DEPARTAMENTO DEINGENIERÍA TELEMÁTICA Y ELECTRÓNICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

Energy-based Fair Queuing: Energy-centric Processor Scheduling Algorithm for Battery-limited Mobile Systems

TESIS DOCTORAL

Jianguo Wei

Master Universitario en Ingeniería de Sistemas y Servicios para la Sociedad de la Información

DIRECTOR

Eduardo Juárez Martínez

Ph.D. ès Sciences Techniques from the Swiss Federal Institute of Technology in Lausanne

2015

Resumen

Los dispositivos móviles modernos disponen cada vez de más funcionalidad debido al rápido avance de las tecnologías de las comunicaciones y computaciones móviles. Sin embargo, la capacidad de la batería no ha experimentado un aumento equivalente. Por ello, la experiencia de usuario en los sistemas móviles modernos se ve muy afectada por la vida de la batería, que es un factor inestable de difícil de control. Para abordar este problema, investigaciones anteriores han propuesto un esquema de gestion del consumo (PM) centrada en la energía y que proporciona una garantía sobre la vida operativa de la batería mediante la gestión de la energía como un recurso de primera clase en el sistema. Como el planificador juega un papel fundamental en la administración del consumo de energía y en la garantía del rendimiento de las aplicaciones, esta tesis explora la optimización de la experiencia de usuario para sistemas móviles con energía limitada desde la perspectiva de un planificador que tiene en cuenta el consumo de energía en un contexto en el que ésta es un recurso de primera clase.

En esta tesis se analiza en primer lugar los factores que contribuyen de forma general a la experiencia de usuario en un sistema móvil. Después se determinan los requisitos esenciales que afectan a la experiencia de usuario en la planificación centrada en el consumo de energía, que son el reparto proporcional de la potencia, el cumplimiento de las restricciones temporales, y cuando sea necesario, el compromiso entre la cuota de potencia y las restricciones temporales. Para cumplir con los requisitos, el algoritmo clásico de fair queueing y su modelo de referencia se extienden desde los dominios de las comunicaciones y ancho de banda de CPU hacia el dominio de la energía, y en base a ésto, se propone el algoritmo energy-based fair queueing (EFQ) para proporcionar una planificación basada en la energía. El algoritmo EFQ está diseñado para compartir la potencia consumida entre las tareas mediante su planificación en función de la energía consumida y de la cuota reservada. La cuota de consumo de cada tarea con restricciones temporales está protegida frente a diversos cambios que puedan ocurrir en el sistema. Además, para dar mejor soporte

a las tareas en tiempo real y multimedia, se propone un mecanismo para combinar con el algoritmo EFQ para dar preferencia en la planificación durante breves intervalos de tiempo a las tareas más urgentes con restricciones temporales.Las propiedades del algoritmo EFQ se evaluan a través del modelado de alto nivel y la simulación. Los resultados de las simulaciones indican que los requisitos esenciales de la planificación centrada en la energía pueden lograrse.

El algoritmo EFQ se implementa más tarde en el kernel de Linux. Para evaluar las propiedades del planificador EFQ basado en Linux, se desarrolló un banco de pruebas experimental basado en una sitema empotrado, un programa de banco de pruebas multihilo, y un conjunto de pruebas de código abierto. A través de experimentos específicamente diseñados, esta tesis verifica primero las propiedades de EFQ en la gestión de la cuota de consumo de potencia y la planificación en tiempo real y, a continuación, explora los beneficios potenciales de emplear la planificación EFQ en la optimización de la experiencia de usuario para sistemas móviles con energía limitada. Los resultados experimentales sobre la gestión de la cuota de energía muestran que EFQ es más eficaz que el planificador de Linux-CFS en la gestión de energía, logrando un reparto proporcional de la energía del sistema independientemente de en qué dispositivo se consume la energía. Los resultados experimentales en la planificación en tiempo real demuestran que EFQ puede lograr de forma eficaz, flexible y robusta el cumplimiento de las restricciones temporales aunque se dé el caso de aumento del el número de tareas o del error en la estimación de energía. Por último, un análisis comparativo de los resultados experimentales sobre la optimización de la experiencia del usuario demuestra que, primero, EFQ es más eficaz y flexible que los algoritmos tradicionales de planificación del procesador, como el que se encuentra por defecto en el planificador de Linux y, segundo, que proporciona la posibilidad de optimizar y preservar la experiencia de usuario para los sistemas móviles con energía limitada.

Abstract

Modern mobile devices have been becoming increasingly powerful in functionality and entertainment as the next-generation mobile computing and communication technologies are rapidly advanced. However, the battery capacity has not experienced an equivalent increase. The user experience of modern mobile systems is therefore greatly affected by the battery lifetime, which is an unstable factor that is hard to control. To address this problem, previous works proposed energy-centric power management (PM) schemes to provide strong guarantee on the battery lifetime by globally managing energy as the first-class resource in the system. As the processor scheduler plays a pivotal role in power management and application performance guarantee, this thesis explores the user experience optimization of energy-limited mobile systems from the perspective of energy-centric processor scheduling in an energy-centric context.

This thesis first analyzes the general contributing factors of the mobile system user experience. Then it determines the essential requirements on the energy-centric processor scheduling for user experience optimization, which are proportional power sharing, time-constraint compliance, and when necessary, a tradeoff between the power share and the time-constraint compliance. To meet the requirements, the classical fair queuing algorithm and its reference model are extended from the network and CPU bandwidth sharing domain to the energy sharing domain, and based on that, the energy-based fair queuing (EFQ) algorithm is proposed for performing energy-centric processor scheduling. The EFQ algorithm is designed to provide proportional power shares to tasks by scheduling the tasks based on their energy consumption and weights. The power share of each time-sensitive task is protected upon the change of the scheduling environment to guarantee a stable performance, and any instantaneous power share that is overly allocated to one time-sensitive task can be fairly re-allocated to the other tasks. In addition, to better support real-time and multimedia scheduling, certain real-time friendly mechanism is combined into the EFQ algorithm to give time-limited scheduling preference to the time-sensitive tasks.

Through high-level modelling and simulation, the properties of the EFQ algorithm are evaluated. The simulation results indicate that the essential requirements of energy-centric processor scheduling can be achieved.

The EFQ algorithm is later implemented in the Linux kernel. To assess the properties of the Linux-based EFQ scheduler, an experimental test-bench based on an embedded platform, a multithreading test-bench program, and an open-source benchmark suite is developed. Through specifically-designed experiments, this thesis first verifies the properties of EFQ in power share management and real-time scheduling, and then, explores the potential benefits of employing EFQ scheduling in the user experience optimization for energy-limited mobile systems. Experimental results on power share management show that EFQ is more effective than the Linux-CFS scheduler in managing power shares and it can achieve a proportional sharing of the system power regardless of on which device the energy is spent. Experimental results on real-time scheduling demonstrate that EFQ can achieve effective, flexible and robust time-constraint compliance upon the increase of energy estimation error and task number. Finally, a comparative analysis of the experimental results on user experience optimization demonstrates that EFQ is more effective and flexible than traditional processor scheduling algorithms, such as those of the default Linux scheduler, in optimizing and preserving the user experience of energy-limited mobile systems.

Acknowledgements

This thesis could not have been completed without the support and effort of many. First of all, I would like to thank all the professors and lab mates in GDEM-CITSEM. My first thank goes to my official supervisor, Eduardo Juárez Martínez, from whom I have learned how to conduct research works with critical thinking. Eduardo is a knowledgeable professor with great patience on guiding students, and he always provides valuable and quick comments to my research work. Thanks to the other professors in GDEM: César Sanz Álvaro, Matías Javier Garrido González, Fernando Pescador del Oso and Pedro José Lobo Perea, for their efforts in providing and creating such a friendly and helpful working environment in the lab. Thanks also to my lab mates, especially Rong Ren, who came to GDEM together with me and we started the Master and PhD at the same time. It has been a pleasure to work together with her during the past four years. Thanks to Juanjo and Enrique, I have benefited a lot through the cooperation and the exchange of idea with them. Thanks to David, Gonzalo, Ernesto, Miguel and Oscar, for their generous help both in the work and in everyday life. Without their help, my life in Madrid would be much tougher. I enjoy talking with them during the lunch and appreciate their efforts to push me speak more Spanish.

