes is an NP-Hard problem [31]. We show that for frame-based systems with energy costs concavely depending on the idle period length, the optimum can be found in constant time when all tasks have to execute the same amount of work, or in linear time for the general case.

For the combination of sleep modes and speed scaling, trade-offs between the two techniques have to be considered. When speed scaling is used, the speed may be decreased to reduce the energy consumption during the execution of tasks, leading to an increase of the execution time and decreased idle time. Decreasing the speed can reduce the length of an idle period to such an extent that its length gets below the break-even time. Hence, there is an interplay between speed scaling and sleep modes that should be carefully considered when minimising the energy consumption using both techniques. Neither maximising the length of the idle period or minimising the speed results in a guaranteed minimised energy consumption, instead a combination of both techniques is required (see Devadas and Aydin [32]).

One particular example of a real-time system is a frame-based system for which optimal speed scaling [76, 88] and combinations of speed scaling and sleep modes [32, 52] were studied. In contrast to prior work in this area, which considers the problem for individual invocations, we consider the problem globally—rather than for individual invocations—for systems that can use both speed scaling and sleep modes and give an analytical and easy to calculate (with polynomial time complexity) optimal solution to this problem. An example is given to show that with sleep modes, considering only individual invocations can lead to high costs, while by considering the global problem the costs can be minimised.

The contributions of this chapter are as follows.

- For the case where all tasks have the same amount of work, we present a schedule for a frame-based system that globally minimises the energy consumption for sleep modes (with multiple devices), and the combination of sleep modes and speed scaling taking into account the interplay between sleep modes and speed scaling.
- For the case where tasks have different workload, we present a dynamic programming approach where scheduling decisions are represented as a path in a DAG.
- Important general properties of schedules that optimally use sleep modes are given and proven.
- For both continuous and discrete speed scaling of a frame-based real-time system we give the optimal speeds.
- For all problems that are considered in this chapter, we give algorithms that find the schedules in either constant or linear time. The optimal speeds can be determined in polynomial time.

The remainder of the chapter is organised as follows. In Section 7.2, the model of the considered system is discussed and the notation is introduced. Section 7.3 discusses sleep modes, general properties of optimal sleep modes and the optimal solution for frame-based systems, and Section 7.4 discusses the optimal combination of sleep modes and speed scaling. An evaluation is given in Section 7.5, where the energy savings that can be attained by real devices are shown. Section 7.6 concludes this chapter with a summary and conclusions.

7.2 System model and notation

In Section 7.2.1 an application model is given. In Section 7.2.2 we present a sleep mode, and in Section 7.2.3 we give a speed scaling model. These models are used throughout this chapter.

7.2.1 application model

In this chapter we study frame-based real-time systems, which are a specific type of real-time systems with periodic tasks. To ease the notation for these periodic tasks, and in correspondence with the previous chapter, we assume that each task is executed just once (i.e. we again use the notation for aperiodic tasks). Hence, if a periodic task is executed N times, we now model this as N tasks.

The tasks are frame-based, meaning that for a *period length* T the n-th task has an *arrival time* $a_n = (n-1)T$ and a *deadline* $d_n = nT$. This leads to the active interval for task T_n, given by $[(n-1)T, nT]$, which is called a frame [32, 52, 76]. While we assume that a frame contains a single task, our results still hold when multiple tasks with the same period ($[(n-1)T, nT]$) are executed within a frame, in that case all tasks within a frame are executed consecutively and we group the tasks within a frame to a single task to ease the notation (see Section 7.3). This assumption has no a negative impact on both sleep modes and speed scaling as will be discussed later.

In contrast to [32, 52], we do not assume that the begin time of the execution of task T_n has to coincide with the arrival time a_n and denote the begin time by $b_n \in [(n-1)T, nT)$. Note that if a task T_n starts as early as possible, and if the next task T_{n+1} starts as late as possible, the idle periods of the two tasks of length I_n and I_{n+1} respectively are adjacent, leading to one big idle period of length $I_n + I_{n+1}$ as depicted in Figure 7.1.

For a task T_n we denote its work by w_n. This work has the Worst Case Work (WCW) as upper bound (denoted by $w^{\max}$), i.e. $w_n \in (0, w^{\max}]$. The execution time of task T_n is denoted by $e_n \in (0, T]$ and depends on the workload and the speed, as is discussed later. We may assume without loss of generality (see Corollary 7.1) that tasks are executed without interruptions, i.e. the completion time c_n is given by $c_n = b_n + e_n$. Hence, within the frame with task T_n, the processor can be idle for a time $I_n = T - e_n$.

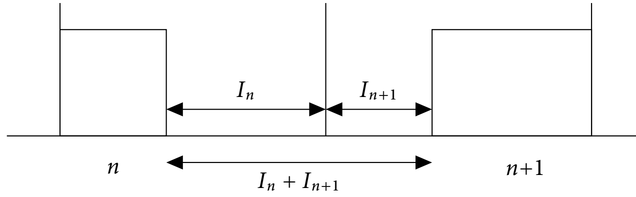


Figure 7.1 – Two idle periods (I_n and I_{n+1}) joined to one bigger idle period ($I_n + I_{n+1}$).

7.2.2 sleep modes

We assume that during the execution of a task, the processor and all peripheral devices are active. When no task is executed, the processor and all peripheral devices are unused and can be put to sleep. In total there are M devices that have at least an idle mode and a sleep mode, although more *low power sleep modes* can be available. Device m has L_m sleep modes, where for each sleep mode ℓ the power consumption is $P_{m,\ell}$. To ease the notation, we assume that the *idle mode* (the normal mode of operation during the idle period) is sleep mode 1 and for ease of notation the active mode is denoted as mode 0. For switching to sleep mode ℓ and back to the idle mode, $T_{m,\ell}$ time is spent (called latency), while the costs for both transitions are given by $E_{m,\ell}$. Hence, switching to mode ℓ should only be considered when the idle time is at least $T_{m,\ell}$. Similarly, switching to a low power mode ℓ should only be considered when the energy consumption does not increase due to switching. This is the case whenever the idle time is at least:

$$B_{m,\ell} = \frac{E_{m,\ell} - T_{m,\ell}P_{m,\ell}}{P_{m,1} - P_{m,\ell}}$$

(see [32]), which is the minimal time before which switching to the sleep mode ℓ saves more energy than it costs, called the *break-even time*. We assume (as in [12] and discussed below) that

$$T_{m,\ell} \leq B_{m,\ell}.$$

It was shown empirically that algorithms that were designed with this assumption in mind still work well when the latency is higher than the break-even time [46].

The energy consumption of device m in the best sleep mode as function of the idle time τ is given by

$$E_m^{\text{sl}}(\tau) = \min_{\ell \in \{1,\ldots,L_m\}} \left[E_{m,\ell} + P_{m,\ell}(\tau - T_{m,\ell}) \right].$$

A similar energy consumption model is used in [12]. In this model, the functions E_m^{sl} are increasing piecewise-linear concave. The sum over all individual devices determines the total energy consumed during the idle period, which is given by

$$E^{\text{sl}}(\tau) = \sum_{m=1}^{M} E_m^{\text{sl}}(\tau).$$

This expresses the total energy that is consumed during the entire idle period as a function of the length of the idle period and is therefore called the *idle-time-energy function*. Since the sum of increasing piecewise-linear concave functions is again increasing piecewise-linear concave, the function that gives the total energy for the idle time is also increasing piecewise-linear concave. To ease the notation of this function, we use the fact that a piecewise-linear function E^{sl} can be represented by linear pieces and therefore can be described by:

$$E^{\text{sl}}(\tau) = \min_{i \in \{1,\ldots,D\}} \left[R_i \tau + Q_i \right],$$

for some D and values $Q_i \geq 0$ and $R_i \geq 0$. We assume that $E^{\text{sl}}(0) = 0$, because an idle period of zero length consumes no energy.

In the derivation of the sleep mode model, we assumed that the break-even time is higher than the latency, i.e. $B_{m,\ell} \geq T_{m,\ell}$. If we look at the devices in Table 2.1, we can see that this assumption holds for many devices that occur in practice. For ease of notation, we furthermore assume without loss of generality there is no idle period before the first frame and after the last frame, otherwise dummy tasks can be added to reflect this situation.

Scheduling influences the effectiveness of sleep modes. A scheduling algorithm should create relatively many idle periods that are bigger than the break-even time. In the following example, we illustrate how the break-even times influence the scheduling decisions.

Example 7.1. *We consider the characteristics from Table 2.1 together with some fictional execution times to illustrate the trade-offs that occur when determining the optimal use of sleep modes. From this table we consider the sensor node from [78], which has four power saving modes. For this device, the energy consumption for the idle time depends on the length of the idle period, as is illustrated in Figure 7.2.*

Let the period length be 100*ms (*$T = 100$*) and the execution times of the four subsequent frames be given by* $e_1 = 15$*,* $e_2 = 78$*,* $e_3 = 90$*,* $e_4 = 100$ *(all in ms). When it is assumed (as in [32]) that all tasks are scheduled to start at the beginning at the frame, the idle periods are as depicted as in Figure 7.3a. The first idle period of 85ms is long enough for the deepest sleep mode, the second idle period of 22ms which is long enough for the second sleep mode and for the third idle period of 10ms only the first sleep mode can be used. The total idle-time energy consumption for this schedule is* 336.92*mJ.*

An alternative schedule is shown in Figure 7.3b, for which the total idle-time energy consumption is 330.01*mJ. The energy consumption is lower with respect to the previous schedule since instead of switching to sleep modes 1 and 2 for the last two idle periods, these idle periods are merged to a single big idle period and only one transition to the sleep mode 3 is used. This schedule requires two transitions to a sleep mode that both have high transition costs.*

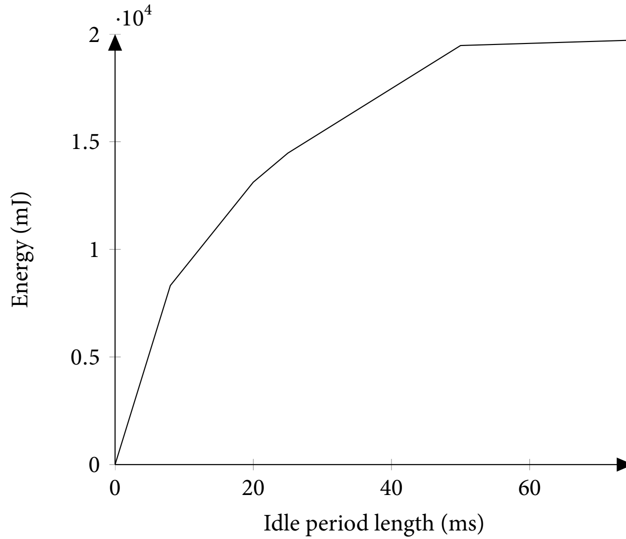


Figure 7.2 – Concave idle-time-energy function (E^{sl}) for a sensor node ([78]).

A third possible schedule that avoids these high costs is shown in Figure 7.3c. The total transition energy is reduced because the device only transitions once to a deep sleep mode instead of twice. Despite that the other idle period is not long enough for a deep sleep mode, the total energy consumption (323.48mJ) is still lower than that of the previous schedule. There are even more schedules possible, but it is easy to see that this is the optimal schedule (see Section 7.3).

The example shows that the schedule influences the lengths of the idle periods and thus also the depth of the sleep mode and how often a transition to sleep modes is made.

7.2.3 Speed scaling

Whereas with sleep modes the energy consumption during idle time is considered, with speed scaling the focus is on energy spent during the execution of a task. Hereby, time can be "traded" for energy by changing the speed. The speed of a task T_n is given by a function that maps time to a speed, this function is called the *speed function* and is denoted by $s_n : \mathbb{R}^+ \to \mathcal{S}$. Note, that in the previous chapters we used a single speed function for the entire application, whereas this chapter uses a speed function for each individual task. Many papers (e.g., [48, 90]) assume that this function is a continuous function that can attain any value in the interval $\mathcal{S} = [s^{\min}, s^{\max}]$, while other papers (e.g., [54]) use a finite set of available speeds (i.e. $\mathcal{S} = \{\bar{s}_1, \ldots, \bar{s}_K\} \subset [s^{\min}, s^{\max}]$).

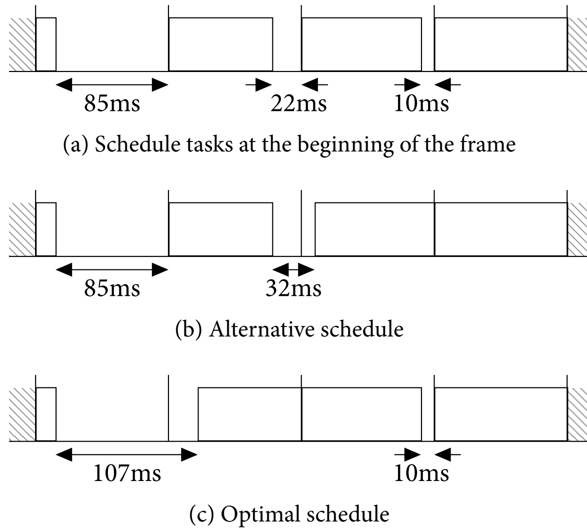


Figure 7.3 – Three possible schedules for the tasks in Example 7.1.

Together with the amount of work w_n, the speed determines the execution time of each task T_n. For a task T_n, the execution time e_n, speed function s_n and workload w_n have to satisfy

$$w_n = \int_0^{e_n} s_n(\tau)d\tau. \tag{7.1}$$

Note, that if the function s_n is given, the execution time e_n can be determined by (7.1).

7.3 Sleep modes

In this section, only scheduling in combination with the optimal use of sleep modes is considered. Since speed scaling is not yet applied, we assume that the processor runs at a constant speed, i.e. for each task T_n, the execution time e_n is known.

7.3.1 Properties of optimal sleep modes

Since the idle-time-energy function E^{sl} is concave as we explained in Section 7.2.2 (e.g., see Figure 7.2), we can derive several properties of optimal solutions. We use these properties to determine an optimal solution.

The first property shows that, instead of having two separate idle periods, it is better to merge them to one single idle period.

Lemma 7.1. *The function E^{sl} is sub-additive, i.e.,*

$$E^{sl}(x+y) \leq E^{sl}(x) + E^{sl}(y).$$

Proof. Recall that $E(0) = 0$, i.e. an idle period of length 0 consumes no energy. Then by using the definition of concavity (see Appendix A) of E^{sl} we get:

$$\begin{aligned}E^{\text{sl}}(x) &= E^{\text{sl}}\left(\frac{x}{x+y}(x+y) + \frac{y}{x+y}0\right)\\ &\geq \frac{x}{x+y}E^{\text{sl}}(x+y) + \frac{y}{x+y}E^{\text{sl}}(0)\\ &= \frac{x}{x+y}E^{\text{sl}}(x+y).\end{aligned}$$

Similarly $E^{\text{sl}}(y) \geq \frac{y}{x+y}E^{\text{sl}}(x+y)$. After adding these two inequalities, the result directly follows. □

The energy savings of merging two idle periods can grow up to 50% as the following consideration shows. Consider a device with a single sleep mode that requires zero power (shutdown). Merging two idle periods of length x and y halves the energy consumption during these idle periods when x and y are both larger than the break-even time. The reason for this is that the transition costs have to be taken into account only once, instead of twice.

Lemma 7.1 and also the next lemma are not restricted to frame-based systems, since they present properties of the sleep mode model and not of the task model (i.e. the theory holds for all task models). For example, this lemma can also be applied to systems with agreeable deadlines. In fact, this lemma can be combined with the research by Angel et al. [9] on sleep modes for agreeable deadlines, to extend their results with multiple devices.