I would also like to thank my family and friends. Thanks to my parents, for their hard working to bring me up and ensure my receiving a good-quality of education, for their greatness and unselfishness to let their only child fly as far as he can and freely chase his life. Thanks to my girlfriend, I can't image I can finish this PhD thesis without her caring, encouragement and sacrifice. Thanks to my close friends in Madrid, Rong Ren, Shan Huang, Meijuan Zhang, Liang Chai and Xiang Wang, for the pure friendship maintained between us during the past four years and a half. Remember the first night we spent together in the Beijing international airport, and the uncountable nights we spent together with or without a reason.

Table of Contents

Resumen i

Abstract iii

Acknowledgements v

List of Figures x

List of Tables xiii

1 Introduction 1

1.1 Background and challenge 1

1.2 Motivation 3

1.2.1 Battery lifetime guarantee 3

1.2.2 User experience optimization 7

1.3 Objectives 10

1.4 Methodology and organization 13

1.5 Contributions 14

2 Related Work 17

2.1 OS-level power management 17

2.1.1 Introduction 17

2.1.2 Performance-centric power management 18

2.1.3 Battery lifetime-aware power management 19

2.1.4 Energy-centric power management 20

2.2 GPOS scheduling algorithms 29

2.2.1 Introduction 29

2.2.2 GPOS scheduling requirements 31

2.2.3 Priority scheduling 33

2.2.4 Real-time scheduling 36

2.2.5 Proportional share scheduling 40

2.3 Summary and discussion 58

3 Energy-Centric Processor Scheduling 62

3.1 Assumptions and Conditions 62

3.1.1 Applications, threads, and tasks 64

3.1.2 Energy accounting 65

3.1.3 Energy allocation 67

3.1.4 Whole view 67

3.2 Energy-centric scheduling model 69

3.3 Power share management 72

3.3.1 Maximum long-term and worst-case power shares 72

3.3.2 Power share protection 77

3.3.3 Power share reallocation 81

3.4 Energy-based fair queuing (EFQ) 85

3.4.1 Challenges of developing energy-based fair queuing 85

3.4.2 Starting-energy fair queueing (SEFQ) 87

3.4.3 Time-constraint compliance under EFQ 89

3.4.4 Borrowed starting-energy fair queuing (BSEFQ) 92

3.5 Summary 96

4 High-Level Modelling and Simulation 99

4.1 SystemC 99

4.2 EFQ modelling in SystemC 100

4.2.1 High-level abstraction 100

4.2.2 EFQ modelling: the consumer 101

4.3 Simulation test-bench design 106

4.3.1 Test-bench architecture 106

4.3.2 Task modelling: the producer 107

4.3.3 Obtaining scheduling results 109

4.4 Task characterizations for simulation 110

4.5 Simulation results 112

4.5.1 Maintaining proportional power sharing 112

4.5.2 Time-constraint compliance 115

4.5.3 Trading off power share and time-constraint compliance 117

4.6 Summary 119

5 Linux-based Implementation 121

5.1 EFQ implementation in the Linux kernel 121
5.1.1 Extend the scheduling-related data structures....123
5.1.2 Deal with priority and kernel load weight 127
5.1.3 EFQ implementation within the core scheduling functions....130
5.1.4 Implement the share protection and reallocation 138
5.1.5 Extend the system call interface 140
5.2 Simulation-based debugging 141
5.2.1 The Linux scheduler simulator 142
5.2.2 Extend the LinSched API for EFQ simulation and debugging.144
5.2.3 An EFQ simulation scenario 147
5.3 Summary 149

6 Experimental Test-bench 151

6.1 Test-bench and methodology overview 151
6.1.1 Test-bench architecture 151
6.1.2 Experimental methodology 152
6.2 Computing platform 154
6.2.1 The hardware environment 154
6.2.2 The software environment 156
6.3 Power supply and measurement system 157
6.4 Benchmark characterization 160
6.5 Multithreading test-bench program 164
6.6 Summary 166

7 Experimental Results 168

7.1 Maintaining proportional power sharing 168
7.1.1 Proportional sharing of the system-wide power 168
7.1.2 Power share protection 170
7.2 Time-constraint compliance 172
7.2.1 Time-constraint compliance under variable workloads 172
7.2.2 Robustness of time-constraint compliance 177
7.3 User experience optimization under energy limit 182

7.3.1 Experimental assumptions and task characterizations 182
7.3.2 Experimental results analysis and discussion 184
7.4 Summary 192

8 Conclusions 194
8.1 Summary and discussion 194
8.2 Limitations and future work 197
8.3 Final words 198

Bibliography 199

List of Figures

Figure 1. 1: General Structure of Energy-centric Power Management 6

Figure 1. 2: Block Diagram of the Methodology and Thesis Organization 13

Figure 3. 1: Energy-centric CPU Scheduling Surroundings 63

Figure 3. 2: Example of System-wide Power in Reference to CPU Execution Time 66

Figure 3. 3: Illustration of the relationship between the system power and CPU time 70

Figure 4. 1: High-level Abstraction of the CPU Scheduling 100

Figure 4. 2: Flow Chart of SystemC-based EFQ Modelling in the Consumer Module 103

Figure 4. 3: Architecture of the SystemC-based Simulation Test-bench 106

Figure 4. 4: Flow Chart of the SystemC-based Task Modelling in the Producer Module 108

Figure 4. 5: Proportional Power Sharing under SEFQ 113

Figure 4. 6: Trading off Power Share and Time-constraint Compliance with EFQ 118

Figure 5. 1: Abstract Description of the Linux-based EFQ Implementation 122

Figure 5. 2: Hierarchy of the Scheduling-related Data Structures in Linux 124

Figure 5. 3: Extension of the Main Data Structures 125

Figure 5. 4: The Linux Kernel Priority Scale 127

Figure 5. 5: Linux Macros for Priority Conversion 127

Figure 5. 6: The Default Niceness Table of Linux-CFS 128

Figure 5. 7: The Modified Niceness Table for Linux-EFQ 129

Figure 5. 8: The Extended Priority Scale for Linux-EFQ 130

Figure 5. 9: Modified Linux Macros for Priority Conversion 130

Figure 5. 10: Code Flow Diagram for the Periodic Scheduler Function *scheduler_tick* 131

Figure 5. 11: Code Flow Diagram for the Main Scheduler Function *schedule* 133

Figure 5. 12: Flow Chart of EFQ Implementation within the Periodic Scheduler *scheduler_tick* 135

Figure 5. 13: Flow Chart of EFQ Implementation upon Task Launch or Wakeup 137

Figure 5. 14: Flowchart of Power Share Management..... 139

Figure 5. 15: List of System Calls for EFQ Scheduling..... 140

Figure 5. 16: Architecture of the LinSched Scheduler Simulator 143

Figure 5. 17: List of Main LinSched Simulation Engine API Functions 144

Figure 5. 18: The New LinSched API Function for Creating EFQ Tasks 145

Figure 5. 19: The LinSched API Function for Generating Energy Loads of Interactive Tasks 146

Figure 5. 20: A LinSched Script for EFQ Simulation based on Linux 148

Figure 6. 1: Architecture Overview of the Experimental Test-bench 151

Figure 6. 2: Overview of the Experimental Methodology 153

Figure 6. 3: Block Diagram of BeagleBoard..... 155

Figure 6. 4: Block Diagram of the Power Supply and Measurement System..... 158

Figure 6. 5: Device Connections of the Power Supply and Measurement System 158

Figure 6. 6: GUI of the Battery Emulator and Simulator..... 160

Figure 6. 7: Structure Diagram of the Multithreading Test-bench Program..... 165

Figure 7. 1: Comparison of the System-Wide Power Share under EFQ and Linux-CFS..... 169

Figure 7. 2: Power Share Protection under EFQ..... 170

Figure 7. 3 : Real-time Performances upon Different Levels of Energy Estimation Error..... 179

Figure 7. 4: Real-time Performances upon Different Number of Background Batch Tasks 181