Whereas Lemma 7.1 merges two idle periods of length x and y to decrease the energy consumption, the next lemma shows that decreasing the length of one idle period by δ and increasing the length of the other idle period by δ may decrease the energy consumption (with $\delta = x$ as a special case in Lemma 7.1).

Lemma 7.2. *For $0 \leq \delta \leq x \leq y$ we have:*

$$E^{sl}(x-\delta) + E^{sl}(y+\delta) \leq E^{sl}(x) + E^{sl}(y).$$

Proof. For $z_1 = x - \delta$, $z_2 = y + \delta$, we get $z_1 \leq x \leq y \leq z_2$. There exists a $\lambda \in [0,1]$ and $\mu \in [0,1]$ with $x = \lambda z_1 + (1-\lambda)z_2$ and $y = \mu z_1 + (1-\mu)z_2$. It can be readily checked that $\mu = 1 - \lambda$. Then by using Lemma 7.1 we get:

$$\begin{aligned}E^{\text{sl}}(x) &= E^{\text{sl}}(\lambda z_1 + (1-\lambda)z_2)\\ &\geq \lambda E^{\text{sl}}(z_1) + (1-\lambda)E^{\text{sl}}(z_2)\end{aligned}$$

and

$$\begin{aligned} E^{\mathrm{sl}}(y) &= E^{\mathrm{sl}}(\mu z_1 + (1-\mu) z_2) \\ &= E^{\mathrm{sl}}((1-\lambda) z_1 + \lambda z_2) \\ &\geq (1-\lambda) E^{\mathrm{sl}}(z_1) + \lambda E^{\mathrm{sl}}(z_2). \end{aligned}$$

Adding both inequalities gives:

$$E^{\mathrm{sl}}(x-\delta) + E^{\mathrm{sl}}(y+\delta) \leq E^{\mathrm{sl}}(x) + E^{\mathrm{sl}}(y).$$

□

This shows that for two idle periods (of lengths x and y, with $x \leq y$) that are interrupted by the execution of a part of some task, it is better to unbalance the lengths of the idle periods by starting the execution earlier or later such that the smallest idle period (of length x) decreases in length, while the longest idle period (of length y) increases in length.

A direct consequence of this lemma is given by the following corollary.

Corollary 7.1. *Given a task that is interrupted by an idle period. The energy consumption does not increase when the task is executed without interruptions.*

This corollary gives the reason why we may assume without loss of generality that there is a single task within a frame: if there are multiple tasks within a frame, it is best to execute them consecutively.

7.3.2 NON-VARIABLE WORK

In general, determining an optimal schedule for a real-time system that exploits sleep modes is NP-hard [31]. For several real-time systems we may use Lemma 7.2 to show that the optimal solution has certain properties which makes it easier to find the optimal solution. In specific situations—like for frame-based real-time applications—the following corollary can be used to determine the optimal schedule directly.

Corollary 7.2. *For frame-based systems, there is an optimal schedule in which each task T_n starts either at time $b_n = a_n$ or at time $b_n = d_n - e_n$.*

Proof. This follows directly from Lemma 7.2. □

This corollary shows that for frame-based systems with N tasks, each task has two possible begin times, reducing the number of possible schedules to 2^N. A direct result is the following corollary.

Corollary 7.3. *For any schedule given by $b_1, \ldots, b_N$ with $b_1 > a_1$, the costs do not increase when the begin time of task T_1 is changed to a_1.*

Proof. Recall that we assumed that the system was not idle before the first task, and we consider the energy consumption from the beginning of the first frame until the end of the last frame. Using this, the corollary is a direct consequence of Lemma 7.2. □

For the special case that all tasks have the same amount of work, an optimal schedule can be determined by using the following theorem.

Theorem 7.1. *For a frame-based system where each task T_n has workload $w_n = \mathcal{W}$ (i.e. $e_n = \frac{\mathcal{W}}{s^{\max}}$) for some $\mathcal{W}$, the schedule implied by*

$$b_n = \begin{cases} a_n, & \text{if } n \text{ is odd;} \\ d_n - e_n, & \text{if } n \text{ is even,} \end{cases}$$

globally minimises the energy consumption.

Proof. The proof is omitted since it is a special case of Theorem 7.2 (to be discussed in Section 7.4). □

As this result holds for any number of tasks, the first tasks are always scheduled in the same optimal way.

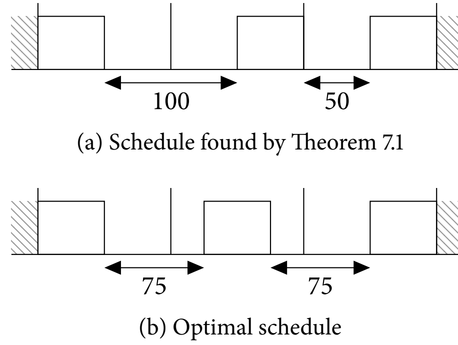
The following examples show the implications of this result.

Example 7.2 (Global and local optimisation). *Let the period be given by $T = 10$, execution times be given by $e_n = 5$ for all n, and the break-even time be given by $B_{1,1} = 6$. If each task starts when the frame begins, the idle periods are of length 5 and are shorter than the break-even time. In the optimal schedule (see Theorem 7.1), the idle periods have length 10, which is longer than the break-even time. Hence, the schedule that executes the tasks at the beginning of each frame cannot switch to a sleep mode, while the optimal schedule can use the sleep mode for each idle period.*

The example illustrates that starting each task at the beginning of the frame—as is done by some papers that were discussed in Section 3.6—may lead to a worst possible energy consumption, while global minimisation (using Theorem 7.1) can lead to a significant reduction of the energy consumption.

We have assumed that the function E^{sl} is concave, and want to emphasise that the results from this section do not necessarily hold when this function is not concave. This is illustrated by the following example.

Example 7.3 (Non-concavity of E^{sl}). *Let the application consist of 3 tasks ($N = 3$) with $T = 100$, and let the execution time be $e_n = 50$ for $n \in \{1, 2, 3\}$, hence $I_n = 50$. Then an optimal schedule is given by $b_1 = 0$, $b_2 = 150$, $b_3 = 250$, as shown in Figure 7.4a.*



(a) Schedule found by Theorem 7.1

(b) Optimal schedule

Figure 7.4 – Schedule for non-concave function E^{sl}.

Now we consider the same scenario, but now E^{sl} is not concave, but is given by the following function (see Figure 7.5):

$$E^{sl}(\tau) = \begin{cases} 4\tau, & \text{if } \tau < 75; \\ 150 + 0.2\tau, & \text{if } \tau \geq 75. \end{cases}$$

The costs for the given schedule using this non-concave function E^{sl} are 370. However, as Corollary 7.2 does not hold for the function E^{sl}, an optimal solution might start the tasks elsewhere in a frame. In fact the unique optimal solution is given by $b_1 = 0$, $b_2 = 125$ and $b_3 = 250$ (see Figure 7.4b) which has the costs 330. For this solution it holds that

$$E^{sl}(75) + E^{sl}(75) \leq E^{sl}(100) + E^{sl}(50).$$

This shows that when E^{sl} is not concave, the schedule from Theorem 7.1 is not necessarily optimal.

7.3.3 variable work

For Theorem 7.1 we assumed for each task T_n that $w_n = \mathcal{W}$. The importance of this assumption for optimality of the schedule given by Theorem 7.1 is demonstrated by the following example:

Example 7.4. *Let the period be $T = 100$ and let there be four tasks ($N = 4$) with execution times $e = (75, 25, 25, 75)$. Furthermore, assume that the processor can operate in two modes: an idle mode and a sleep mode. The idle-time-energy function is given by $E^{sl}(\tau) = \min\{\tau, 100 + 0.2\tau\}$, where the break-even time is 125. The schedule given in Theorem 7.1 leads to the following begin times: $b = (0, 175, 200, 325)$ (see Figure 7.6a). All resulting idle periods are too small for switching to the sleep mode, hence the energy consumption cannot be reduced by using sleep modes. However, the schedule given by $b = (0, 100, 275, 325)$ (see Figure 7.6b) creates an idle period of length 150 between the second and the third task, hence the energy consumption can be reduced using sleep modes (from 200 to 180).*

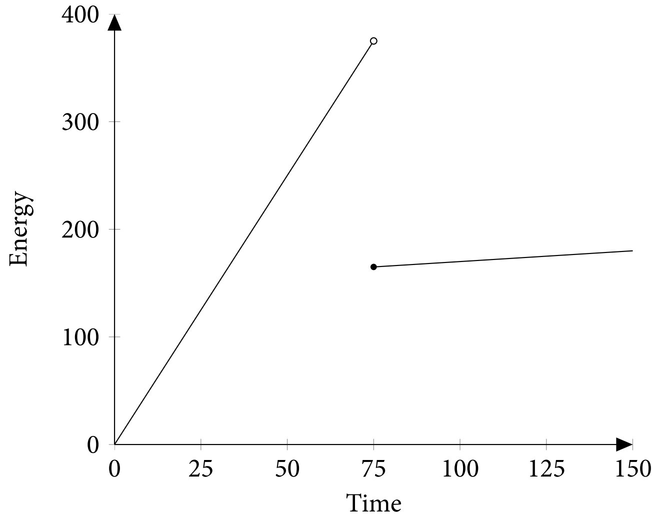
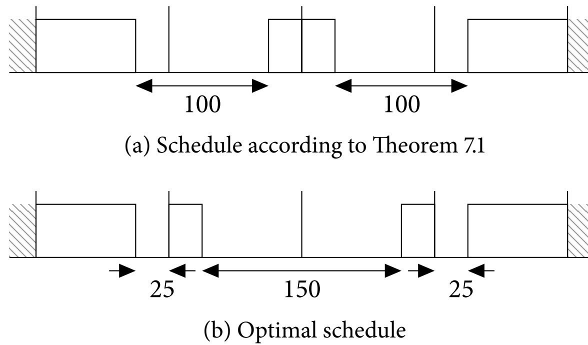


Figure 7.5 – Non-concave function E^{sl}.



(a) Schedule according to Theorem 7.1

(b) Optimal schedule

Figure 7.6 – Four tasks with variable work, $B_{1,1} = 125$.

This example shows that when not all tasks have the same workload, the schedule that is suggested by Theorem 7.1 does not necessarily produce an optimal result. Instead, other candidates for the optimal solution must be considered. As mentioned before, there are 2^N candidates for the optimal solution. In the following, we first reduce the number of candidates, before we give a procedure that finds the optimal solution for variable work in linear time.

Given a task T_n with a begin time b_n, there are at most two generally good choices for b_{n+1}. The following lemma shows that when $b_n = d_n - e_n$, the next scheduling choice follows directly:

Lemma 7.3. *Given $b_n = d_n - e_n$, then using $b_{n+1} = a_{n+1}$ never costs more energy than using $b_{n+1} = d_{n+1} - e_{n+1}$.*

Proof. When task T_{n+1} is scheduled to start at $b_{n+1} = d_{n+1} - e_{n+1}$, then there is an idle period of length $T - e_{n+1}$ between task T_n and task T_{n+1}. Moving task T_{n+1} to start at a_{n+1} reduces the length of the idle period between T_n and T_{n+1} from $T - e_{n+1}$ to 0, and creates a new idle period of the same length after task T_{n+1}, hence the costs do not change. Moreover, when this idle period can be merged with the following idle period, a bigger idle period is created and the energy consumption may decrease (see Lemma 7.1). □

Using Lemma 7.3, the energy minimisation problem can be solved using dynamic programming. Given a scheduling choice (b_n) for task T_n, there is a limited number of choices for task T_{n+1}. First, if task T_n starts at $b_n = a_n$, the next task begins at either $b_{n+1} = a_{n+1}$, using an idle period for task T_n of which the costs can be directly computed to be $E^{sl}(T - e_n)$, or the task begins at $b_{n+1} = d_{n+1} - e_{n+1}$, leading to an idle period starting in frame n and ending in frame $n+1$ for which the costs can be directly computed to be $E^{sl}(2T - e_n - e_{n+1})$. Second, if task T_n starts at $b_n = d_n - e_n$, we can assume (by Lemma 7.3) that the next task begins at $b_{n+1} = a_{n+1}$ and the costs for the idle period following task T_{n+1} depend on when task T_{n+2} starts.

We depict our dynamic programming approach with a weighted directed acyclic graph G—called the *energy graph*—in which the edges represent energy costs. A vertex v_n ($n \in \{1, \dots, N\}$) represents the execution of task T_n at time $b_n = a_n$. An additional vertex v_{N+1} is introduced as sink vertex that models the completion of the computation, hence there are $N+1$ nodes in the energy graph. Clearly, the vertices $V(G)$ of energy graph G are given by $v_1, \dots, v_N, v_{N+1}$.

The edges represent the scheduling decisions: an edge (v_n, v_{n+1}) implies that tasks T_n and T_{n+1} are scheduled at $b_n = a_n$ and $b_{n+1} = a_{n+1}$, while an edge (v_n, v_{n+2}) implies that tasks T_n, T_{n+1} and T_{n+2} are scheduled at $b_n = a_n$, $b_{n+1} = d_{n+1} - e_{n+1}$ and $b_{n+2} = a_{n+2}$. Note that, given the previous discussion, these are the only relevant scheduling choices (i.e. edges that skip more tasks do not have to be considered). The weight of an edge (v_n, v_m) (denoted by $\omega_{n,m}$) gives the energy consumption that belongs to the scheduling decisions for tasks $T_n, \dots, T_{m-1}$. Hence, the edge (v_n, v_{n+1}) has weight $\omega_{n,n+1} = E^{sl}(T - e_n)$ while the edge (v_n, v_{n+2}) has weight $\omega_{n,n+2} = E^{sl}(2T - e_n - e_{n+1})$. The edges of energy graph G for $n \in \{1, \dots, N\}$ are thus given by (v_n, v_{n+1}) and for $n \in \{1, \dots, N-1\}$ there are edges given by (v_n, v_{n+2}).

Since we may assume without loss of generality (see Corollary 7.3) that the first task starts at time $b_1 = a_1$ (represented by v_1), we only need to consider paths through the graph that start at vertex v_1. A path from vertex v_1 to vertex v_{N+1} corresponds to a schedule. The sum of the weights on this path gives the energy costs associated with this schedule. Now, the shortest path (i.e. the path with the lowest summed weights) from v_1 to v_{N+1} gives a schedule with the minimal energy consumption.

The shortest path can be easily determined using Dijkstra's shortest path algorithm [33], which has a polynomial time complexity. Since our energy graph has a special structure (e.g., no cycles, a low degree, etc), Dijkstra's algorithm can be reduced to Algorithm 7.4 and executed in linear time. In this algorithm, the costs of the shortest path up to vertex v_n is given by costs_n and the predecessor in the shortest path is given by pred_n, such that the shortest path (i.e. the minimum energy schedule) can be easily reconstructed.