Figure 7. 5: Power Management and Optimization within One Epoch 185

Figure 7. 6: Desired Power Consumptions when all Tasks are Behaving Normally 188

Figure 7. 7: Protecting Task Power upon Abnormal Behaviors from Benchmark *rt_fft* 189

List of Tables

Table 3. 1: Comparison of the Proportional Share Scheduling Models for Network, CPU and Energy....71

Table 3. 2: Example of Computing the Maximum Long-term and Worst-case Power Shares....75

Table 3. 3: Example of Recalculating the Effective Weight for Power Share Protection....79

Table 3. 4: Example of Power Share Reallocation upon the Temporary Releasing of RC Power Shares....83

Table 4. 1: Characterization of Tasks in the Simulation....111

Table 4. 2: Comparison in Performance of Time-sensitive Tasks....115

Table 5. 1: Characterization of Tasks for A Linux-based Simulation of EFQ....148

Table 6. 1: Power Profiles of the Basic Components of Benchmarks....162

Table 6. 2: Characterization of Benchmarks with Constant Workload....163

Table 6. 3: Characterization of Periodic Benchmarks with Variable Workload....163

Table 7. 1: Benchmark Characterizations for Power Share Protection Experiment....170

Table 7. 2: Benchmark Characterizations with Variable Workloads....173

Table 7. 3: Time-constraint Compliance under Variable Workloads....175

Table 7. 4: Benchmark Characterizations for Robustness Validation of Time-constraint Compliance....178

Table 7. 5: Benchmark Characterizations for Demonstration on User Experience Optimization....182

Table 7. 6: Characterization of User Preference and Energy Allocation....184

Table 7. 7: Epoch-based User Experience Optimization under EFQ....185

Table 7. 8: Comparison of System Performance under BSEFQ and Linux Scheduler when the Benchmark rt_fft is Abnormally-behaved....188

Chapter 1

Introduction

1.1 Background and challenge

Currently, general purpose operating system (GPOS)-based mobile devices such as Smartphones and tablets are experiencing significant improvements in hardware performance and functionality, making them capable of handling multiple applications simultaneously to meet the diversity of user needs. In a typical usage scenario of GPOS-based mobile devices, a user can simultaneously browse webpages, listen to music while having the push notifications turned on to stay informed of the latest status from the email box and social network accounts. The increased complexity and functionality in many GPOS-based mobile devices has motivated a transformation of the system usefulness assessment from a quality of service (QoS) approach to a quality of experience (QoE) approach [1, 2]. Instead of assessing the system usefulness purely based on the computational application performance, the user experience is now referred as the assessment basis.

In the meantime, along with the growth in computational frequency, data transmission bandwidth and software complexity, the energy demands of modern mobile applications are increasing higher as well, making the battery discharging rate faster than ever. Unfortunately, the battery capacity is not experiencing a corresponding augment; instead, it is further restricted by the relentless trend to make mobile devices lighter, smaller, and thinner. Because the usefulness of mobile systems not only depends on the computational speed and application functionality, but also is limited by the battery lifetime, the battery energy in modern mobile devices has become a scarce resource that is as important as machine resources like the CPU, memory, and network bandwidth [3, 4, 5].

Against the above background, system designers have to conquer the problem of how to optimize the mobile system user experience under the battery energy limit.

This problem has been becoming increasingly prominent on modern mobile systems due to the strained relationship between the system performance and the battery energy. However, user experience optimization under the battery energy limit is a difficult engineering problem because the user experience of a system depends on a variety of possibly incompatible factors, such as the application performance, multitasking ability, system fluency and response time, as well as the battery lifetime. For instance, simultaneously executing multiple applications can increase the difficulty of ensuring the performance of each application, while improving the overall system performance from the application performance and/or the multitasking ability will lead to a higher battery draining rate and a shorter battery lifetime.

When a user is running multiple applications on a battery-limited mobile device, usually he or she has different preferences on the applications, and there is a requirement concerning how long the battery needs to last for the most-preferred applications. While the ability to guarantee a target battery lifetime increases the confidence and the sense of security of the user on using the mobile system, in many cases, a lack of such guarantee may turn the user to be a “coward” who is cautious on every operation that can cause additional energy expenditures. Because the battery lifetime affects the availability of the mobile system and service, a failure to achieve the expected battery lifetime can significantly degrade the quality of experience (QoE) of the system and make it unacceptable to the user. It is thus obvious that, under the battery energy limit, to achieve a user-expected battery lifetime is the most fundamental requirement of a mobile system user and is one of the most important factors in the assessment of the overall user experience [2, 6]. Especially, in the emergence of energy scarcity, to provide a guarantee on the battery lifetime is likely to be the dominant requirement on a mobile system. Just to list a couple of scenarios in which a specific battery lifetime is considered as a more important factor than the pure application performance: a presenter is adjusting the slides on his way to the conference while having the music turned on for inspiration and relaxation, the battery lifetime is more concerned than the music quality and screen brightness; or, a sports fan is watching a match broadcast while participating a forum discussion with

friends, maintaining the mobile device until the end of the match is more desirable than a good video quality and a quick forum update rate.

Unfortunately, the battery lifetime of many modern mobile systems, especially Smartphones, is notoriously fragile. Even if the mobile system is not specifically assigned certain resource-intensive tasks, the battery might be quickly depleted without the user even noticing. Besides, guarantee a target battery lifetime is a tricky work from the user side if the operating system is not properly designed. Users have no idea if launching a new application or introducing some extra operations to the mobile system at certain moments will finally cause a failure to reach the expected battery lifetime. Therefore, in comparison with the other contributing factors of the mobile system user experience, such as the application performance and the multitasking ability, the battery lifetime is a more unstable factor that is hard to control.

1.2 Motivation

For battery-limited mobile systems, because the battery lifetime is the most fundamental, though unstable, contributing factor of the user experience, it is believed that the first step of user experience optimization under energy limit is to provide a battery lifetime guarantee to the system. More specifically, in a battery-limited mobile system that is simultaneously running multiple applications, the basis of user experience optimization is to guarantee a target battery lifetime to the user-preferable applications. Once the requirement on the battery lifetime is satisfied, the user experience of the mobile system can be further optimized from the other aspects, such as enhance the application performance and enable the simultaneous support of more applications. However, in all cases, the additional optimizations should be done without impairing the target battery lifetime.

1.2.1 Battery lifetime guarantee

Guarantee the battery lifetime of energy-limited mobile devices is a complex work

that requires a comprehensive power and energy management from the hardware level to the operating system (OS) level and the application level. Especially, the operating system should play a pivotal role because on one side, it is aware of the battery discharging state and the power states of hardware devices, and on the other side, it can learn the requirements of the user and the applications through the user/OS and application/OS interfaces.

Until now, there have been a large number of OS-level power management (PM) schemes proposed to deal with the energy issue. Unfortunately, the majority of them are not strong energy-aware enough to provide a battery lifetime guarantee. These PM schemes are considered as best-effort and performance-centric in energy saving in the sense that they only make the best effort to reduce the energy consumption and extend the battery lifetime under the application performance constraints but fail to provide a guarantee on the battery lifetime. Among these performance-centric PM schemes, dynamic power management (DPM) [7, 8, 9] and dynamic voltage and frequency scaling (DVFS) [10, 11, 12, 13] are the most well-known and widely-applied schemes. With the performance-centric strategy, hardware resources such as the CPU and memory are managed as the first-class resources to preferentially guarantee the pure application performance, while the energy management and power optimization are at best placed on a secondary position.

To provide a specific guarantee on the battery lifetime, the energy as a resource should be raised to a position that is at least equivalently important as the hardware resources [5]. Based on this belief, a number of battery lifetime-aware PM schemes [5, 14, 15] have been proposed. Basically, in order to achieve a specific battery lifetime, these PM schemes frequently check the residual energy in the battery and use that information to guide the adaptation of the energy demands of the applications. However, because the functioning of these PM schemes relies on the cooperation and the self-adaption of the applications, the range of achievable battery lifetime is significantly restricted by the application degradation levels. Besides, it is impossible to widely apply these PM schemes in a general system where the applications are not necessarily adaptive.