Algorithm 7.4 Shortest path for energy graph.

```
for i = {1, ..., N} do
  if i = 1 or costs_{i-1} + ω_{i-1,i} < costs_{i-2} + ω_{i-2,i} then
    costs_i = costs_{i-1} + ω_{i-1,i}
    pred_i = i - 1
  else
    costs_i = costs_{i-2} + ω_{i-2,i}
    pred_i = i - 2
  end if
end for
```

This procedure is illustrated using the following example:

Example 7.5. *We again consider the situation from Example 7.4, given by* $T = 100$, $e_1 = 75$, $e_2 = 25$, $e_3 = 25$ *and* $e_4 = 75$. *The idle-time-energy function is chosen to be* $E^{sl}(\tau) = \min\{\tau, 100 + 0.2\tau\}$ *and has as break-even time of* 125. *The energy graph G has* 5 *vertices:* $v_1, \dots, v_5$. *The edges representing the energy consumption are given as follows. The case that task* T_1 *starts at* $b_1 = a_1$, *and task* T_2 *at* $b_2 = a_2$ *leads to an idle period of length 25 and the energy consumption of the respective idle period is* $E^{sl}(T-e_1) = 25$. *This is modelled using the edge* (v_1, v_2) *with weight* 25. *When task* T_2 *starts at* $b_2 = 2T - e_2$ *the idle time between tasks* T_1 *and* T_2 *together is* $2T - e_1 - e_2$ *and the energy consumption for the idle time of both tasks is* $E^{sl}(2T - e_1 - e_2) = 100$, *this is modelled using the edge* (v_1, v_3) *with weight* 100. *The complete energy graph is depicted in Figure 7.7. The shortest path from* v_1 *to* v_5 *in this graph can be determined using Algorithm 7.4 and is given by* (v_1, v_2, v_4, v_5). *Each task* T_n *of which the node is in the path is scheduled as* $b_n = a_n$, *hence* $b_1 = 0, b_2 = 100, b_4 = 300$. *When the node for task* T_n *is not in the path,* $b_n = d_n - e_n$, *hence it follows that* $b_3 = 75$. *The energy consumption is the weighted length of the path, hence* 180 *energy is consumed.*

Minimising the shortest path is not the same as maximising the length of the largest idle period, as is illustrated by the following example.

Example 7.6. *Let* $T = 100$, $e_1 = 35$, $e_2 = 25$, $e_3 = 25$ *and* $e_4 = 35$, *and let the idle-time-energy function given be* $E^{sl}(\tau) = \min\{\tau, 100 + 0.2\tau\}$, *i.e. the break-even time is* 125. *Figure 7.8 shows the corresponding energy graph. The biggest idle period that is possible in a schedule (of length* 130*) can be created by using the edge* (v_2, v_4), *the only path from* v_1 *to* v_5 *containing this edge is* (v_1, v_2, v_4, v_5) *which has costs* 260. *The*

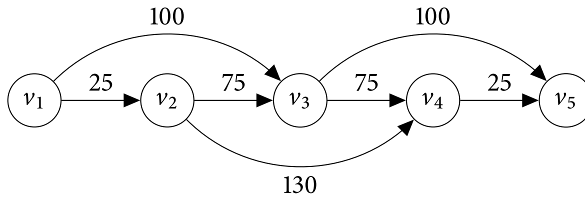


Figure 7.7 – Energy graph for Example 7.5.

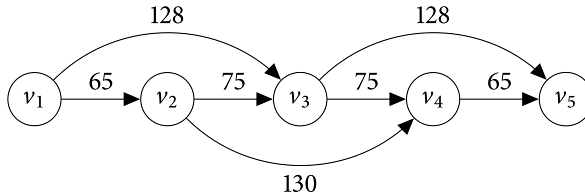


Figure 7.8 – Energy graph for Example 7.6.

shortest path from v_1 to v_5 (of length 256) is given by (v_1, v_3, v_5). Hence, maximising the length of the longest idle period does not necessarily lead to a minimised energy consumption.

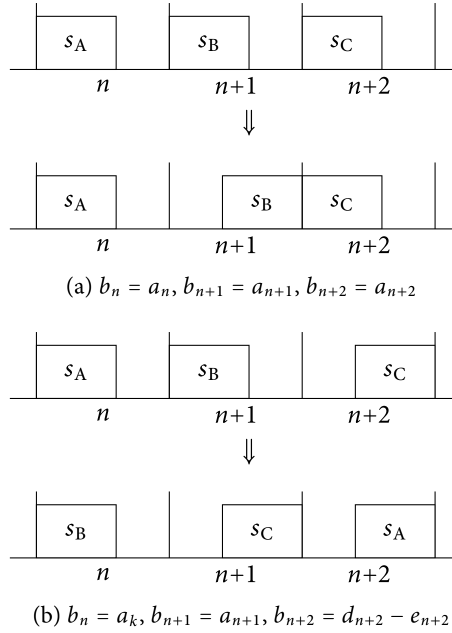
7.4 Speed scaling

Since the results on the optimal schedules hold—as will be shown—independently of the speed function, we initially assume that the speed for each task T_n is given by a function $s_n : \mathbb{R}^+ \to \mathcal{S}$ without requiring this function to be known. The combination of sleep modes and speed scaling might seem counter-intuitive at first: neither maximising the idle period length nor minimising the speed will necessarily result in a minimisation of the energy. Instead, a simultaneous optimisation that considers both sleep modes and speed scaling is required.

7.4.1 non-variable work

When all tasks have the same amount of work, it becomes easier to determine an optimal schedule. This is used by the following lemma which helps to find an optimal schedule of the tasks.

Lemma 7.4. *Assume that the work for all tasks T_i is given by $w_i = \mathcal{W}$. Furthermore, assume that a schedule is given with $b_n = a_n$ and $b_{n+1} = a_{n+1}$ for some n. When using the begin time $b_{n+1} = d_{n+1} - e_{n+1}$ instead of using $b_{n+1} = a_{n+1}$, speed functions*



(a) $b_n = a_n$, $b_{n+1} = a_{n+1}$, $b_{n+2} = a_{n+2}$

(b) $b_n = a_k$, $b_{n+1} = a_{n+1}$, $b_{n+2} = d_{n+2} - e_{n+2}$

Figure 7.9 – Alternative scheduling for Lemma 7.4.

exist by which the energy costs do not increase. When we change the schedule such that task T_{n+1} finishes at d_{n+1}, there exists a speed function which ensures that the deadline is still met and the energy costs do not increase by these changes.

Proof. Assume the speed functions for tasks T_n, T_{n+1} and T_{n+2} are given by s_A, s_B and s_C respectively. After changing b_{n+1} according to the lemma, we have to find a new speed function for tasks T_n, T_{n+1} and T_{n+2}. Two situations have to be considered (by Corollary 7.2); in both situations the energy consumption does not increase:

(i) $n + 2 > N$ or $b_{n+2} = a_{n+2}$: In this situation, only the begin time of task T_{n+1} changes, while we keep the speed function for each task the same. Then the idle period that follows task T_{n+1}—which is not followed by an idle period of a next task—now occurs before task T_{n+1} as depicted (for $b_{n+2} = a_{n+2}$) in Figure 7.9a. Since no idle period becomes smaller and all speed functions remain the same, the energy consumption does not increase. Actually, an opportunity for further energy reduction might be created since the idle periods of tasks T_n and T_{n+1} are adjacent.

(ii) $n+2 \leq N$ and $b_{n+2} = d_{n+2} - e_{n+2}$: In this situation, the begin time of task T_{n+1} changes, and also the speed functions of some tasks are interchanged as described below. In the old situation, there was an idle period of length I_n after

the execution of task T_n and there is an idle period of length $I_{n+1} + I_{n+2}$ between tasks T_{n+1} and T_{n+2}. By using $b_{n+1} = d_{n+1} - e_{n+1}$ and interchanging the speed functions such that in the new situations for tasks T_n, T_{n+1} and T_{n+2} respectively the speed functions s_B,s_C and s_A are used, the situation as depicted in Figure 7.9b is created. Here the length of the idle period between task T_n and task T_{n+1} has now the length $I_{n+1} + I_{n+2}$ and the idle period between task T_{n+1} and task T_{n+2} has length I_n, meaning that the energy consumption remains the same.

□

The problem of finding a schedule and speeds that together minimise the energy consumption is a difficult problem, since this generally requires simultaneous optimisation for both sleep modes and speed scaling. For frame-based systems with non-variable work we will show that first optimising for sleep modes and then for speed scaling will lead to a global minimum. This is stated in the following theorem.

Theorem 7.2. *Assume a frame-based system where each task has the same workload, i.e. $w_i = \mathcal{W}$ for all tasks T_i. Let the begin times be given by $b_1, \dots, b_N$ and speed functions be given by $s_1, \dots, s_N$. Then there are alternative speed functions $\tilde{s}_1, \dots, \tilde{s}_N$ such that together with the schedule implied by*

$$\tilde{b}_n = \begin{cases} a_n, & \text{if } n \text{ is odd;} \\ d_n - e_n, & \text{if } n \text{ is even,} \end{cases}$$

the energy consumption is not higher than the energy consumption of the schedule given by $b_1, \dots, b_N$ and $s_1, \dots, s_N$.

Proof. For task T_1, using $\tilde{b}_1 = a_1$ does not increase the energy consumption by Corollary 7.3. Now assume the theorem does not hold and consider a schedule for which $\hat{b}_1 = \tilde{b}_1, \hat{b}_2 = \tilde{b}_2, \dots, \hat{b}_k = \tilde{b}_k$ and $\hat{b}_{k+1} \neq \tilde{b}_{k+1}$ with k as large as possible. Note, that $k \leq N - 1$ due to the assumption that the theorem does not hold. There are two possibilities:

» k is odd: Hence $\tilde{b}_k = \hat{b}_k = a_k$ and by Lemma 7.4, there is a schedule such that $\hat{b}_{k+1} = d_{k+1} - e_{k+1}$ and the begin times for tasks $T_1, \dots, T_k$ remain the same, which does not lead to an increase of the energy costs. This contradicts that k was chosen as high as possible.

» k is even: Hence $\tilde{b}_k = \hat{b}_k = d_k - e_k$ and by Lemma 7.3, there is a schedule such that $\hat{b}_{k+1} = a_{k+1}$ and the begin times for tasks $T_1, \dots, T_k$ remain the same, which does not lead to an increase of the energy costs. This contradicts that k was chosen as high as possible.

Since in both cases, the assumption that the theorem does not hold leads to a contradiction, the theorem is proven. □

Theorem 7.2 shows that for the optimal combination of sleep modes and speed scaling, we can take the optimal sleep mode schedule from Theorem 7.2 and then determine the optimal speed functions in a second step. Since the speeds are either chosen from the set $[s^{\min}, s^{\max}]$ or from the finite set $\{\bar{s}_1, \dots, \bar{s}_K\} \subset [s^{\min}, s^{\max}]$, both cases are treated separately in the following two sections.

7.4.2 OPTIMAL CONTINUOUS SPEED SCALING

In the state of the art work by Devadas and Aydin [32], the power function given by $p(s) = \gamma_1 s^3 + \gamma_2$ is used. In the following, we use the same power and task model, and use results from [32] to find an analytic solution to the problem considered in this section. As mentioned in Section 2.6.1, we may assume that $s_n(t)$ is a constant function, hence we now may assume that s_n is a value chosen from the interval $[s^{\min}, s^{\max}]$.

The speed influences the execution time e_n and the active energy that devices consume for time e_n, therefore we can no longer treat the active energy of all devices as a constant. The energy consumption during the execution of task T_n by device m is determined as $P_{m,0}(T - \frac{w_n}{s_n})$. For ease of notation, we assume in the remainder of this chapter that the active time energy consumption is part of E^{sl}.

Given the schedule from Theorem 7.2, there are two possible situations for each task T_n:

» $b_n = a_n$ and $b_{n+1} = d_{n+1} - e_{n+1}$.

In this situation, the idle period of tasks T_n and T_{n+1} together have length $2T - e_n - e_{n+1}$. For finding the optimal speed, we solve the following optimisation problem:

Optimisation Problem 7.1.

$$\begin{aligned}
\min_{s_n, s_{n+1} \in [s^{\min}, s^{\max}]} \quad & p(s_n)\frac{w_n}{s_n} + p(s_{n+1})\frac{w_{n+1}}{s_{n+1}} \\
& + E^{sl}\left(2T - \frac{w_n}{s_n} - \frac{w_{n+1}}{s_{n+1}}\right), \\
s.t. \quad & \frac{w_n}{s_n} \leq T, \\
& \frac{w_{n+1}}{s_{n+1}} \leq T.
\end{aligned}$$

The cost function consists of the energy when active and the energy consumed during the idle period (given by $E^{\text{sl}}(I)$). We determine the speeds that minimise these costs, while making sure that both task T_n and task T_{n+1} can individually be executed within a frame of size T, as expressed by the two constraints. Either the minimum is attained inside the interior of the feasible region (the speeds that are allowed by the constraints), or the minimum is attained on the boundaries of the feasible region. In the first case

the minimum can be determined by solving the unconstrained problem (i.e. only minimising the cost function). This problem has a minimum for which holds that $s_n = s_{n+1}$ (see Section 2.6.1), by which the problem reduces to a one dimensional problem ($s_n = s_{n+1}$, only one degree of freedom). This specific problem is solved by Devadas and Aydin [32]. In the second case, where the solution is on the boundary, we consider the cases for $s_n = \max\{s^{\min}, \frac{w_n}{T}\}$, $s_n = s^{\max}$, $s_{n+1} = \max\{s^{\min}, \frac{w_{n+1}}{T}\}$ and $s_{n+1} = s^{\max}$ which are minimal speeds, maximal speeds and the lowest speeds by which the deadline is met. When looking for a solution on the boundary, one dimension of the problem is removed, reducing it to the problem that was solved in [32]. Since there are five candidate solutions that can be analytically found in $O(D)$ time (D is the number of linear pieces) using the results from [32], we can determine the costs for each of the five candidate solutions and use the one that minimises the costs.

» Other situations.

In the other situations, the n-th frame has an idle time of length I_n and is not adjacent to an idle period of another task (i.e. task T_{n-1} or task T_{n+1}). In that case the following optimisation problem has to be solved:

Optimisation Problem 7.2.

$$\min_{s_n \in [s^{\min}, s^{\max}]} \quad p(s_n)\frac{w_n}{s_n} + E^{sl}\left(T - \frac{w_n}{s_n}\right),$$
$$s.t. \quad \frac{w_n}{s_n} \leq T.$$

Again we can use [32] to find a solution to this problem.

As the solution in [32] can be found in $O(D)$ time (i.e. it depends linearly on the number of low power states for all devices), and the schedule is determined in $O(1)$ time, the time complexity for finding a schedule that optimally combines sleep modes and speed scaling is $O(D)$. As finding the solution to the above optimisation problems is already discussed in [32], we do not repeated the method for finding a solution here. Instead we give a brief example to illustrate this approach.

Example 7.7. *Let a frame-based real-time system be given with period* $T = 10$, *work* $w_n = 5$ *for all n and break-even time* $B_{1,1} = 0.4$. *Furthermore, consider an active power and idle power of* 1, *and a power consumption in the sleep mode of* 0.01. *We restrict the speeds to* $\mathcal{S} = [0.1, 1]$ *and choose* $p(s) = s^3$.

We can solve Optimisation Problem 7.2 to find the optimal speeds, because we have an idle period of length $I = 10 - \frac{5}{s_n}$ *that is not merged with another idle period. The speed should be at least* $\frac{\mathcal{W}}{T} = \frac{1}{2}$, *otherwise the deadline will be missed. The energy consumption for speed scaling as a function of the speed is given by*

$$E^{ss}(s_n) = w_n s_n^2.$$

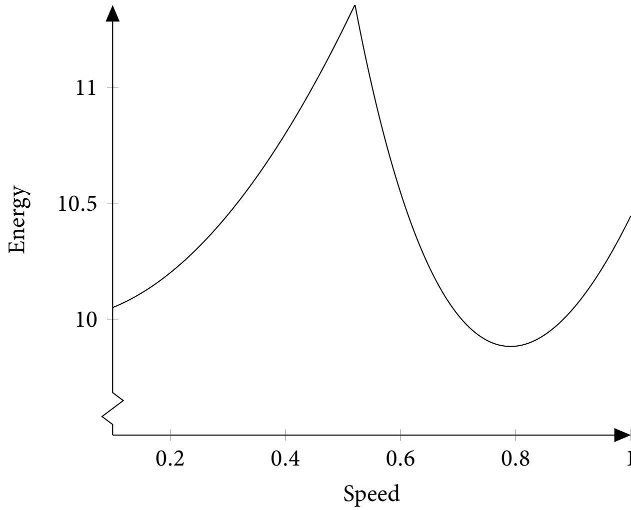


Figure 7.10 – Total energy consumption (sleep modes + discrete speed scaling) as function of the effective speed.

To ease the optimisation, the energy consumption for sleep modes will also be expressed in terms of the speed. Since E^{sl} is a function of the idle time, the idle time has to be determined in terms of the speed, which is:

$$I(s_n) = T - \frac{w_n}{s_n}.$$

The total energy as a function of the speed becomes

$$E^{sl}(I(s_n)) + E^{ss}(s_n),$$

and is shown in Figure 7.10. If the speed is at least $\frac{w_n}{T-B_{1,1}} \approx 0.52$ (which is the peak in the figure), the idle time is long enough to switch to the sleep mode. For speeds below this speed, no sleep mode but only speed scaling is used. For speeds above this speed, the device can be put to sleep mode during the idle period. Devadas and Aydin [32] show that these two different cases have to be considered to find a minimum and give an analytic solution to this problem. Since the minimal speed should be at least $\frac{1}{2}$, the speed that is feasible and minimises the energy consumption is close to 0.8 as can be seen from inspection of Figure 7.10. This shows that there is an interplay between sleep modes and speed scaling: instead of choosing the lowest speed that is allowed (speed $\frac{1}{2}$) or instead of using a speed that maximises the idle period (speed 1), a speed is used that maximises energy savings using both sleep modes and speed scaling. For a thorough discussion and exact calculation of the optimal speeds, we refer the interested reader to the work of Devadas and Aydin [32].

7.4.3 Optimal discrete speed scaling

In case only a finite number of speeds are allowed, i.e. the speeds from the set $\mathcal{S} = \{\bar{s}_1, \ldots, \bar{s}_K\}$, a different procedure is deployed to determine the optimal speeds. To specify a solution for task T_n, let $w_{n,1} \geq 0$ work be executed at speed $\bar{s}_1$, $w_{n,2} \geq 0$ work be executed at speed $\bar{s}_2$, etc (see Section 2.6.4). In total we must have $\sum_{i=1}^{K} w_{n,i} = w_n$, meaning that the workload of task T_n is distributed over the available speeds. As for the situation with continuous speeds, two cases are considered:

» $b_n = a_n$ and $b_{n+1} = d_{n+1} - e_{n+1}$.
In this case, the idle period for frames n and $n+1$ together have length $I_n + I_{n+1}$, where $I_\ell = T - \left[\sum_{i=1}^{K} \frac{r_{\ell,i}}{\bar{s}_i}\right]$, $\ell \in \{n, n+1\}$. The function E^{sl} is piecewise linear, where piece j is characterised by two values Q_j and R_j, meaning that the energy consumption for the idle period is determined as $R_j I_j + Q_j$.
Hence, for some linear piece j, the optimal speeds are determined by solving the following optimisation problem.

Optimisation Problem 7.3.

$$\begin{aligned}
\min_{\substack{w_{n,1},\ldots,w_{n,K} \\ w_{n+1,1},\ldots,w_{n+1,K}}} \quad & \left[\sum_{i=1}^{K} p(\bar{s}_i)\frac{w_{n,i} + w_{n+1,i}}{\bar{s}_i}\right] \\
& + R_j\left[2T - \sum_{i=1}^{K} \frac{w_{n,i} + w_{n+1,i}}{\bar{s}_i}\right] + Q_j + R_0 \sum_{i=1}^{K} \frac{w_{n,i} + w_{n+1,i}}{\bar{s}_i}, \\
s.t. \quad & \sum_{i=1}^{K} \frac{w_{n,i}}{\bar{s}_i} \leq T, \\
& \sum_{i=1}^{K} \frac{w_{n+1,i}}{\bar{s}_i} \leq T, \\
& \sum_{i=1}^{K} w_{n,i} = w_n, \\
& \sum_{i=1}^{K} w_{n+1,i} = w_{n+1}, \\
& w_{n,i} \geq 0\,, \ \textit{for all } i \in \{1, \ldots, K\}, \\
& w_{n+1,i} \geq 0\,, \ \textit{for all } i \in \{1, \ldots, K\}.
\end{aligned}$$

Here the first term of the cost function is the energy consumption of the processor during the active period, the term with the constant R_0 is the active power of the devices during the active period and the energy during the idle period is given by the linear terms with coefficients Q_j and R_j (see Section 7.2.2). This problem is a linear program, which can be solved in polynomial time. The above optimisation problem is solved D times, namely once for each $j \in \{1, \ldots, D\}$ and we choose the j that gives the best solution.

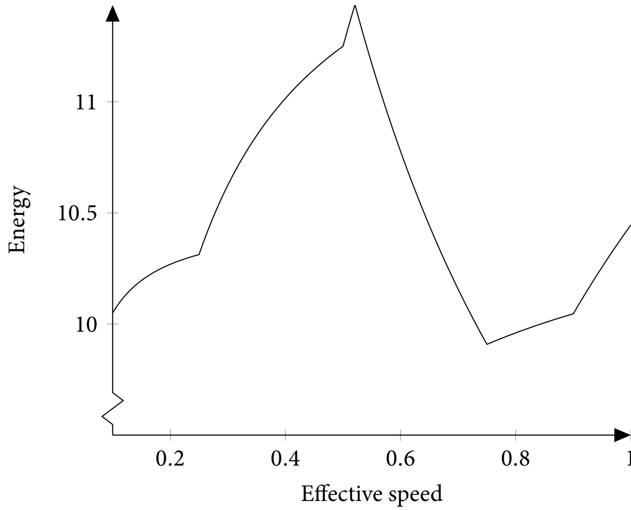


Figure 7.11 – Total energy consumption (sleep modes + discrete speed scaling) as function of the effective speed.

» Other situations.

For the other cases, no idle periods can be merged and the optimisation problem is given by the following optimisation problem.

Optimisation Problem 7.4.

$$\min_{w_{n,1},\dots,w_{n,K}} \left[\sum_{i=1}^{K} p(\bar{s}_i)\frac{w_{n,i}}{\bar{s}_i}\right] + R_j\left[T - \sum_{i=1}^{K}\frac{w_{n,i}}{\bar{s}_i}\right] + Q_j + R_0\sum_{i=1}^{K}\frac{w_{n,i}}{\bar{s}_i},$$

$$\begin{aligned} s.t.\quad & \sum_{i=1}^{K}\frac{w_{n,i}}{\bar{s}_i} \le T,\\ & \sum_{i=1}^{K} w_{n,i} = w_n,\\ & w_{n,i} \ge 0\,,\ \textit{for all } i \in \{1,\dots,K\}. \end{aligned}$$

This is again a linear problem, which can be solved in polynomial time. The costs for each candidate solution should be calculated and the feasible candidate with the lowest costs minimises the energy.

In Example 7.7, we calculated the optimal sleep modes and speed scaling settings for continuous speed scaling. The next example repeats this for a finite set of speeds.

Example 7.8. *We again consider the frame-based real-time system with period $T = 10$, work $w_n = 5$ and break-even time $B_{1,1} = 0.4$. A single device is used with active power*

and idle power of 1 *and when put to sleep the power consumption is* 0.10. *The speeds* $\mathcal{S} = \{0.1, 0.25, 0.5, 0.75, 0.9, 1\}$ *are available and we again use the power function* $p(s) = s^3$ *for the processor.*

As in Example 7.7, the energy consumption for optimal sleep modes can be determined if the speed is fixed. Since not all speeds in the interval $[0.1, 1]$ *are available, the unavailable speeds are simulated (see Section 2.6.5). The average speed that is obtained this way is referred to as the* effective speed. *Figure 7.11 shows the energy consumption as function of the effective speed. The optimal speed is now* 0.75.

7.4.4 variable work

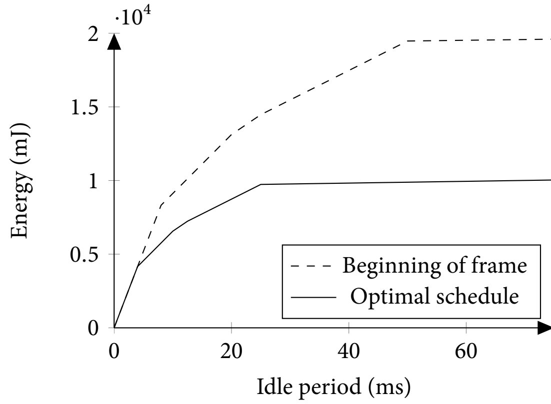
Note, that Lemma 7.3 still holds when speed scaling is used. Therefore, the structure of the energy graph for frame-based systems with speed scaling is the same as the structure for frame-based systems without speed scaling. The only thing which changes are the costs corresponding to the edges in the graph. These values can be determined by using the procedures that were discussed in Section 7.4.2 and Section 7.4.3.

For w_n work and when only considering sleep modes, $E^{\text{sl}}(T - \frac{w_n}{s^{\max}})$ is used as the weight for an edge that encodes the costs of task T_n. When continuous speed scaling is used, $p(s_n)\frac{w_n}{s_n} + E^{\text{sl}}\left(T - \frac{w_n}{s_n}\right)$ is used as costs for this edge, however s_n is not yet known. When constructing the graph, s_n can be determined (locally) and the energy for task T_n can be used as weight in the graph. After constructing this graph, a shortest path algorithm can be applied to find the schedule and speeds that minimise the energy consumption. This last step works exactly as in Section 7.3.3.

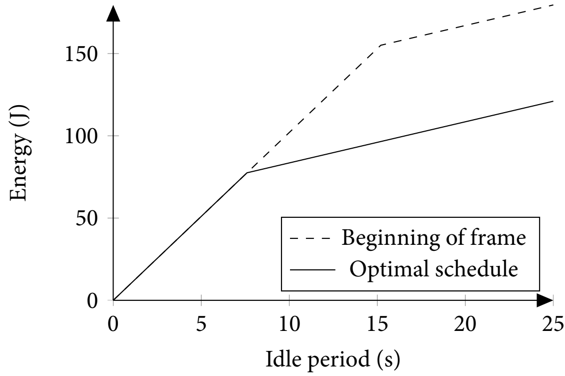
7.5 Evaluation

To get more insights in optimal scheduling for sleep modes, we use the measurements of the devices given in Table 2.1 and compare our method to the approach of Devadas and Aydin [32]. In our evaluation, we consider two settings to show what can be gained when the results from Section 7.3 are applied. In the first setting, we assume all tasks start as soon as they arrive, as is assumed by, e.g., Devadas and Aydin [32] and Kong et al. [52]. This setting is illustrated by the graphs "Beginning of frame" in Figure 7.12. For the other setting we assume the tasks are scheduled according to the results from Section 7.3, denoted by "Optimal schedule" in Figure 7.12. Here, we assume many tasks are scheduled and thus every two idle periods can be merged into a bigger idle period, which enables significant energy savings. We use the average energy per frame, to make it possible to compare the energy savings to the first setting.

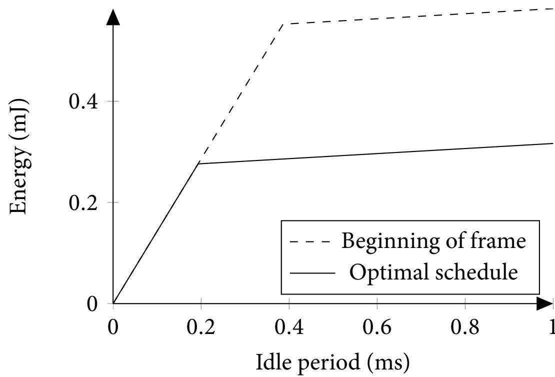
Figure 7.12a shows the energy consumption of the sensor node from [78], where each piece of the graph corresponds to a low power sleep mode. The break-even time for the first low power mode is 5ms, which is shown for the situation where the tasks start at the beginning of the frame. When the tasks are scheduled optimally,



(a) Sensor node



(b) Harddisk (IBM Ultrastar 36Z15)



(c) Ethernet card (WaveLAN)

Figure 7.12 – Energy consumption.

the idle period of two frames can be combined and the idle period of 2.5ms in two consecutive frames are combined to a bigger idle period of 5ms shared by both frames (again, see Figure 7.1), this makes it possible to switch earlier to a low power state. Furthermore, by joining two idle periods, the costs for switching to a low power state and back have to be taken into account only once. This explains why the costs for an idle period of length 60ms per frame is halved when the results from Section 7.3 are used: the state transition only takes place in 50% of the frames, hence only 50% of the transition energy is required.

7.6 Conclusions

In this chapter, we discussed several general properties of optimal sleep modes. One important property—given the assumptions in this chapter—is that it is best to either start each task as soon as possible or as late as possible. For frame-based systems, these properties are used to find a globally optimal schedule that minimises the energy consumption using sleep modes. This is done for nonvariable work and for variable work.

In addition to optimal scheduling for sleep modes, we have shown how to combine sleep modes and speed scaling optimally. We have proven that for frame-based real-time systems, first optimising for sleep modes and then for speed scaling, leads to a global minimum of the energy consumption. Simultaneous optimisation for speed (speed scaling) and idle period length (sleep modes) is required, since neither maximising the idle period length nor minimising the speed will necessarily minimise the energy consumption.

Compared to the state-of-the-art research, the algorithms from this chapter create more opportunities to use sleep modes and will therefore use less energy. We have shown that in case the energy consumption in the low power state is zero (total shutdown of the device), the idle-time energy consumption is reduced by up to 50% with respect to the state of the art.

Chapter 8

Conclusions and Recommendations

This chapter summarises the results presented in this thesis (Section 8.1), gives answers to the research questions from Chapter 1 (Section 8.2) and proposes directions for future research (Section 8.3).

8.1 Summary

The topic of this thesis is energy minimisation under deadline constraints by using algorithmic power management techniques. Algorithmic power management makes it possible for software developers to significantly reduce the energy consumption of computing devices. Software can be used to decrease the speed of devices in order to lower their energy consumption (speed scaling), or it can put them in a low power sleep mode (sleep modes).

In Chapter 4 we studied uniprocessor speed scaling, both for offline and for online optimisation. For energy efficient operation in both the offline and online situation, algorithms must avoid unnecessary speed changes. When two consecutive tasks use different speeds, and when the average speed (work for both tasks divided by execution time of both tasks) is feasible for both tasks, the energy consumption can be decreased by using this average speed. This fact is used by many speed scaling algorithms to minimise the energy consumption. For the case that static power is accounted for until the last deadline, the offline problem was solved using the *RecursiveSmoothing* algorithm by Huang and Wang [43]. For the other case where static power is accounted for until the last completion, RecursiveSmoothing is no longer optimal and we proposed a different approach. It may happen that by using a higher speed for the final tasks, the length of the schedule can be decreased, and with it the static energy consumption. More precisely, if increasing speeds to the critical speed (s^{crit}) decreases the schedule length, the total energy consumption may be reduced (see Section 4.3.2).

For online optimisation, current approaches rarely use the fact that unnecessary speed changes should be avoided. Instead, greedy approaches are commonly used in the literature. These predict the workload for the task at hand and (locally) choose the lowest speed that is allowed. We presented two algorithms that, in contrast to the related work, avoid unnecessary speed changes, namely the RA-SS algorithm that uses predictions of future work, and the PRA-SS algorithm that only requires knowledge of the average workload. We have shown that these algorithms reduce the energy consumption (by up to 55%) with respect to greedy approaches, and are very robust against inaccurate predictions and many modelling inaccuracies.