In summary, the above-mentioned PM schemes are either unable to provide battery lifetime guarantee with their performance-centric strategy or overly restricted by the self-adaptive applications with limited degradation levels. To further extend a guaranteed battery lifetime, the user has to manually turn off some unimportant applications or less-used functions and peripherals (e.g. Wi-Fi, GPS). This leaves the painful trading-off between the overall system performance and the target battery lifetime to the user itself. However, without the information and control from the OS side, a user can easily fall into a dilemma: on one side, launch a new application or enhance the quality of existing applications may undermine the target battery lifetime; while on the other side, keep or degrade the original system setting may lose the chance to optimize the user experience within the target battery lifetime.

To guarantee a user-specified battery lifetime for general mobile systems with no specific requirement on the applications, it has been proposed that energy, rather than hardware machine resources, should be managed as the first-class resource in the system [16, 17, 18]. Correspondingly, power optimization and energy management should be considered prior to the optimization of the application performance and diversity. Power management schemes that explicitly manage the energy as the first-class system resource are known as energy-centric PM schemes [18].

As shown in Figure 1.1, the functioning of an energy-centric PM scheme relies on the cooperation of three modules: the energy allocation module, the energy-centric scheduling module and, the energy accounting module [4, 19].

The energy allocation module is in charge of restricting the battery discharging rate as well as allocating the battery energy among applications; while a proper restriction of the battery discharging rate sets the average system power that is required to achieve a target battery lifetime, a proper energy distribution among different applications guarantees to the user-preferred application(s) an energy allocation that is consistent with its actual energy demand. Therefore, the energy allocation module is pivotal in battery lifetime guarantee and energy quanta guarantee for user-preferred applications; it builds the basis of user experience optimization for energy-limited mobile systems.

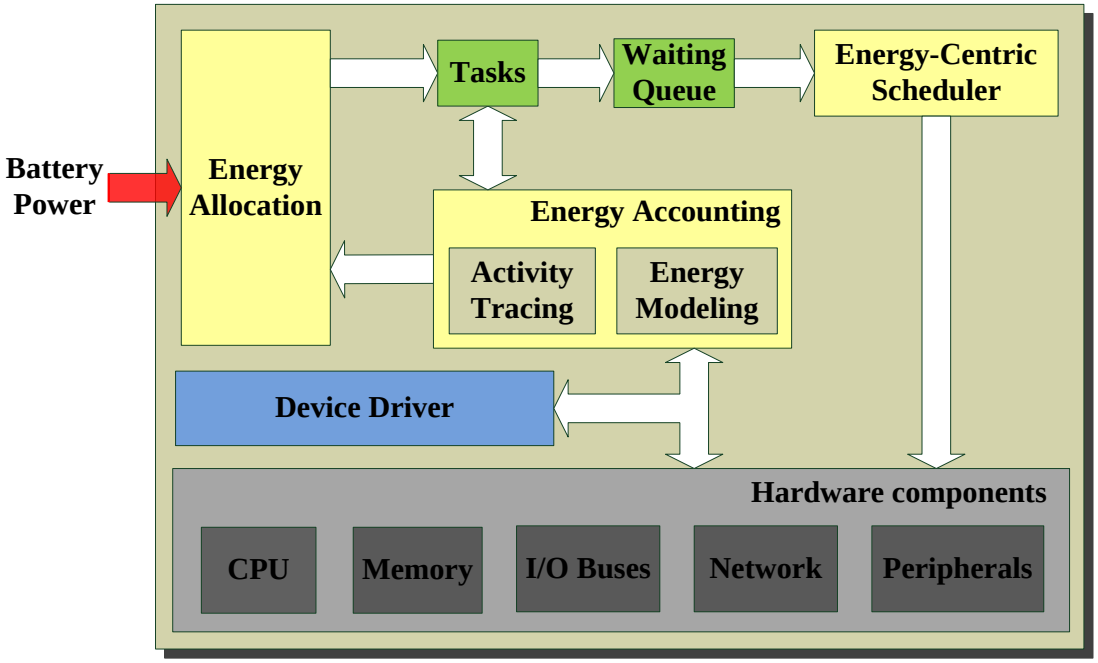


Figure 1. 1: General Structure of Energy-centric Power Management [4, 19]

The energy-centric scheduling module is in charge of scheduling the battery energy to the applications. While applications can be allocated shares of energy that are consistent with their energy demands or proportional to the user-specified ratio with the energy allocation module, the ability that applications can consume energy proportionally depends upon the schedulers (e.g. CPU, network and disk) that control access to the machine resources [20]. The energy allocation module works interactively with the energy-centric scheduling module to maximize the battery energy utilization; if one application is not able to consume its energy quota, either because it does not require that much energy or because it does not obtain enough dispatching opportunities, the energy allocation module may reduce the energy quota of the application so that other applications can gain a larger one, and vice versa. Therefore, with proper or user-specified allocation of energy that has already been made in the energy allocation module, to ensure that each application has adequate energy spending opportunity, the energy-centric scheduling module should ensure a proportional sharing of the system power in accordance with the energy allocation ratio [4]. Otherwise, some applications may fail to consume their allocated energy quota and the actual energy sharing among applications will shift away from that

originally desired due to the energy quota re-adjusting from the energy allocation module. Besides, with a proper energy quota reserved for a time-sensitive application, the scheduler needs to work intelligently to let the task consume its share of energy within the time constraints. Otherwise, the application performance cannot be guaranteed. Therefore, the energy-centric scheduling module should be carefully designed to achieve a proportional and timely sharing of the battery power; it determines the additional improvement on the user experience that can be achieved on top of a guaranteed battery lifetime.

Finally, the energy accounting module is in charge of mapping the hardware device energy consumptions to the applications. To achieve that, the energy consumption on hardware devices should be accurately modelled and correctly accounted to the corresponding application that causes the device activities. The energy accounting module is a fundamental and bridging module of the energy allocation and energy-centric scheduling because, without a properly functioning energy accounting module, there will be no correct information on how much energy has been consumed by each application [4].

An in-depth survey and discussion on the three essential modules of energy-centric power management is provided in Chapter 2.

1.2.2 User experience optimization

Considering the strong guarantee on battery lifetime that can be provided by energy-centric power management schemes, the work of this dissertation focuses on the optimization of mobile system user experience from the energy-centric scheduling module in an energy-centric context. Based on a guaranteed target battery lifetime, the user experience of the mobile system can be further optimized by satisfying other system and user requirements. Basically, a better user experience is achieved when the performance of those user-preferred applications is always guaranteed during the battery lifetime. Generally, the user experience optimization in energy-centric systems can be considered from three aspects: the battery energy utilization, the application performance and, the multitasking ability.

The battery energy utilization is measured by the residual energy in the battery when the target battery lifetime is reached. In energy-centric systems, the residual battery energy reflects to what extent the total available energy is utilized to provide services within the battery lifetime. Therefore, it should be minimized to optimize the user experience of the mobile system. On the contrary, high residual battery energy indicates overly conservative energy management and lost opportunities on performance enhancement [20]. To maximize the battery energy utilization, the energy allocation module should dynamically adjust the energy quota of applications in accordance with their energy demands, and the energy-centric scheduling module should let the applications proportionally share the system power in accordance with the energy allocation ratio. Chapter 2 provides an introduction on the mechanisms utilized to maximize the battery energy utilization in the energy allocation module.

The application performance is measured in different ways depending on the type of the applications. Generally, three types of applications are considered in this dissertation: batch, interactive and real-time (both soft and hard); the latter two types are collectively referred as time-sensitive applications. Batch applications are computationally intensive; therefore, the performance is determined by the average energy serving rate (or the average power) that increases linearly to the allocated energy quota. Instead, time-sensitive applications have periodic energy demands; their performance not only depends on the allocated energy quota but also is affected by the time (or how timely) each application is allowed to consume its energy quota. Specifically, the performance of interactive applications is measured by the response time; an interactive application can be allocated an energy quota that is enough to finish the requested work in each period, but how long it takes to finish the work may vary depending upon when the interactive application is allowed to consume energy. Similarly, for soft and hard real-time applications whose performance is measured by the deadline miss ratio, even with the energy quota that is enough to meet all the energy demands in the long term, the performance can still be variable depending on if the application is scheduled the demanded amount of energy before each deadline. Therefore, for time-sensitive applications, apart from scheduling the energy

proportionally, the energy-centric scheduling module should also schedule energy with time-awareness.