In Chapter 5, global speed scaling for multicore systems (i.e. all cores run at the same speed) was discussed. For such systems where the execution of tasks may be restricted by precedence constraints and all tasks have a common arrival time and a common deadline, the speed selection and scheduling problem was treated. When such applications are executed on a global speed scaling system, it is best to increase the speed when only a few cores are active, such that the speed can be decreased when many cores are active. By this, the energy consumption increases for only a few cores, while it decreases for many cores. However, this increase/decrease should be limited, as otherwise the increased energy consumption of the few cores is no longer (over)compensated by the decreased energy consumption for many cores. This suggests a balance of the speeds for the times when m cores are active and when n cores are active. More precisely, the optimal speed that is used when m cores are active should be multiplied by a factor $\sqrt[\alpha]{\frac{m}{n}}$ to obtain the optimal speed when n cores are active.

To minimise the energy consumption on a global speed scaling system, speed scaling and scheduling must be considered simultaneously. Since this is an NP-hard problem, scheduling heuristics must be used. To characterise the energy minimising schedule, we derived a scheduling criterion (referred to as the *weighted makespan*) that implicitly takes the optimal speeds into account (Section 5.4). This weighted makespan is given by

$$\bar{S} = \sum_{m=1}^{M} \omega_m \sqrt[\alpha]{m},$$

where ω_m is the total duration in work (e.g., in clock cycles) for which exactly m cores are active, and α is a system dependent constant. Minimising our scheduling criterion and then choosing the optimal speeds globally minimises the energy consumption. For the specific situation of two cores, we have shown that minimising the makespan also minimises the weighted makespan (Section 5.4.3), while this generally does not hold for more than two cores (Section 5.4.3).

As there are already many existing scheduling algorithms, we did not introduce new algorithms but we determined how well some of these algorithms are at minimising the energy consumption. Many of the algorithms were designed to minimise the makespan. Therefore, we determined the relation between minimising the makespan and minimising our weighted makespan. Theorem 5.2 gives an approxi-

mation ratio for the weighted makespan in terms of an approximation ratio for the makespan. This can be used to determine how good existing scheduling algorithms are at minimising the energy consumption. Independently of the scheduling algorithm, we characterised the best case schedule (Section 5.5.1) to show how much energy can be gained in theory with respect to the state of the art (up to 44% for 16 cores). For the specific situation where the LPT scheduling algorithm is used, simulations show that energy savings of more than 30% can be achieved (Section 5.5.2).

In Chapter 6 we extended the above global speed scaling problem to instances with individual arrival times and deadlines for all tasks. Here, we subdivided the time line into so-called *pieces* (Section 6.2), where the subdivision depends on the arrival times and deadlines of tasks. Due to the definition of a piece, the number of active cores does not change during a piece. Based on this property we introduced a transformation (Section 6.3) that reduces this global speed scaling multicore problem to a well-known uniprocessor problem with agreeable deadlines. This uniprocessor problem (and with it, the multicore problem) can be solved using the algorithms presented in Chapter 4. For the online situation, we suggested in Section 6.4 how to choose efficient speeds.

We studied sleep modes in Chapter 7 for so-called *frame-based real-time systems*. In such real-time systems, each task (or group of tasks) is executed within a frame. Since idle times of adjacent frames can be merged to longer idle intervals, the length of idle intervals and with it the effectiveness of sleep modes can be influenced by appropriate scheduling decisions. We have shown that it is not sufficient to optimise for the longest idle time; instead, a schedule that globally minimises the energy consumption is desired. We derived algorithms that find this energy minimising schedule.

If sleep modes and speed scaling are combined, we considered a trade-off. Speed scaling can be used to reduce the energy consumption, but then it also reduces the length of an idle interval. Since the length of the idle interval determines the effectiveness of sleep modes, this leads to an interplay between speed scaling and sleep modes. For frame-based real-time systems where all tasks have equal work, we have shown that we can first determine the optimal schedule and then determine the optimal speeds (Theorem 7.2). When not all tasks have the same workload, dynamic programming can be used to solve the problem of scheduling for sleep modes combined with speed scaling in linear time (Chapter 7).

8.2 Conclusions

In the introduction (Chapter 1) several research questions were presented. The research in the subsequent chapters provided answers for these questions. In the following, the answers to these questions are summarised.

For uniprocessor systems:

» *What are the optimal speeds when static power is present?*

To minimise the energy consumption of a real-time system with agreeable deadlines, the speeds should be set as low as possible, and unnecessary speed changes must be avoided, which is done by the RecursiveSmoothing algorithm (described in Section 4.3.1). An additional step is required for the case where the static power is only accounted for until the completion time of the last task. When the schedule length can be decreased by increasing the speed of a task to the critical speed (s^{crit}), the total energy consumption may be decreased. In Chapter 4, we studied an algorithm that uses these ideas to minimise the energy consumption.

» *How to choose the optimal speeds online when only (possibly inaccurate) predictions of the amount of work are given?*

As mentioned above, it is better to use the average speed (if the real-time constraints allow for it), because this lowers the energy consumption (see also Section 2.6.1). In Section 4.4 the RA-SS and PRA-SS algorithms are introduced that use this result. These algorithms are based on predictions of the work, or of the average work, to determine the speed of future tasks. With only a prediction of the average work and applying PRA-SS, we show that an energy reduction of up to 55% with respect to a greedy approach (that does know the future) is possible.

» *How can speed scaling be combined with sleep modes?*

When speed scaling is used to reduce the energy consumption, it simultaneously reduces the length of an idle interval. Since the effectiveness of sleep modes is affected by the length of the idle interval, both problems must be considered simultaneously. For a frame-based real-time system where all tasks have an equal workload, we have shown that it is optimal to first schedule for optimal sleep modes, and then determine the optimal trade-off between sleep modes and speed scaling locally (for frames). For frame-based real-time systems where the workload varies among tasks, dynamic programming is used to find the optimal solution.

For multicore global speed scaling systems:

» *What are the optimal speeds for global speed scaling?*

In Chapter 5 we studied the situation where all tasks on a global speed scaling system have a common deadline. The optimal speeds depend on the amount of cores that are active at a given time. The optimal speeds for the time periods wherein n cores are active and the time periods wherein m cores are active are related by $s_m = s_n \sqrt[\alpha]{\frac{n}{m}}$.

This relation was also used for the variant of the problem where all tasks have individual arrival times and deadlines. For this, we used a substitution of variables (in Chapter 6) that was inspired by the above mentioned relation between optimal speeds. With this substitution of variables, the multicore problem can be transformed to a uniprocessor problem, solved using uniprocessor algorithms and then transformed back to obtain the solution to the multicore problem.

- *What characterises the energy minimising schedule?*

  The energy consumption is influenced by both the schedule and the chosen speeds. In order to minimise the energy consumption, both problems have to be solved simultaneously (this problem is NP-hard).

  To characterise the energy minimising schedule, the weighted makespan is used as scheduling criterion. This criterion takes the optimal speeds into account, and by minimising it we also minimise the energy consumption. For two cores, minimising the makespan also minimises the weighted makespan, while this generally does not hold for more than two cores.

- *How well do existing scheduling algorithms minimise the energy consumption?*

  The approximation ratio given by Theorem 5.2 can be used to determine how effective many scheduling algorithms from the literature are at minimising the energy consumption. For the LPT scheduling algorithm specifically, our simulations showed that energy savings of more than 30% with respect to the state-of-the-art work can be achieved (Section 5.5.2).

8.3 Recommendations for future research

8.3.1 Online global speed scaling

Section 6.4 suggests how online scaling can be used for global speed scaling systems. Herein, we assumed that a schedule is given. The situation in which a schedule must be determined is not considered and is left for future research. This is a difficult problem, since feasibility must be ensured. The research from Chapter 5 suggests that a scheduling algorithm inspired by LPT may work well, but we did not evaluate this.

However, both the evaluation from Chapter 4, on online speed scaling, and the evaluation from Chapter 5, on global speed scaling, suggest that a combination of our online speed scaling and global speed scaling techniques may be a fruitful direction for future research.

8.3.2 Local speed scaling for tasks with precedence constraints

Local speed scaling for tasks with precedence constraints is an unsolved and important theoretical problem. Even the case where the tasks have been scheduled and only speeds need to be determined ($P_M \mid a_n{=}a; d_n{=}d; \text{prec}; \text{sched} \mid E$) is currently unsolved. The power equality (discussed in Section 2.6.6) can be used as a first step toward solving the problem.

The following example illustrates why this problem may be difficult.

Example 8.1. *Consider the power function $p(s) = s^3$ for a three processor system with local speed scaling. The tasks have precedence constraints as given in Figure 8.1a. All tasks share the common deadline $d = 1$.*

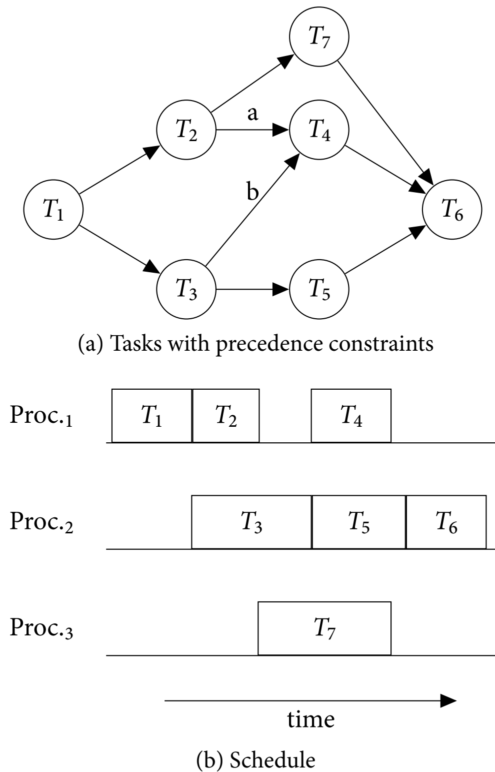


Figure 8.1 – Precedence constraints and schedule for Example 8.1.

We keep the work a variable in this example, to demonstrate the influence of the work on the solution. The schedule (with some arbitrarily chosen workload) is given in Figure 8.1b. Note, that the position of the gaps in the schedule will change when the workload changes. The optimisation problem is: for a given schedule (processor assignment and ordering), what is the optimal speed assignment that minimises the energy consumption, respects precedence constraints and meets the deadline?

Due to convexity of the power function, in the optimal solution it must hold that $s_1 = s_6$*. To ease the discussion, we consider two situations:*

(a) Task T_2 *finishes before task* T_3*, or at the same time.*
In the discussion below, we may assume that the edge "a" between task T_2 *and* T_4 *does not exist, as (with the given assumption) it does not influence the optimal solution. In the optimal solution, we have* $e_2 + e_7 = e_3 + e_4 = e_3 + e_5$ *(same execution time for tasks, avoiding gaps in the schedule), otherwise the energy consumption can be decreased by decreasing the speed of a task that is next to a gap in the schedule. These relations can be used to determine the speeds of these tasks. Using the power equality, the relation between the speeds* s_3*,* s_4 *and* s_5 *can*

be determined. It can also be used to relate speeds s_1*,* s_2 *and* s_3*. Now enough information is available to find the optimal speeds.*

(b) *Task* T_2 *finishes after task* T_3*.*
In the discussion below, we may assume that the edge "b" between tasks T_3 *and* T_4 *does not exist, as (with the given assumption) it does not influence the optimal solution. Then it holds for the optimal solution that* $e_2 + e_7 = e_2 + e_4 = e_3 + e_5$*. Again, using the convexity of the power function and using the power equality, the optimal speeds can be determined.*

A straightforward method for finding the optimal speeds is by calculating the energy consumption for both situations and selecting the one with the lowest costs.

This example suggests that the local speed scaling problem with a given schedule of tasks with precedence constraints may be difficult. The continuous problem requires discrete decisions, namely whether some task finishes before or after some other task. It is unclear how many of these decision points may occur, and if there is an efficient (polynomial time) algorithm to make these decisions.

8.3.3 measurements on real systems

The common approach in many papers on algorithmic power management (see, e.g., the surveys [3, 45] and Chapter 3 of this thesis), is to use simulations to quantify the energy reduction that results from using algorithmic power management algorithms. It would be beneficial to verify power management algorithms on real hardware, to see how well they perform. Especially nonuniform power due to different types of tasks (with different switching characteristics) is rarely researched. Although some of our algorithms are robust against modelling errors (see Chapter 4), the algorithms may be improved by using the knowledge that is obtained from measurements on real systems.

8.3.4 influence of shared resources

In this thesis, the influence of caches, arbitration on buses, etc., was ignored. Without this assumption, it is not possible to draw deterministic conclusions. In future research, the measured influence of this assumption and stochastic methods should be considered.

Appendix **A**

Mathematical Background

This thesis uses convex optimisation, heuristic algorithms and scheduling theory. This appendix provides a short introduction to convex optimisation (Section A.1), heuristic algorithms (Section A.2) and list scheduling (Section A.3).

A.1 Convex optimisation

This section contains a short introduction to convex optimisation, for details see Boyd and Vandenberghe [21].

A.1.1 convex sets

An important concept in this thesis is a *convex set*.

Definition A.1 (Convex set). *A set $\mathcal{C}$ is convex if and only if:*

$$\forall x, y \in \mathcal{C}, \lambda \in [0,1] : \lambda x + (1-\lambda) y \in \mathcal{C}.$$

Intuitively, this means that whenever two values are in a set, all values on a straight line between these two values are also within this set. In this thesis, convex sets are mainly used for decision variables and inequality constraints (to be discussed).

A.1.2 convex functions

Definition A.2 (Convex function). *A (vector) function $f : \mathcal{C} \to \mathbb{R}$ on a convex set $\mathcal{C}$ is convex if and only if:*

$$\forall x, y \in \mathcal{C}, \lambda \in [0,1] : f(\lambda x + (1-\lambda) y) \leq \lambda f(x) + (1-\lambda) f(y).$$

It is intuitively easy to explain a convex function using a scalar function. When a line is drawn between any two function values (e.g., $f(x)$ and $f(y)$), the function

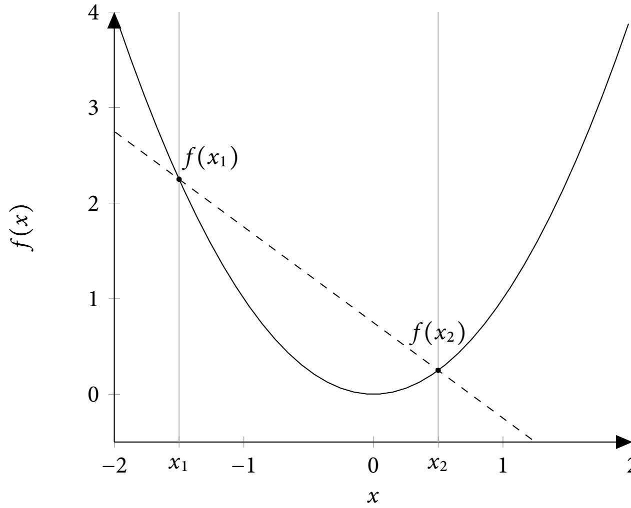


Figure A.1 – Convex function.

lies beneath this line. This intuition is depicted in Figure A.1. Examples of scalar convex functions are $f(x) = ax + b$, $f(x) = x^2$ and $f(x) = e^x$.