Besides, more possibilities of user experience optimization can be explored by distinguishing the pure application performance and the user-perceived performance of time-sensitive applications. Although the user-perceived performance is dominated by the pure application performance, and in many cases, increasing the pure application performance also increases the user-perceived performance, they are not strictly linear to each other. For interactive applications, since the human can only distinguish the difference in delay above a certain level, let us say 100ms, then, reducing the response time to a level that is lower than 100ms will no longer improve the user-perceived performance. Meanwhile, improving the user-perceived performance does not necessarily require a decrease of the response time. For example, the response time with almost constant length can give users smoother experience and possibly higher satisfaction than the response time with big variability, even if the latter has a smaller average value than the former. As far as real-time applications are concerned, depending on whether the missed deadlines are soft ones or hard ones, the impact on user-perceived performance is totally different. For soft real-time applications, missing a few numbers of deadlines may slightly degrade the user-perceived performance, or can be hardly perceived by the user. Take the multimedia applications for example, if a video frame cannot be decoded before a deadline, it can cheat on the user by sending the former decoded frame to the screen, in many cases, the user just fails to tell the differences. Even if the deadlines are continuously missed, a multimedia application can always reduce its picture size or quality to meet the following deadlines. However, for hard real-time applications or tasks, miss one deadline may cause serious problems, such as system freeze, which will significantly degrade the user's satisfaction on the system. Therefore, to further optimize the user experience in energy-centric systems, the energy-centric scheduling should make use of the gap between the user-perceived performance and the pure application performance, and work intelligently to ensure a smooth and timely scheduling of the energy to the time-sensitive applications.

Finally, the multitasking ability is measured by the maximum number of applications that can be simultaneously supported in the system to meet the diverse user requirements. Ideally, the optimal user experience is achieved when the battery energy utilization, the application performance and the multitasking ability are all optimized. However, under the battery energy limit and the processor frequency limit, increase the application performance and enhance the multitasking ability are two conflicting operations when the energy and processor bandwidth are scarce. To optimize the user experience, the application performance and the multitasking ability should be traded off based on the user preferences. For instance, in order to support the launch of new applications in an energy-limited system, the user may prefer to reduce the energy quota of the currently running applications and degrade their performance to the minimum acceptable levels. In another more specific example, the user may maintain the performance of the most-preferable application(s) and degrade the performance of the least-preferable ones to allow the launch of new applications.

1.3 Objectives

While some potential methods of user experience optimization are overly dependent on the specific user preferences which are still not available when energy-centric power management schemes are designed, the work of this dissertation is dedicated to exploring one general approach on user experience optimization from the perspective of energy-centric scheduling. Considering that the CPU scheduler is the core component of the energy-centric scheduling module that interacts with the energy allocation module and affects the battery energy utilization, and, in addition, the CPU scheduler plays a pivotal role in balancing the application performance and the multitasking ability, this thesis work focuses on energy-centric processor (or CPU) scheduling to investigate the user experience optimization on battery-limited mobile systems.

To optimize the user experience of an energy-centric system, the design of the energy-centric processor scheduling algorithm should at least consider the following three general requirements.

Firstly, as the core component of the energy-centric scheduling module, the energy-centric CPU scheduler should be designed to achieve a proportional sharing of the system power among applications by taking into account the system-wide energy consumption on different hardware devices. This is to say that, under an energy-centric CPU scheduler, the energy consumption on other hardware devices can affect the allocation of the CPU time quanta to the applications and thus, balance the system-wide energy and power sharing among tasks in a user-desired ratio. This is the pivotal feature that distinguishes an energy-centric CPU scheduler to a traditional one. Therefore, the first requirement on the energy-centric processor scheduling is the ability to support a proportional sharing of the system power among applications.

Secondly, the energy-centric CPU scheduler should first provide performance guarantees for time-sensitive applications, and in addition to that, optimize the multitasking ability without impairing the guaranteed performances. When the energy and processor bandwidth are scarce, the optimizations of application performance and multitasking ability are in conflict. However, within the CPU frequency limit, the CPU scheduling algorithm can be properly designed to optimize the application number while preserving the performance of those already admitted time-sensitive applications. This requirement has a threefold meaning: first, the share of energy and CPU that is reserved for each time-sensitive application should be protected from the competition of new-joining applications; second, the resource share reservation for each time-sensitive application should not be overly made so that a larger number of time-sensitive applications can be supported; third, the maximum resource share in total that can be reserved for time-sensitive applications should be optimized. Since this is a real-time scheduling problem, the second requirement on the energy-centric processor scheduling is marked as the ability to provide support on time-constraint compliance.

Thirdly, the energy-centric CPU scheduler should be able to balance between the application power share and the time-constraint compliance when is necessary. For time-sensitive applications that have a highly-fluctuating workload over the periods, pursuing strict time-constraint compliance requires a significant over-reservation of

the system power share, while the allowance of missing a few time constraints may greatly reduce the required power share. Especially, if any user-preferred application becomes ill-behaved by demanding excessive energy in short terms, potential high power pulses should be restricted to maintain the performance of other user-preferred applications. Because of the above concerns, the third requirement on energy-centric CPU scheduler is the ability to provide a flexible trade-off between the application power share and the time-constraint compliance.

Based on the requirements discussed above, we set the objectives of this work as proposing and developing energy-centric processor scheduling algorithms that can provide supports in proportional power sharing, time-constraint compliance, and, in addition, a flexible trade-off between them. On top of the algorithm proposal and development, this work also aims to explore and demonstrate the potential of employing energy-centric processer scheduling in the user experience optimization for energy-limited mobile systems. For clarity, the above objectives are decomposed into the following four objectives:

- OBJECTIVE 1: Theoretical proposal of energy-centric processor scheduling algorithms

- OBJECTIVE 2: Implementation of energy-centric processor scheduling in general purpose operating systems (GPOS). In this work, the target GPOS is the Linux operating system considering its open-source property

- OBJECTIVE 3: Analysis and discussion of the experimental results based on the Linux implementation of energy-centric processor scheduling

- OBJECTIVE 4: Experimental demonstration and exploration of user experience optimization based on the Linux implementation of energy-centric processor scheduling

1.4 Methodology and organization

To achieve the above specified objectives, we follow a methodology that is demonstrated in Figure 1.2. Through the overview of the methodology, Figure 1.2 also shows the organization of this dissertation.

Chapter 2 surveys the related work on OS-level power management and general purpose operating system (GPOS) scheduling. At the end of Chapter 2, follows a discussion on applying traditional GPOS scheduling algorithms to the energy-centric processor scheduling for the target of achieving proportional system power sharing. Then, based on the related work investigation, and a set of assumptions and conditions that are made on the applications, energy accounting and energy allocation, Chapter 3 presents the theoretical proposal of one energy-centric processor scheduling algorithm called energy-based fair queuing (EFQ).

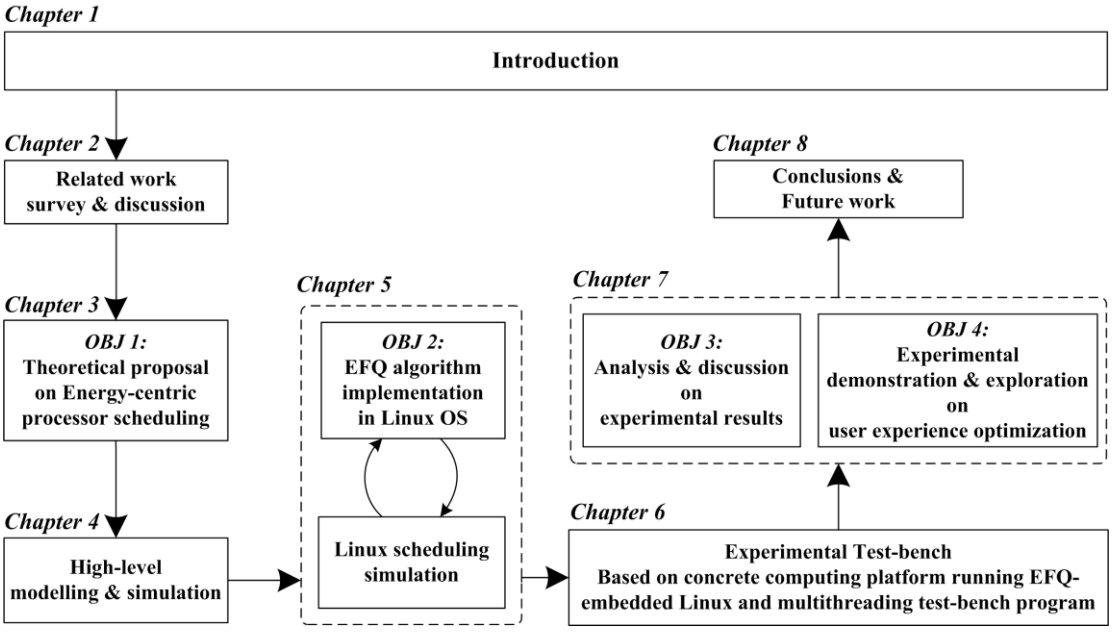


Figure 1. 2: Block Diagram of the Methodology and Thesis Organization

Before moving to the Linux implementation of the EFQ algorithm, in Chapter 4, a high-level modelling and simulation of the algorithm is taken to verify its scheduling behaviors in response to specifically-defined task sets and energy requesting patterns. The high-level simulation allows a convenient and flexible

pre-evaluation of the EFQ properties in reference to the requirements of energy-centric processor scheduling. After that, Chapter 5 presents the implementation of the EFQ algorithm in the Linux kernel. Considering the complexity and difficulty of the Linux implementation work, a Linux scheduling simulation tool is employed for user-space debugging.

To design experiments and assess the EFQ implementation in Linux, an experimental test-bench based on a concrete Linux computing platform and a multithreading test-bench program is built up in Chapter 6. To focus on the assessment of the energy-centric processor scheduling and avoid the complexity of building up the whole energy-centric system, the modules of energy allocation and energy accounting are not specifically implemented. Instead, we first profile the power consumption of each benchmarks and then program the threads of the multithreading test-bench program with the benchmarks and their power profiles. Then, in Chapter 7, based on the experimental test-bench, experiments assuming various EFQ scheduling scenarios are designed and the results are analyzed and discussed. We first analyze the EFQ scheduling results on proportional power sharing and time-constraint compliance, after that, explore the user experience optimization of energy-limited mobile systems through specifically-designed experiments on the EFQ scheduling and the default Linux scheduling. Finally, Chapter 8 concludes the thesis work and suggests the directions for future research.

1.5 Contributions

This thesis work investigates the user experience optimization of mobile systems under the energy limit. The starting point of the dissertation is that, the battery lifetime is one of the most important while unstable contributing factors of the mobile system user experience, and therefore, energy-centric power management schemes should be available to provide strong guarantee on the battery lifetime. Based on this point, this work explores the design of energy-centric processor scheduling for the user experience optimization of battery-limited mobile systems. To be specific, the contributions of this dissertation include:

1. Energy-centric scheduling model. The classical reference model for proportional resource sharing is extended from the traditional CPU and network scheduling domain to the energy scheduling domain, and based on that, a practical energy-centric scheduling model is proposed for proportional power share scheduling.

2. Power share management mechanisms. This work provides an insight into the power share management under the energy-centric proportional power share scheduling model, and proposes efficient mechanisms on power share protection and power share reallocation.

3. Starting-energy fair queuing (SEFQ) scheduling algorithm. This work provides a comparative and analytical study of the challenges in developing energy-based fair queuing (EFQ) algorithms for proportional power sharing, then proposes the starting-energy fair queuing (SEFQ) algorithm that is of low time complexity, low implementation complexity, and near-optimal fairness.

4. Borrowed starting-energy fair queuing (BSEFQ) scheduling algorithm. The thesis provides an analytical and exploratory study of the time-constraint compliance under energy-based fair queuing (EFQ) scheduling, and proposes the borrowed starting-energy fair queuing (BSEFQ) algorithm that adds real-time scheduling support on top of proportional power sharing

5. High-level modelling and simulation of the EFQ algorithms. This thesis provides a methodology to model and simulate the scheduling algorithms from a high level perspective. The high-level abstraction and simulation allow a convenient and flexible pre-evaluation of the scheduling behavior with specifically-designed task set and workloads.

6. Linux-based implementation of EFQ scheduling. The proposed EFQ scheduling algorithm is implemented in the Linux kernel. A debugging method based on the Linux scheduler simulation is developed for user space simulation and debugging of the EFQ scheduler.

7. A thorough evaluation of the EFQ implementation in Linux. Experiments based on a concrete computing platform and a multithreading test-bench program is designed to evaluate the Linux-based EFQ implementation, from the properties of proportional system power sharing and power share protection to those of time-constraint compliance and its robustness under energy estimation errors and task number variations.

8. Experimental and analytical exploration of the user experience optimization for energy-limited mobile systems. Through a comparative analysis of the experimental results under the EFQ scheduler and the default Linux scheduler, this work explores the potential of employing EFQ algorithm in optimizing the mobile system user experience under the energy limit.

Chapter 2

Related Work

Developing energy-centric processor scheduling algorithms requires an investigation of the related works on both power management and GPOS scheduling. In this chapter, different OS-level power management mechanisms are firstly surveyed, with a focus on the energy-centric power management schemes; and then, GPOS scheduling algorithms that range from priority scheduling to real-time scheduling and proportional share scheduling are investigated and comparatively discussed. At the end of this chapter, the related works on power management and GPOS scheduling are summarized and the possibilities of applying traditional GPOS scheduling algorithms on energy-centric processor scheduling is discussed.

2.1 OS-level power management

2.1.1 Introduction

Energy and power management is a system-wide issue that requires a support from the bottom hardware level up to the operating system (OS) level and the application level. Among the different levels, the OS level is pivotal in energy saving and management because it enables the interaction between the application level and the hardware level. While hardware devices can be designed to work on multiple power modes, and applications can be programed to be energy-aware and self-adaptive in energy requesting [14, 15], only with the support of the operating system can these features be exploited to manage the energy consumption. For example, one self-adaptive application can only be informed the low battery state through the OS, and if the application reduces its energy demands, again, only through the OS can the application inform the device drivers to change device-access patterns or device power modes to decrease the hardware energy consumption.

With the battery energy limit imposed on mobile systems, OS-level power management (PM) schemes need to balance between the overall application performance and the battery lifetime. While the guarantee of application performance relies on a proper management of the machine resources, such as the CPU, memory, and network bandwidth, the guarantee of battery lifetime requires a proper management of the energy as a system-wide resource. Depending on the priorities given to the resources, OS-level PM schemes are generally separated into three categories: performance-centric, battery lifetime-aware, and energy-centric. In the remaining of this section, the three types of power management schemes will be separately introduced, with a focus on the essential modules of energy-centric power management.

2.1.2 Performance-centric power management

In performance-centric PM schemes, the management of the machine resources is given a higher priority than the energy management and power optimization. Therefore, PM schemes that are performance-centric usually do the best effort to reduce the system energy consumption under the application performance constraints.

Most performance-centric PM schemes generally fall into two categories: dynamic power management (DPM) [7-9] that runs the workload to completion at the maximum CPU and disk speed and then rests the system in the longest low-power mode; and dynamic voltage and frequency scaling (DVFS) [10-13] that assumes the highest energy saving can be achieved by running at the lowest hardware setting under the performance constraints. One apparent limitation of those performance-centric PM schemes is that they are unable to provide a battery lifetime guarantee to the mobile systems. Under performance-centric PM schemes, all applications are first allowed to freely consume energy as they demand to guarantee the desired application performance, energy saving is considered only after the performance goal is achieved.