The above intuition is generalised by Jensen's inequality, which shows that for any set of function arguments $x_1, \ldots, x_n$, the function value at the average of these arguments is below the average of function values of the individual arguments. The discrete version of Jensen's inequality is as follows:

Definition A.3 (Jensen's inequality (discrete)). *Let $f : \mathcal{C} \to \mathbb{R}$ be a convex function on a convex set $\mathcal{C}$. Furthermore, let $x_1, \ldots, x_N \in \mathcal{C}$ and $\lambda_1, \ldots, \lambda_N \in \mathbb{R}_0^+$. Then*

$$f\left(\frac{\sum_{n=1}^{N} \lambda_n x_n}{\sum_{n=1}^{N} \lambda_n}\right) \leq \frac{\sum_{n=1}^{N} \lambda_n f(x_n)}{\sum_{n=1}^{N} \lambda_n}.$$

The continuous variant of Jensen's inequality does not consider individual function arguments $x_1, \ldots, x_N$, but a function $g : \mathcal{X} \to \mathcal{C}$ that gives the function arguments that are considered.

Definition A.4 (Jensen's inequality (continuous)). *Let $f : \mathcal{C} \to \mathbb{R}$ be a convex function on a convex set $\mathcal{C}$, and let $g : \mathcal{X} \to \mathcal{C}$ be a function such that $\int_{\mathcal{X}} g(x)dx = 1$ (i.e. $\mathcal{X}$ has measure 1). Then*

$$f\left(\int_{\mathcal{X}} g(x)dx\right) \leq \int_{\mathcal{X}} f(g(x))dx.$$

Besides that convex functions often occur in practice, they have properties that are favourable for optimisation. One of these properties is given by the following lemma.

Lemma A.1. *Any local minimiser of a convex function over a convex set is also a global minimiser of this function.*

Proof. See Boyd and Vandenberghe [21]. □

Related to convex functions are concave functions, which are defined as:

Definition A.5 (Concave function)**.** *A (vector) function f is concave, if and only if $-f$ is convex.*

Examples of concave functions are $f(x) = ax + b$, $f(x) = \sqrt[\alpha]{x}$ (for $\alpha \geq 1$, $x \geq 0$) and $f(x) = -x^2$.

A.1.3 Convex optimisation

In general, a convex optimisation problem is given in the following form.

Optimisation Problem A.1 (Convex optimisation problem)**.** *Let $\mathcal{C}$ be a convex set and let $f : \mathcal{C} \to \mathbb{R}$, $g_1 : \mathcal{C} \to \mathbb{R}, \dots, g_N : \mathcal{C} \to \mathbb{R}$ be convex functions. Then the following optimisation problem is a* convex optimisation problem.

$$\begin{aligned} \min_{x \in \mathcal{C}} \quad & f(x), \\ \textit{s.t.} \quad & g_n(x) \leq 0, \textit{ for } n \in \{1, \dots, N\}. \end{aligned}$$

The constraint functions $g_1, \dots, g_N$ restrict variables in the convex set $\mathcal{C}$ to a subset of allowed solutions. This set $\mathcal{F} = \{x \in \mathcal{C} \mid g_n(x) \leq 0,\ \ n \in \{1, \dots, N\}\}$ is called the feasible set, or also called the set of *feasible solutions*. To solve the minimisation problem, we must find the feasible solution that minimises the convex function $f(x)$ (called the cost function).

The feasible set has an important property, stated by the following lemma.

Lemma A.2 (Convexity of a feasible set)**.** *The feasible set of a convex optimisation problem is convex.*

Proof. See Boyd and Vandenberghe [21]. □

This lemma shows (together with Lemma A.1) that a local minimiser of a convex optimisation problem is also a global minimiser. This guarantees that, when an iterative technique that gradually descends towards the minimiser is used (a descent technique), the end result is globally optimal.

Besides iterative techniques, there are also direct techniques for finding the global minimiser. An example of such an approach is solving the Karush Kuhn Tucker (KKT) conditions. The KKT conditions are a set of equations that, when solved, give the global minimiser to a convex optimisation problem. Because the KKT conditions are not used this thesis, they are not presented here and the interested reader is referred to the textbook by Boyd and Vandenberghe [21] for details.

A.2 Heuristic algorithms

Many problems that are studied in this thesis are NP-hard. For this reason *heuristic algorithms* are used, which are algorithms that aim at finding a solution to the problem that lies close to the optimal solution. To quantify how good such an algorithm is, the notion of ρ−approximation is used to quantify how close the solution approaches the optimal solution. This is defined as follows.

Definition A.6 (ρ-approximation algorithm). *A heuristic algorithm is referred to as a ρ-approximation algorithm (with* approximation ratio ρ*), when it is guaranteed that for each problem instance we have*

$$x \leq \rho \, OPT,$$

where $\rho \geq 1$, OPT are the optimal costs, and x are the costs found by the algorithm.

The approximation ratio is used to bound the inaccuracy of a heuristic. In the case that $\rho = 1$, the algorithm is guaranteed to find the optimal solution. For some problems (for example the Travelling Salesman Problem) it can be proven that no ρ-approximation exists, in such cases the problem is said to be *inapproximable*.

For some classes of problems, ρ-approximation heuristics exist for every $\rho = 1 + \epsilon$ (where $\epsilon > 0$). For this family of heuristics, an algorithm with a better approximation ratio is obtained by decreasing ϵ toward zero. This family of heuristics is called a Polynomial Time Approximation Scheme (PTAS), and is defined by.

Definition A.7 (Polynomial Time Approximation Scheme). *A PTAS is an algorithm with parameter ϵ that is polynomial in the input size and, for every $\epsilon > 0$, it yields a $(1 + \epsilon)$-approximation.*

Note, that a PTAS has a polynomial time complexity for a fixed ϵ, but the complexity of the algorithm may grow exponentially with $1/\epsilon$.

It is desirable to have a PTAS with a time complexity that is also polynomially bounded in $1/\epsilon$, otherwise the algorithm may not be feasible for small ϵ. Such approximation scheme is called a *Fully Polynomial Time Approximation Scheme* (FPTAS).

A.3 List scheduling

This section contains a short introduction to scheduling, for details see Pinedo [70].

In Chapter 5 of this thesis, we consider tasks with precedence constraints. Herein we consider N tasks where task T_n has execution time e_n. These tasks can have precedence constraints $T_n \prec T_m$, which means task T_n must finish before task T_m may start.

A popular objective for scheduling problems is makespan (schedule length) minimisation, where the N tasks with precedence constraints are scheduled on M processors. Since this scheduling problem is NP-hard, this problem is not solved exactly, but heuristics are used. A popular scheduling algorithm is *list scheduling*, a scheduling algorithm that receives as input a list of tasks in order of their priority. This algorithm takes the highest priority task from the list for which all preceding tasks have been scheduled, and assigns this task to the processor with the earliest end time. This procedure is repeated until all tasks are scheduled.

The effectiveness of list scheduling depends on the priorities, i.e. the order in which the tasks in the list are sorted. In this thesis, the Longest Processing Time (LPT) priority (or rule) for list scheduling is used (referred to as the LPT algorithm), which means that tasks are sorted in (decreasing) order of execution time. This is a $(4/3 - 1/(3M))$-approximation algorithm [34]. For the case that there are no precedence constraints, there exists a PTAS for this problem [39].

Appendix B

Problem Notation

The notation for algorithmic power management from Section 2.5 is repeated in Table B.1.

Table B.1 – Notation for algorithmic power management problems.

Field	Entry	Meaning
$\mathfrak{a}$	1	Single processor
	P_M	M parallel processors
	ss	Speed scaling is supported
	nonunif	A nonuniform power function is used (ss implied)
	disc	Discrete speed scaling is used (ss implied)
	global	Global speed scaling is used (ss implied)
	sl	Sleep modes supported
$\mathfrak{b}$	a_n	Arrival time
	$a_n{=}a$	Same arrival time a for all tasks
	d_n	Deadline constraint
	$d_n{=}d$	Same deadline constraint d for all tasks
	$w_n{=}w$	All tasks have workload w
	agree	Agreeable deadlines ($a_n \leq a_m \Leftrightarrow d_n \leq d_m$)
	lami	Laminar instances ($[a_i, d_i] \subset [a_j, d_j] \vee [a_j, d_j] \subset [a_i, d_i] \vee [a_i, d_i] \cap [a_j, d_j] = \varnothing$)
	prec	Tasks have precedence constraints
	pmtn	Preemptions are allowed
	prio	Tasks have a fixed priority
	migr	Task migration is allowed
	sched	A schedule is given
$\mathfrak{c}$	E	Minimise the energy consumption

Acronyms

A	ACPI	Advanced Configuration and Power Interface
C	CMP	Chip Multi Processor
D	DAG	Directed Acyclic Graph
	DPM	Dynamic Power Management
	DSP	Digital Signal Processing
	DVFS	Dynamic Voltage and Frequency Scaling
E	EDF	Earliest Deadline First
G	GOP	Group Of Pictures
I	ICT	Information and Communications Technology
	ILP	Integer Linear Program
K	KKT	Karush Kuhn Tucker
L	LPT	Longest Processing Time
P	PRA-SS	Periodic Robust and Adaptive Speed Scaling
	PTAS	Polynomial Time Approximation Scheme
R	RA-SS	Robust and Adaptive Speed Scaling
	RM	Rate Monotonic
S	STG	Standard Task Graph
W	WCW	Worst Case Work

Nomenclature

General notation

α	Exponent for dynamic power, page 15
γ_1	Constant for dynamic power, page 14
γ_2	Constant for static power, page 15
γ_3	Constant for static power, page 15
$\mathcal{S}$	Set of available speeds, page 16
$\mathcal{W}$	Some constant work, page 83
a_n	Arrival time of task T_n, page 12
b_n	Begin time of task T_n, page 12
c_n	Completion time of task T_n, page 12
d_n	Deadline of task T_n, page 12
e_n	Execution time of task T_n, page 12
K	Number of discrete speeds, page 16
M	Number of processors/cores, page 12
N	Number of tasks, page 12
$p(s)$	Power function, page 16

$\bar{p}(s)$	Energy-per-work function, page 21
$s(t)$	Speed function, page 16
s^{crit}	Critical speed, page 21
$s^{\max}$	Upper bound for s_n / $s(t)$, page 41
$s^{\min}$	Lower bound for s_n / $s(t)$, page 41
s_n	Speed of task T_n, page 16
$\bar{s}_k$	k-th discrete speed, page 22
t^B	Start time of application, page 21
t^C	End time of application, page 21
$w^{\max}$	Upper bound for work (Worst Case Work), page 41
w_n	Work of task T_n, page 12
$w_{n,k}$	Work for task T_n at discrete speed $\bar{s}_k$, page 22

Chapter 4

ρ	Prediction window size, page 53
a^0	Phase of arrival times, page 52
d^0	Phase of deadlines, page 52
$\hat{d}_k(\hat{w}_k)$	Robust deadline of task T_k depending on $\hat{w}_k$, page 50
T	Period length, page 52
w^{AVG}	Average work, page 53
$\hat{w}_k$	Prediction of work of task T_k, page 41

CHAPTER 5

β	Approximation ratio for makespan, page 80
ω_m	Work for m active cores, page 68
$\bar{\beta}$	Approximation ratio for weighted makespan, page 80
d	Deadline of application, page 66
I_n	Length of i-th interval, page 69
$p_m(s)$	Power function for m active cores, page 67
$\bar{p}_m(s)$	Energy-per-work function for m active cores, page 67
S	Makespan, page 66
s_m	Speed for m active cores, page 68
$\bar{S}$	Weighted makespan, page 74
W	Total work, page 68

CHAPTER 6

Π_k	Piece, page 89
a_n	Arrival time piece Π_n, page 90
b_n	Begin time piece Π_n, page 90
c_n	Completion time piece Π_n, page 90
d_n	Deadline piece Π_n, page 90
m_n	Number of active cores piece Π_n, page 90
N	Number of pieces, page 89
s_n	Speed piece Π_n, page 90
$\mathring{s}_n$	Speed task T_n (equivalent uniprocessor problem), page 91
w_n	Work piece Π_n, page 90
$\mathring{w}_n$	Work task T_n (equivalent uniprocessor problem), page 91

(v_i, v_j)	Edge, page 109
$\omega_{i,j}$	Weight for edge (v_i, v_j), page 109
$B_{m,\ell}$	Break-even time of device m to and from sleep mode ℓ, page 100
D	Number of linear pieces, page 101
E^{sl}	Idle-time energy function, page 100
$E_{m,\ell}$	Transition energy of device m to and from sleep mode ℓ, page 100
I_n	Length of idle period in n-th frame, page 99
$P_{m,\ell}$	Power of device m in sleep mode ℓ, page 100
Q_i	Constant for linear piece i, page 101
R_i	Constant for linear piece i, page 101
$s_n(t)$	Speed function for nth frame, page 102
T	Period length, page 99
$T_{m,\ell}$	Transition latency of device m to and from sleep mode ℓ, page 100
v_n	Vertex, page 109

Bibliography

[1] "Xerf's test media collection," April 2013. [Online]. Available: http://media.xiph.org/video/derf/ (Cited on page 55).

[2] ACPI, "Advanced configuration and power interface standard," 2011. [Online]. Available: http://www.acpi.info (Cited on page 97).

[3] S. Albers, "Energy-efficient algorithms," *Commun. ACM*, vol. 53, no. 5, pp. 86–96, may 2010. [Online]. Available: http://doi.acm.org/10.1145/1735223.1735245 (Cited on page 129).

[4] S. Albers and A. Antoniadis, "Race to idle: new algorithms for speed scaling with a sleep state," in *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '12. SIAM, 2012, pp. 1266–1285. [Online]. Available: http://dl.acm.org/citation.cfm?id=2095116.2095216 (Cited on pages 28 and 32).

[5] S. Albers, F. Müller, and S. Schmelzer, "Speed scaling on parallel processors," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '07. New York, NY, USA: ACM, 2007, pp. 289–298. [Online]. Available: http://doi.acm.org/10.1145/1248377.1248424 (Cited on pages 33, 34, and 35).

[6] S. Albers, A. Antoniadis, and G. Greiner, "On multi-processor speed scaling with migration: extended abstract," in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 279–288. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989539 (Cited on pages 25, 28, 33, and 34).

[7] A. Alimonda, S. Carta, A. Acquaviva, and A. Pisano, "Non-linear feedback control for energy efficient on-chip streaming computation," in *Industrial Embedded Systems, 2006. IES'06. International Symposium on*, Oct 2006, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/IES.2006.357456 (Cited on page 35).

[8] N. Alon, Y. Azar, G. J. Woeginger, and T. Yadid, "Approximation schemes for scheduling," in *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '97. Philadelphia, PA, USA: Society for Industrial and

Applied Mathematics, 1997, pp. 493–500. [Online]. Available: http://dl.acm.org/citation.cfm?id=314161.314371 (Cited on page 34).

[9] E. Angel, E. Bampis, and V. Chau, "Low complexity scheduling algorithm minimizing the energy for tasks with agreeable deadlines," in *LATIN 2012: Theoretical Informatics*, D. Fernández-Baca, Ed. Springer Berlin Heidelberg, 2012, vol. 7256, ch. Lecture Notes in Computer Science, pp. 13–24. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29344-3_2 (Cited on pages 28, 32, 33, and 104).

[10] E. Angel, E. Bampis, F. Kacem, and D. Letsios, "Speed scaling on parallel processors with migration," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. Spirakis, Eds. Springer Berlin Heidelberg, 2012, vol. 7484, ch. Lecture Notes in Computer Science, pp. 128–140. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_15 (Cited on pages 25, 33, and 34).

[11] A. Antoniadis and C.-C. Huang, "Non-preemptive speed scaling," in *Algorithm Theory – SWAT 2012*, F. Fomin and P. Kaski, Eds. Springer Berlin Heidelberg, 2012, vol. 7357, ch. Lecture Notes in Computer Science, pp. 249–260. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31155-0_22 (Cited on pages 13, 28, and 31).

[12] J. Augustine, S. Irani, and C. Swamy, "Optimal power-down strategies," *SIAM Journal on Computing*, vol. 37, no. 5, pp. 1499–1516, oct. 2008. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/05063787X (Cited on pages 17 and 100).

[13] H. Aydin and Q. Yang, "Energy-aware partitioning for multiprocessor real-time systems," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 113–121. [Online]. Available: http://dl.acm.org/citation.cfm?id=838237.838347 (Cited on page 36).

[14] E. Bampis, C. Dürr, F. Kacem, and I. Milis, "Speed scaling with power down scheduling for agreeable deadlines," *Sustainable Computing: Informatics and Systems*, vol. 2, no. 4, pp. 184–189, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.suscom.2012.10.003 (Cited on pages 28 and 33).

[15] E. Bampis, D. Letsios, and G. Lucarelli, "Green scheduling, flows and matchings," in *Algorithms and Computation*, K.-M. Chao, T.-s. Hsu, and D.-T. Lee, Eds. Springer Berlin Heidelberg, 2012, vol. 7676, ch. Lecture Notes in Computer Science, pp. 106–115. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35261-4_14 (Cited on pages 25, 33, and 34).

[16] E. Bampis, A. Kononov, D. Letsios, G. Lucarelli, and I. Nemparis, "From preemptive to non-preemptive speed-scaling scheduling," in *Computing and Combinatorics*, D.-Z. Du and G. Zhang, Eds. Springer Berlin Heidelberg, 2013, vol. 7936, ch. Lecture Notes in Computer Science, pp. 134–146. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38768-5_14 (Cited on pages 18, 28, 31, 32, 33, 35, and 42).

[17] N. Bansal, T. Kimbrel, and K. Pruhs, "Speed scaling to manage energy and temperature," *J. ACM*, vol. 54, no. 1, pp. 3:1–3:39, mar 2007. [Online]. Available: http://doi.acm.org/10.1145/1206035.1206038 (Cited on pages 28 and 30).

[18] N. Bansal, H.-L. Chan, and K. Pruhs, "Speed scaling with an arbitrary power function," in *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '09. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2009, pp. 693–701. [Online]. Available: http://dl.acm.org/citation.cfm?id=1496770.1496846 (Cited on page 21).

[19] P. Baptiste, M. Chrobak, and C. Dürr, "Polynomial-time algorithms for minimum energy scheduling," *ACM Trans. Algorithms*, vol. 8, no. 3, pp. 26:1–26:29, jul 2012. [Online]. Available: http://doi.acm.org/10.1145/2229163.2229170 (Cited on pages 28, 32, and 38).

[20] B. D. Bingham and M. R. Greenstreet, "Energy optimal scheduling on multiprocessors with migration," in *Parallel and Distributed Processing with Applications, 2008. ISPA '08. International Symposium on*, Dec 2008, pp. 153–161. [Online]. Available: http://dx.doi.org/10.1109/ISPA.2008.128 (Cited on pages 33 and 34).

[21] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004. (Cited on pages 30, 131, and 133).

[22] D. P. Bunde, "Power-aware scheduling for makespan and flow," in *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '06. New York, NY, USA: ACM, 2006, pp. 190–196. [Online]. Available: http://doi.acm.org/10.1145/1148109.1148140 (Cited on page 36).

[23] S. Carta, A. Alimonda, A. Pisano, A. Acquaviva, and L. Benini, "A control theoretic approach to energy-efficient pipelined computation in MPSoCs," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 4, pp. 27:1–27:28, sep 2007. [Online]. Available: http://doi.acm.org/10.1145/1274858.1274865 (Cited on page 35).

[24] P. Chaparro, J. González, G. Magklis, C. Qiong, and A. González, "Understanding the thermal implications of multi-core architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 8, pp. 1055–1065, aug. 2007. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2007.1092 (Cited on pages 7, 14, 36, and 64).

[25] J.-J. Chen and C.-F. Kuo, "Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms," in *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, 2007, pp. 28–38. [Online]. Available: http://dx.doi.org/10.1109/RTCSA.2007.37 (Cited on page 7).

[26] S. Cho and R. G. Melhem, "On the interplay of parallelization, program performance, and energy consumption," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 3, pp. 342–353, mar 2010. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2009.41 (Cited on pages 15, 64, 66, and 88).

[27] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram, "Frame-based dynamic voltage and frequency scaling for a MPEG decoder," in *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '02. New York, NY, USA: ACM, 2002, pp. 732–737. [Online]. Available: http://doi.acm.org/10.1145/774572.774680 (Cited on pages 35, 40, 56, and 57).

[28] S. Clevers and R. Verweij, "ICT stroomt door," Ministerie van Economische Zaken, Den Haag, Tech. Rep., 2007. (Cited on page 1).

[29] E. G. Coffman and R. L. Graham, "Optimal scheduling for two-processor systems," *Acta Informatica*, vol. 1, no. 3, pp. 200–213, 1972. [Online]. Available: http://dx.doi.org/10.1007/BF00288685 (Cited on page 82).

[30] P. de Langen and B. Juurlink, "Leakage-aware multiprocessor scheduling," *Journal of Signal Processing Systems*, vol. 57, no. 1, pp. 73–88, 2009. (Cited on page 36).

[31] V. Devadas and H. Aydin, "Real-time dynamic power management through device forbidden regions," in *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, april 2008, pp. 34–44. [Online]. Available: http://dx.doi.org/10.1109/RTAS.2008.21 (Cited on pages 98 and 105).

[32] V. Devadas and H. Aydin, "On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications," *Computers, IEEE Transactions on*, vol. 61, no. 1, pp. 31–44, Januari 2012. [Online]. Available: http://dx.doi.org/10.1109/TC.2010.248 (Cited on pages 14, 37, 38, 66, 98, 99, 100, 101, 114, 115, 116, and 119).

[33] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959, 10.1007/BF01386390. [Online]. Available: http://dx.doi.org/10.1007/BF01386390 (Cited on page 110).

[34] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969. [Online]. Available: http://www.jstor.org/stable/2099572 (Cited on pages 84 and 135).

[35] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," *Annals of Discrete Mathematics. v5*, pp. 287–326, 1977. [Online]. Available: http://dx.doi.org/10.1016/S0167-5060(08)70356-X (Cited on page 18).

[36] M. Grant and S. Boyd, "CVX: Matlab software for disciplined convex programming, version 1.21," apr 2011. [Online]. Available: http://cvxr.com/cvx/ (Cited on page 85).

[37] G. Greiner, T. Nonner, and A. Souza, "The bell is ringing in speed-scaled multiprocessor scheduling," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 11–18. [Online]. Available: http://doi.acm.org/10.1145/1583991.1583996 (Cited on pages 33 and 34).

[38] J.-J. Han, X. Wu, D. Zhu, H. Jin, L. T. Yang, and J. L. Gaudiot, "Synchronization-aware energy management for VFI-based multicore real-time systems," *Computers, IEEE Transactions on*, vol. 61, no. 12, pp. 1682–1696, Dec 2012. [Online]. Available: http://dx.doi.org/10.1109/TC.2012.136 (Cited on pages 36 and 66).

[39] D. S. Hochbaum and D. B. Shmoys, "A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach," *SIAM J. Comput.*, vol. 17, no. 3, pp. 539–551, jun 1988. [Online]. Available: http://dx.doi.org/10.1137/0217033 (Cited on pages 83 and 135).

[40] A. Holtzman *et al.*, "libmpeg2," July 2008. [Online]. Available: http://libmpeg2.sourceforge.net (Cited on page 54).

[41] C.-h. Hsu and W.-c. Feng, "When discreteness meets continuity: Energy-optimal DVS scheduling revisited," Los Alamos National Laboratory, Tech. Rep. LA-UR 05-3104, February 2005. [Online]. Available: http://sss.cs.vt.edu/pubs/tr05-3104.pdf (Cited on pages 21, 23, 28, and 30).

[42] F. Hu and J. Evans, "Power and environment aware control of Beowulf clusters," *Cluster Computing*, vol. 12, no. 3, pp. 299–308, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10586-009-0085-z (Cited on page 17).

[43] W. Huang and Y. Wang, "An optimal speed control scheme supported by media servers for low-power multimedia applications," *Multimedia Systems*, vol. 15, no. 2, pp. 113–124, 2009. [Online]. Available: http://dx.doi.org/10.1007/s00530-009-0153-5 (Cited on pages 14, 20, 23, 28, 32, 39, 44, 88, and 123).

[44] X. Huang, K. Li, and R. Li, "A energy efficient scheduling base on dynamic voltage and frequency scaling for multi-core embedded real-time system," in *Algorithms and Architectures for Parallel Processing*, A. Hua and S.-L. Chang, Eds. Springer Berlin / Heidelberg, 2009, vol. 5574, ch. Lecture Notes in Computer Science, pp. 137–145, 10.1007/978-3-642-03095-6_14. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03095-6_14 (Cited on page 36).

[45] S. Irani and K. R. Pruhs, "Algorithmic problems in power management," *SIGACT News*, vol. 36, no. 2, pp. 63–76, jun 2005. [Online]. Available: http://doi.acm.org/10.1145/1067309.1067324 (Cited on page 129).

[46] S. Irani, S. Shukla, and R. Gupta, "Algorithms for power savings," in *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '03. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, pp. 37–46. [Online]. Available: http://dl.acm.org/citation.cfm?id=644108.644115 (Cited on pages 17 and 100).

[47] S. Irani, S. Shukla, and R. Gupta, "Algorithms for power savings," *ACM Trans. Algorithms*, vol. 3, no. 4, pp. 41:1–41:23, nov 2007. [Online]. Available: http://doi.acm.org/10.1145/1290672.1290678 (Cited on pages 20, 21, 22, 28, 30, 32, 39, and 43).

[48] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *Proceedings of the 1998 international symposium on Low power electronics and design*, ser. ISLPED '98. New York, NY, USA: ACM, 1998, pp. 197–202. [Online]. Available: http://doi.acm.org/10.1145/280756.280894 (Cited on pages 14, 39, and 102).

[49] R. Jejurikar, C. Pereira, and R. Gupta, "Leakage aware dynamic voltage scaling for real-time embedded systems," in *Design Automation Conference, 2004. Proceedings. 41st*, ser. DAC '04. New York, NY, USA: ACM, 2004, pp. 275–280. [Online]. Available: http://doi.acm.org/10.1145/996566.996650 (Cited on page 22).

[50] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd, "Power7: IBM's next-generation server processor," *Micro, IEEE*, vol. 30, no. 2, pp. 7–15, march-april 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.38 (Cited on pages 7, 14, and 64).

[51] A. Kandhalu, J. Kim, K. Lakshmanan, and R. R. Rajkumar, “Energy-aware partitioned fixed-priority scheduling for chip multi-processors,” in *17th International Conference on Embedded and Real-Time Computing Systems and Applications*, vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 93–102. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/RTCSA.2011.75 (Cited on pages 7, 14, 36, and 64).

[52] F. Kong, Y. Wang, Q. Deng, and W. Yi, “Minimizing multi-resource energy for real-time systems with discrete operation modes,” in *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, july 2010, pp. 113–122. [Online]. Available: http://dx.doi.org/10.1109/ECRTS.2010.18 (Cited on pages 38, 98, 99, and 119).

[53] Y.-K. Kwok and I. Ahmad, “Static scheduling algorithms for allocating directed task graphs to multiprocessors,” *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, dec 1999. [Online]. Available: http://doi.acm.org/10.1145/344588.344618 (Cited on pages 36 and 75).

[54] W.-C. Kwon and T. Kim, “Optimal voltage allocation techniques for dynamically variable voltage processors,” *ACM Transactions on Embedded Computing Systems*, vol. 4, no. 1, pp. 211–230, 2005. [Online]. Available: http://dx.doi.org/10.1145/1053271.1053280 (Cited on pages 15, 23, 24, 25, 28, 30, 31, 91, and 102).

[55] S. Lee and J. Kim, “Using dynamic voltage scaling for energy-efficient flash-based storage devices,” in *SoC Design Conference (ISOCC), 2010 International*, 2010, pp. 63–66. [Online]. Available: http://dx.doi.org/10.1109/SOCDC.2010.5682971 (Cited on page 2).

[56] W.-K. Lee, S.-W. Lee, and W.-O. Siew, “Hybrid model for dynamic power management,” *Consumer Electronics, IEEE Transactions on*, vol. 55, no. 2, pp. 656–664, may 2009. [Online]. Available: http://dx.doi.org/10.1109/TCE.2009.5174436 (Cited on page 17).

[57] Y. C. Lee and A. Y. Zomaya, “Energy conscious scheduling for distributed computing systems under different operating conditions,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 8, pp. 1374–1381, aug. 2011. (Cited on page 36).

[58] K. Li, “Energy efficient scheduling of parallel tasks on multiprocessor computers,” *The Journal of Supercomputing*, vol. 60, no. 2, pp. 223–247, 2012. [Online]. Available: http://dx.doi.org/10.1007/s11227-010-0416-0 (Cited on page 64).

[59] K. Li, “Scheduling precedence constrained tasks with reduced processor energy on multiprocessor computers,” *Computers, IEEE Transactions on*, vol. 61, no. 12, pp. 1668–1681, 2012. [Online]. Available: http://dx.doi.org/10.1109/TC.2012.120 (Cited on pages 37, 64, 66, 83, and 84).

[60] M. Li, B. Liu, and F. Yao, “Min-energy voltage allocation for tree-structured tasks,” *Journal of Combinatorial Optimization*, vol. 11, no. 3, pp. 305–319, 2006. [Online]. Available: http://dx.doi.org/10.1007/s10878-006-7910-6 (Cited on pages 13, 20, 28, and 33).

[61] M. Li, A. C. Yao, and F. F. Yao, "Discrete and continuous min-energy schedules for variable voltage processors." *Proc. Natl. Acad. Sci. U.S.A.*, vol. 103, no. 11, pp. 3983–3987, 2006. [Online]. Available: http://dx.doi.org/10.1073/pnas.0510886103 (Cited on pages 28 and 30).

[62] Y.-H. Lu, L. Benini, and G. De Micheli, "Operating-system directed power reduction," in *Proceedings of the 2000 international symposium on Low power electronics and design*, ser. ISLPED '00. New York, NY, USA: ACM, 2000, pp. 37–42. [Online]. Available: http://doi.acm.org/10.1145/344166.344189 (Cited on page 17).

[63] Y.-H. Lu, L. Benini, and G. De Micheli, "Power-aware operating systems for interactive systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 10, no. 2, pp. 119–134, april 2002. [Online]. Available: http://dx.doi.org/10.1109/92.994989 (Cited on page 17).

[64] Z. Lu, J. Lach, M. Stan, and K. Skadron, "Reducing multimedia decode power using feedback control," in *Proceedings 21st International Conference on Computer Design*, 2003, pp. 489–496. [Online]. Available: http://dx.doi.org/10.1109/ICCD.2003.1240945 (Cited on page 35).

[65] J. Luo and N. K. Jha, "Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems," in *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '00. Piscataway, NJ, USA: IEEE Press, 2000, pp. 357–364. [Online]. Available: http://dl.acm.org/citation.cfm?id=602902.602983 (Cited on page 36).

[66] J. L. March, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato, "A new energy-aware dynamic task set partitioning algorithm for soft and hard embedded real-time systems," *The Computer Journal*, vol. 54, no. 8, pp. 1282–1294, 2011. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxr008 (Cited on pages 7, 14, 36, and 64).