Performance-centric PM schemes are considered as low energy-aware, because they are unaware of the power-related system states and are not able to adjust the

application energy requesting based on the battery discharging state. In a performance-centric PM scheme, all applications are allowed to run freely without being imposed any power or energy restriction. Therefore, even energy saving is made with the best-effort, the battery is likely to be depleted before the most preferable applications finish their assignments.

2.1.3 Battery lifetime-aware power management

Battery lifetime-aware PM schemes are those that are aware of the battery discharging state and are able to adapt the application performance according to the user-specified battery lifetime [5, 14, 15]. In this case, the energy management is given at least equal priority in reference to the machine resource management.

Under battery lifetime-aware PM schemes, it has been demonstrated that a target battery lifetime can be achieved if the applications can self-adapt their performance-related activities (so as the energy demands) based on the residual energy in the battery. Specifically, Flinn *et al* [14, 15] first developed on the Odyssey platform [21] a battery lifetime-aware PM scheme. The Odyssey achieves the battery lifetime by periodically measuring the residual battery energy, predicting future energy demand based on present and past power usages, and notifying applications with an up-call if adaptation of energy demands is needed. A similar PM scheme was later developed by Neugebauer *et al* [5] on the Nemesis OS [22]. Nemesis also requires applications to be energy-aware and cooperative, but introduces an economic model to provide feedback to the applications.

Although PM schemes based on application adaptation provide a possibility to achieve a target battery lifetime, the space to set a user-desired lifetime for preferable applications is limited by the degradation level of the applications. For instance, let us assume the expected lifetime is 2 hours, but the maximum lifetime achievable when all applications are degraded to the lowest fidelity is 90 minutes or even less, then the system fails to provide a meaningful guarantee on the battery lifetime. Moreover, the requirement of applications to be adaptive and energy-aware impedes these PM schemes from being widely applied in general systems. Considering the millions of

applications that have been existed in the market, rewriting all of them is a huge work, not to mention that not all of the applications are able to be programmed as adaptive.

2.1.4 Energy-centric power management

In energy-centric PM schemes, the energy is explicitly and globally managed as the first-class resource in the system. Unlike machine resources which are exclusive to one hardware device, energy is global to all devices and can affect the use of each machine resource in the system [20]. The energy consumption in a system goes in three dimensions: time, application and device [4]. While the power optimization in most PM schemes is explored only from one or two dimensions, managing the energy globally as the first-class resource allows the energy to be controllable in three dimensions. This extends the design space of developing strong energy-aware PM schemes, therefore, brings more opportunities to the power optimization and the achievement of advanced energy goals.

The advantage of designing a system to be energy-centric is obvious. First, an energy-centric framework is compatible with the existing power optimization efforts such as DPM, DVFS and application self-adaptation, which can be combined into the energy-centric framework to form more advanced PM schemes. Then, it does not require the applications to be adaptive and energy-aware, non-energy-aware applications are also combined into the framework and their roles in the system-wide energy management can be further exploited. Finally, guaranteeing a user-specified battery lifetime for critical applications becomes a straightforward work that can be achieved as long as the energy is properly distributed both over time and among the applications.

A complete energy-centric PM scheme is composed of three modules: the energy allocation module, the energy accounting module and the energy scheduling module. To the best of the author's knowledge, strictly, ECOSystem [4, 18, 20] and Cinder [19] are the only two reported energy-centric systems that explicitly and globally manage energy as the first-class resource; and particularly, ECOSystem is the only one that has implemented all the three modules of an energy-centric PM scheme. In the

following subsections, the related works on the three essential modules of energy-centric power management will be separately reviewed and discussed.

2.1.4.1 Energy allocation

The energy allocation module does two things, on one side, it restricts the battery discharging rate by allocating energy to the whole system in a speed that is determined by the target battery lifetime and the remaining battery energy, and on the other side, it allocates energy to the applications to support their execution during the expected lifetime. Specifically, when the energy available in the battery is adequate to support the normal execution of all applications during a target battery lifetime, each application is allocated a share of energy that is consistent with its actual energy demand; otherwise, the energy consumption of those least-preferred applications are properly restricted so that enough remaining energy can be reserved to the most-preferred applications to run them with user-acceptable performance until the expected lifetime is reached. If a user-preferred application supports self-adaption with multiple fidelities, based on the user preference, it is possible that the application degrades its quality of service and correspondingly reduces its energy demand so that to allocate the saved energy to the other active applications or to enable the launch of a new application under the target lifetime constraint.

The energy allocations of different applications should be protected from each other, so that no application can use the energy quota of other applications. Furthermore, an amount of energy consumed on various devices should be subtracted from the energy quota of the application that causes the device activities. Since energy is a system-wide resource that is commonly shared by different devices, and an amount of energy spent on one device is no longer available for other devices, resource abstraction techniques such as the resource container [23] are employed to achieve a uniform and explicit management of the energy resource in the system. Different approaches have been proposed to implement the energy allocation module. In the ECOSystem [4, 18, 20], an energy abstraction called *currentcy* is defined to represent the amount of energy that is available for the system within a fixed amount

of time. The target lifetime is achieved by dividing it into a number of fixed-length time intervals named *epochs* and limiting the *currentcy* available in each *epoch*; the limited amount of *currentcy* is released at the beginning of each *epoch* and is further allocated to different applications according to energy demands (with adequate *currentcy*) or user-specified proportions (with scarce *currentcy*). Instead, in the Cinder OS [19], the abstractions of *reserve* and *tap* are proposed to manage the energy on the basis of the applications: while *reserves* store an amount of energy for a target application or its child processes, *taps* control the maximum rate at which the energy is consumed by placing a rate limit on the energy flowing between the battery and the reserves so that a fixed amount of energy is transferred per unit time.

The workloads of some applications tend to be variable over time and the actual energy demand may drop to a level that is lower than the energy allocation; consequently, certain resource containers or reserves may gradually accumulate energy that no other application can use. To avoid this situation and maximize the battery energy utilization (minimize the residual battery energy) within the battery lifetime, both ECOSytem and Cinder have implemented certain energy conserving mechanisms to reclaim the unused energy and redistribute it to other needy tasks. The idea of energy reclaiming and redistributing is similar in ECOSystem and Cinder. Basically, the capacity of each resource container (or reserve) is bounded to a certain limit; once the accumulated energy of a resource container reaches to its capacity limit, it will stop the energy infusion and remaining energy allocation will be redistributed to other resource containers whose capacity has not yet reached to the limit. Particularly, the energy conserving mechanism in ECOSystem achieves a considerably low residual energy (less than 1%) by dynamically adjusting the capacity of each task's resource container based on the actual energy needs [20].

2.1.4.2 Energy accounting

The energy accounting module concerns how the energy consumed on each hardware device can be correctly mapped to the application that causes the device activities.

Energy accounting in modern mobile systems is a challenging work because many device activities are asynchronous to the processes and applications may interact with each other through Inter-process Communication (IPC) calls [19]. It is very likely that energy is simultaneously consumed on multiple devices whose activities are caused by various applications. In this case, it is difficult to distinguish the energy consumption of different applications. A potential power management scheme may require the energy accounting to be on-line and real-time to support the high-level policies, such as energy-aware scheduling. This presents additional challenges to the energy accounting module.

The implementation of an energy accounting module requires two basic elements: an energy modeling mechanism that meters or estimates the amount of energy consumed on each device, and an activity tracking mechanism that identifies device activities from applications and attributes the device's energy consumption to applications. There have been a number of efforts to deal with the energy accounting [24-37]. Many of them only focus on energy modeling [25-31], because it is the basis of energy profiling and management. And in many scenarios, only one or a few synchronous and independent applications are considered.

The PowerScope [24] is one of the earliest works that maps energy consumption to applications. To collect measurement data of the system when it is executing applications, the PowerScope requires two synchronized components: a system monitor that samples the program counter (PC) and process identifier (PID) of the currently executing process, and a digital multi-meter that samples the power consumption of the whole system. Then, the collected data is used to profile the energy consumption of applications based on an off-line energy analyzer. The idea of PowerScope is clear and the implementation is easy, however, it is difficult to combine this work into PM schemes because it relies on an extra digital multi-meter to measure the energy consumption and the energy accounting is done off-line. Besides, since PowerScope only measures the whole system power instead of the power of individual devices, it would fail to accurately attribute the energy consumption to the correct applications. A practical energy accounting module

requires not only an accurate and on-line activities tracking mechanism but also a fast, cost-effective, and accurate on-line power model for individual devices.