[67] A. Nelson, O. Moreira, A. Molnos, S. Stuijk, B. T. Nguyen, and K. Goossens, "Power minimisation for real-time dataflow applications," in *2011 14th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, August 2011, pp. 117–124. [Online]. Available: http://dx.doi.org/10.1109/DSD.2011.19 (Cited on page 56).

[68] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang, "Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 5, pp. 695–708, 2013. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2012.2235126 (Cited on pages 15, 43, and 61).

[69] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 89–102. [Online]. Available: http://doi.acm.org/10.1145/502034.502044 (Cited on pages 35, 56, and 57).

[70] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 4th ed. Springer, 2012. (Cited on pages 35 and 134).

[71] J. A. Pouwelse, K. G. Langendoen, R. L. Lagendijk, and H. J. Sips, "Power-aware video decoding," in *22nd Picture Coding Symposium, Seoul, Korea*. Seoul, Korea: Citeseer, April 2001, pp. 303–306. [Online]. Available: http://www.pds.ewi.tudelft.nl/pubs/papers/pcs2001.pdf (Cited on pages 35, 40, 56, and 57).

[72] K. Pruhs, R. van Stee, and P. Uthaisombut, "Speed scaling of tasks with precedence constraints," in *Approximation and Online Algorithms*, T. Erlebach and G. Persinao, Eds. Springer Berlin / Heidelberg, 2006, vol. 3879, ch. Lecture Notes in Computer Science, pp. 307–319, 10.1007/11671411_24. [Online]. Available: http://dx.doi.org/10.1007/11671411_24 (Cited on pages 23, 24, 33, 34, and 64).

[73] G. Quan and X. S. Hu, "Minimal energy fixed-priority scheduling for variable voltage processors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 22, no. 8, pp. 1062–1071, 2003. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2003.814948 (Cited on pages 28 and 30).

[74] R. Rao and S. Vrudlhula, "Energy optimal speed control of devices with discrete speed sets," in *Proceedings. 42nd Design Automation Conference, 2005.*, 2005, pp. 901–904. [Online]. Available: http://dx.doi.org/10.1109/DAC.2005.193943 (Cited on page 2).

[75] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz, "Bounding energy consumption in large-scale MPI programs," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC '07*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 49–1. [Online]. Available: http://doi.acm.org/10.1145/1362622.1362688 (Cited on pages 23, 36, and 66).

[76] C. Rusu, R. Melhem, and D. Mossé, "Maximizing rewards for real-time applications with energy constraints," *ACM Transactions on Embedded Computing Systems*, vol. 2, no. 4, pp. 537–559, November 2003. [Online]. Available: http://doi.acm.org/10.1145/950162.950166 (Cited on pages 98 and 99).

[77] A. Shye, B. Scholbrock, G. Memik, and P. A. Dinda, "Characterizing and modeling user activity on smartphones: summary," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 1, pp. 375–376, jun 2010. [Online]. Available: http://doi.acm.org/10.1145/1811099.1811094 (Cited on page 1).

[78] A. Sinha and A. Chandrakasan, "Dynamic power management in wireless sensor networks," *Design Test of Computers, IEEE*, vol. 18, no. 2, pp. 62–74, mar/apr 2001. [Online]. Available: http://dx.doi.org/10.1109/54.914626 (Cited on pages 17, 18, 101, 102, and 119).

[79] A. Soria-Lopez, P. Mejia-Alvarez, and J. Cornejo, "Feedback scheduling of power-aware soft real-time tasks," in *Computer Science, 2005. ENC 2005. Sixth Mexican International Conference on*, 2005, pp. 266–273. [Online]. Available: http://dx.doi.org/10.1109/ENC.2005.19 (Cited on page 35).

[80] L. Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 International Conference on Power Aware Computing and Systems, HotPower'10*. USENIX Association, 2010, pp. 1–8. [Online]. Available: https://www.usenix.org/legacy/events/hotpower10/tech/full_papers/LeSueur.pdf (Cited on page 16).

[81] Y. Tan, P. Malani, Q. Qiu, and Q. Wu, "Workload prediction and dynamic voltage scaling for MPEG decoding," in *Proceedings of the 2006 conference on Asia South Pacific design automation - ASP-DAC '06*, 2006, pp. 911–916. (Cited on pages 35 and 36).

[82] T. Tobita and H. Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," *Journal of Scheduling*, vol. 5, no. 5, pp. 379–394, 2002. [Online]. Available: http://dx.doi.org/10.1002/jos.116 (Cited on pages 36 and 84).

[83] Y. Wang, Q. Xie, A. Ammari, and M. Pedram, "Deriving a near-optimal power management policy using model-free reinforcement learning and bayesian classification," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 41–46. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024735 (Cited on page 17).

[84] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Mobile Computing*, T. Imielinski and H. F. Korth, Eds. Springer US, 1996, vol. 353, ch. The Kluwer International Series in Engineering and Computer Science, pp. 449–471, 10.1007/978-0-585-29603-6_17. (Cited on page 14).

[85] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark, "Formal online methods for voltage/frequency control in multiple clock domain microprocessors," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems - ASPLOS-XI*, 2004, pp. 248–259. [Online]. Available: http://dx.doi.org/10.1145/1024393.1024423 (Cited on page 35).

[86] W. Wu, M. Li, and E. Chen, "Min-energy scheduling for aligned jobs in accelerate model," *Theoretical Computer Science*, vol. 412, no. 12–14, pp. 1122–1139, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397510007048 (Cited on pages 28 and 32).

[87] R. Xu, D. Mossé, and R. Melhem, "Minimizing expected energy in real-time embedded systems," in *Proceedings of the 5th ACM international conference on Embedded software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 251–254. [Online]. Available: http://doi.acm.org/10.1145/1086228.1086274 (Cited on page 38).

[88] R. Xu, R. Melhem, and D. Mossé, "A unified practical approach to stochastic DVS scheduling," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, ser. EMSOFT '07. New York, NY, USA: ACM, 2007, pp. 37–46. [Online]. Available: http://doi.acm.org/10.1145/1289927.1289939 (Cited on page 98).

[89] C.-Y. Yang, J.-J. Chen, and T.-W. Kuo, "An approximation algorithm for energy-efficient scheduling on a chip multiprocessor," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 468–473. [Online]. Available: http://dx.doi.org/10.1109/DATE.2005.51 (Cited on pages 66 and 91).

[90] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," in *Proceedings of IEEE 36th Annual Foundations of Computer Science*, 1995, pp. 374–382.

[Online]. Available: http://dx.doi.org/10.1109/SFCS.1995.492493 (Cited on pages 14, 20, 21, 23, 28, 29, 31, 39, 66, 88, and 102).

[91] H.-S. Yun and J. Kim, "On energy-optimal voltage scheduling for fixed-priority hard real-time systems," *ACM Trans. Embed. Comput. Syst.*, vol. 2, no. 3, pp. 393–430, aug 2003. [Online]. Available: http://doi.acm.org/10.1145/860176.860183 (Cited on page 30).

[92] D. Zhang, D. Guo, F. Chen, F. Wu, T. Wu, T. Cao, and S. Jin, "TL-plane-based multi-core energy-efficient real-time scheduling algorithm for sporadic tasks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 47:1–47:20, jan 2012. [Online]. Available: http://doi.acm.org/10.1145/2086696.2086726 (Cited on pages 14 and 36).

[93] D. Zhu, R. Melhem, and B. R. Childers, "Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 14, no. 7, pp. 686–700, july 2003. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2003.1214320 (Cited on pages 36 and 56).

[94] Q. Zhu, F. M. David, C. F. Devaraj, Z. Li, Y. Zhou, and P. Cao, "Reducing energy consumption of disk storage using power-aware cache management," in *Proceedings of the 10th International Symposium on High-Performance Computer Architecture. Madrid.*, Februari 2004, pp. 118–129. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2004.10022 (Cited on page 17).

[95] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of energy-cognizant scheduling techniques," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 7, pp. 1447–1464, July 2013. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2012.20 (Cited on page 36).

[96] T. Zitterell and C. Scholl, "A probabilistic and energy-efficient scheduling approach for online application in real-time systems," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 42–47. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837287 (Cited on page 36).

List of Publications

[MG:1] M. E. T. Gerards and J. Kuper, “Optimal DPM and DVFS for frame-based real-time systems,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 41:1–41:23, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1145/2400682.2400700

[MG:2] M. E. T. Gerards, J. L. Hurink, and J. Kuper, “On the interplay between global DVFS and scheduling tasks with precedence constraints,” 2014, (under review).

[MG:3] M. E. T. Gerards, P. K. F. Hölzenspies, and J. Kuper, “Optimal slack reclamation using robust and adaptive dynamic voltage and frequency scaling,” 2014, (under review).

[MG:4] M. E. T. Gerards, J. L. Hurink, P. K. F. Hölzenspies, J. Kuper, and G. J. M. Smit, “Analytic clock frequency selection for global DVFS,” in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turin, Italy*, 2014, pp. 512–519. [Online]. Available: http://dx.doi.org/10.1109/PDP.2014.103

[MG:5] K. C. H. Blom, M. E. T. Gerards, A. B. J. Kokkeler, and G. J. M. Smit, “Nonminimum-phase channel equalization using all-pass CMA,” in *24th International Symposium on Personal Indoor and Mobile Radio Communications, PIMRC 2013, London, United Kingdom*, 2013, pp. 1467–1471. [Online]. Available: http://dx.doi.org/10.1109/PIMRC.2013.6666373

[MG:6] M. Gerards, C. Baaij, J. Kuper, and M. Kooijman, “Higher-order abstraction in hardware descriptions with CλaSH,” in *Proceedings of the 14th EUROMICRO Conference on Digital System Design, DSD 2011, Oulu, Finland*, 2011, pp. 495–502. [Online]. Available: http://dx.doi.org/10.1109/DSD.2011.69

[MG:7] J. Kuper, C. Baaij, M. Kooijman, and M. Gerards, “Architecture specifications in CλaSH,” in *System Specification and Design Languages*, ser. Lecture Notes in Electrical Engineering, T. J. Kaźmierski and A. Morawiec, Eds. Springer New York, 2012, vol. 106, pp. 191–206. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-1427-8_12

[MG:8] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "CλaSH: Structural descriptions of synchronous hardware using Haskell," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, 2010, pp. 714–721. [Online]. Available: http://dx.doi.org/10.1109/DSD.2010.21

[MG:9] J. Kuper, C. Baaij, M. Kooijman, and M. Gerards, "Exercises in architecture specification using CλaSH," in *Specification Design Languages (FDL 2010), 2010 Forum on*, 2010, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1049/ic.2010.0149

[MG:10] C. P. R. Baaij, M. Kooijman, J. Kuper, M. E. T. Gerards, and E. Molenkamp, "Tool demonstration: CLasH - from Haskell to hardware," in *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell, Edinburgh, Scotland*. New York: ACM, 2009, pp. 3–3.

[MG:11] M. Gerards, J. Kuper, A. Kokkeler, and B. Molenkamp, "Streaming reduction circuit," in *Proceedings of the 12th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools, Patras, Greece*, 2009, pp. 287–292. [Online]. Available: http://dx.doi.org/10.1109/DSD.2009.141

Index

Symbols

3-partition problem, 34

A

ACPI, 97
Active cores, 67
Agreeable deadlines, 8, 13, 32, 35, 39
Algorithmic power management, 7
Amount of parallelism, 68
Amount of work, 16
Approximation ratio, 31, 80, 134
Arrival time, 12
Average power, 3
Average power function, 23
Average speed, 3, 4

B

BCD algorithm, 32
Begin time, 12
Break-even time, 17, 97, 100
Buffer occupancy levels, 35

C

Common deadline, 36
Completion time, 12
Concave, 17
Concave function, 133
Constant speed, 20
Continuous speed, 16
Continuous speed scaling, 16
Convex, 4, 131
Convex flow problem, 25, 34
Convex function, 131
Convex optimisation, 133
Convex optimisation problem, 133
Convex power function, 20
Convex set, 131
Critical interval, 29
Critical speed, 22

D

DAG, 2
Deadline, 12
Density, 29
Dijkstra's shortest path algorithm, 110
Discrete speed, 16
Discrete speed scaling, 16, 22, 31
DPM, 1
DVFS, 1, 14
Dynamic power, 14
Dynamic programming, 32

E

Earliest Deadline First algorithm, 29
Effective speed, 119
Energy graph, 109
Energy overhead, 15
Energy-efficient scheduling, 7
Energy-per-work function, 22, 67
Equivalent uniprocessor problem, 92
Execution time, 12, 16

F

Feasible set, 133
Feasible solutions, 133
Feedback control, 35
Firm real-time, 2
Flash storage, 2
Flow problem, 25
FPTAS, 134
Frame, 37, 99
Frame-based system, 37, 98, 99

G

General tasks, 12, 28, 34
Global speed scaling, 14
Global DVFS, 7
Global speed scaling, 7, 36, 63
Greedy, 56
Greedyslack, 56

H

Hard drives, 2
Hard real-time, 2
Heuristic, 34
Heuristic algorithms, 134

I

Idle interval, 17
Idle-time energy function, 17
Idle-time-energy function, 101
Inefficient speed, 21

J

Jensen's inequality, 20, 132

K

KKT conditions, 133

L

Laminar instances, 13, 33
Latency, 100
Linear program, 22
List scheduling, 134
Local speed scaling, 14
Local DVFS, 7
Local speed scaling, 7, 63

M

Makespan, 12, 66, 73
Migrations of tasks, 34
Multicore processors, 63
Multiprocessor scheduling, 4
Multiprocessor system, 23

N

Network cards, 2
Nominal speed, 3
Nonconvex power function, 21
Nonuniform power, 14
Nonuniform power function, 25
NP-hard, 4, 6, 32, 34, 98

O

Offline optimisation, 40
Offline speed scaling, 43
Online, 35
Online optimisation, 40
Online speed scaling, 35, 50
Optimal schedules, 64
Overhead, 43

P

Parallelism, 65
Perfect predictor, 57
Period, 37
Periodic arrival times, 52
Periodic deadlines, 52
Pieces, 89
Power equality, 24
Power function, 4, 67
PRA-SS, 40, 52
Precedence constraint, 66
Precedence constraints, 2, 12, 36
Prediction window, 53
Predictions, 35
Preemptions, 28, 34
PTAS, 34, 83, 134

R

RA-SS, 40, 50
Real-time, 2
RecursiveSmoothing algorithm, 44
Reduction, 6
Research questions, 7
Robust, 40
Robust deadline, 50

S

Scheduling criterion, 64, 73
Scheduling trade-offs, 6
Server problem, 37
Simplifying assumptions, 60
Simulation, 23
Sleep modes, 5, 16, 97
Soft real-time, 2
Speed, 16, 68
Speed fluctuations, 45
Speed function, 16
Speed scaling, 1, 14
Standard task graph set, 36
Static energy, 43, 45
Static power, 15
Streaming applications, 2
Subset sum problem, 6
Substitution of variables, 25, 31
Switched capacitances, 31

T

Task graph, 64
Tasks, 12
Three field notation, 18
Time overhead, 15
Total work, 66
Transition delay overhead, 15
Transition latency, 5, 17
Tree-structured tasks, 13
Two-processor scheduling problem, 82

U

Uniprocessor system, 39

V

Video decoder, 2

W

Weighted directed acyclic graph, 109
Weighted makespan, 65, 75
Wide task graphs, 37
Work, 16, 66
Worst Case Work (WCW), 41

Y

YDS algorithm, 28