Methods based on hardware performance monitor counters (PMCs) have been shown as a good solution to model the energy consumption of devices that support PMCs. Since Bellosa, in [25] correlated PMCs to energy consumption, PMCs combined with linear regression methods have been widely used to support on-line energy estimation that is free of extra hardware power meters [26-31]. To achieve an on-line energy model that can estimate the energy consumption with acceptable accuracy and low overhead, a minimum set of PMCs that are highly correlated to the energy consumption have to be properly selected, and their correlation coefficients with the energy consumption have to be determined in advance through an off-line calibration phase for obtaining the linear energy model. The set of PMCs are usually platform-dependent and may vary in a different platform.

Early works of PMCs only focus on the energy modeling of processors, these methods are later applied to model the memory energy consumption [29, 30], and the frequencies of CPU, memory and I/O buses can be combined with PMCs to form an energy model under voltage and frequency scaling [30]. However, for those devices (e.g. I/O peripherals, disk, network interface card and display) that cannot be directly monitored by PMCs, their power consumption has to be modelled based on some device-specific state variables (e.g. idle, active, low-power, transmit) and parameters (e.g. device I/O rate, network download and upload data rate, and display brightness level) [31].

While PMC-based energy modeling is believed as a promising mean for on-the-fly power estimations, the availability and uniqueness of appropriate PMC-events for each platform prevents it from being widely employed as a general energy modeling method for diverse platforms. For this reason, some systems, especially most mobile systems, model the power consumption of all system devices based on device-specific state variables and parameters [32-37]. When the system is running applications, the power consumption of different devices can be estimated based on the readable state variables and device-specific parameters. Similar to the

PCM-based energy modeling methods, to obtain the device power model, it also requires a one-time, off-line calibration phase to stress the devices under test with variable workloads. The purpose is to expose all of the power states and correlate the power consumption to the state variables and device-specific parameters [35].

Depending on the accuracy of the measurement tools used (e.g. power meter, battery interface or voltage sensor) and the set of exercising applications employed in the calibration phase, different levels of error will be incurred to the device power model. The generated device power model usually consists of two parts: the base power of a power state that is related to a state variable and the active power that is related to device-specific parameters. For example, the CPU power is determined by whether it is idle or active, and also by the CPU-specific parameters such as CPU utilization and frequency; and the network interface power is determined by the device states such as doze, receive and transmit, as well as the transmitting or receiving bit rate.

Power state-based methods tend to have worse accuracy than the PMC-based methods due to the lack of deep knowledge on processor and memory activities; however, they should incur lower overheads because only high-level metrics (e.g. power state, CPU frequency) are employed during the on-line energy estimation. This property makes high-rate energy estimation to be possible for a system. Specifically, an energy estimation rate as high as 100Hz (or one per 10ms) has been reported in [36, 37]. High-rate energy estimation is particularly appreciated by energy accounting and energy-aware scheduling in multitasking systems. For simplicity, existing energy-centric systems like ECOSystem [4, 18, 20] and Cinder [19] and several recently-developed energy accounting mechanisms [36, 37] all employ power state-based methods to estimate the device energy consumption.

Activity tracking mechanisms in energy accounting are usually device-specific. For hardware devices that can only be part of one synchronous activity at a time, such as CPU, tracking the activity is as easy as logging the activity events (e.g. CPU scheduling events). For devices that work in an asynchronous manner and can execute a common operation on behalf of multiple tasks, such as disk, network interface and

radio, energy accounting can be very complex and challenging. Take the ECOSystem [18] for example, activity tracking in disk requires inserting the resource container ID (or task ID) into the system calls and sharing the energy cost of spinning up and down among the related tasks; and activity tracking in network interface requires adding the container ID into the socket structure and the TCP/IP implementation.

Activity tracking mechanisms are also OS-specific. Tracking activities related to GPS and network interface in Window Mobile 6 needs rerouting the system calls and modifying their implementation mechanisms; while tracking activities in Android requires logging system calls and events in the Linux kernel level, application framework level as well as Dalvik VM level [36]. In Quanto [37] where energy tracking should be achieved across networked embedded systems (or nodes), the authors defined an activity label that contains the origin activity node to represent each activity and propagate the label along with the causally related operations to track energy consumption. In Linux, Inter-process communication (IPC) calls can significantly complex the activity tracking and energy accounting, because the IPC is implemented mainly based upon messages and the calling processes is difficult to identify. However, this is not a big issue to the Cinder OS [19] because its IPC mechanisms allow a simple and accurate tracking of resource use across IPC calls.

2.1.4.3 Energy-centric scheduling

The energy scheduling module controls each application's energy consumption rate, and thus should ensure that each application has adequate energy spending opportunity corresponding to the allocated energy quota. If it does not work properly, some applications would spend energy in a slower speed than others and may fail to run out of their energy quotas (due to the maximum power limit) before the next energy infusion arrives. In extreme situations, it may end up that certain applications temporarily monopolize the machine resources in the beginning of an epoch (or infusion period) and quickly go to idle after draining their resource containers, but others are initially starved for an unknown time interval before getting the energy

spending opportunity. Thanks to the energy conserving mechanisms implemented in ECOSystem [4, 18, 20] and Cinder [19], the unspent energy can be reclaimed and redistributed to other applications to maximize the energy utilization. However, in the above case, the energy is no longer consumed in proportion to the originally-desired ratio of energy allocation.

To achieve a desired energy distribution among applications, the scheduling policies should be designed to allow the system power being shared in proportion to the allocated energy quota. A proportional power sharing among applications guarantees a minimum power for each application and thus, avoids the appearance of energy starvation on any application. However, developing a proportional power-share scheduling module is a complex work, because the energy consumption of applications should be considered in a system-wide manner when making scheduling decisions.

While a proper energy abstraction mechanism (e.g. resource container) and an accurate energy accounting mechanism can help the energy scheduling module be globally aware of each application's energy use, how the different schedulers (e.g. CPU, disk, network) should work cooperatively to achieve the system-wide proportional energy-sharing goal is a challenging problem. The CPU scheduler is the core scheduler that leads the energy scheduling for all applications; therefore, it should make scheduling decisions in reference to the energy quota and system-wide energy consumption of each application. That is to say, the energy expenditures on both CPU and non-CPU resources should be considered in each CPU scheduling decision. For other device schedulers, such as the disk scheduler and network scheduler, it is not necessary (and way too complicated) to consider the global energy consumption in each scheduling decision. But when the device bandwidth appears as a bottleneck, the energy quota of applications must be referred in determining or adjusting the bandwidth shares of local resources. This should avoid throttling certain application's execution on CPU when there is data flow bottleneck from disk I/O or network interface [20].

In Cinder OS [19], the authors implemented the energy allocation and energy

accounting modules but left the energy scheduling as future work. In ECOSystem [4, 18, 20], the authors implemented an energy-centric CPU scheduler based on Linux, and explored how it can achieve the desired proportional energy use upon bottlenecked network bandwidth. Their results showed that proportional sharing of the system power is achievable through energy-centric scheduling. To the best of the author's knowledge, the ECOSystem is the only work that concerns energy-centric processor scheduling up to the moment this thesis is written.

However, the work on energy-centric scheduling in ECOSystem is limited by the general-purpose workload that is employed in the application characterizations [4]. Since the application workload is not well-defined and lacks specific information such as deadlines, how the energy-centric scheduling will affect the application performance and user experience is an unanswered question. Besides, ECOSystem only provides a simple description on how the energy-centric scheduler behaves, neither the scheduling algorithm is formally structured nor the concrete implementation scheme is published. Therefore, the specific properties and scheduling overhead of the algorithm are unknown. The work of this dissertation investigates energy-centric CPU scheduling by employing specifically-defined workloads, and explores how energy-centric scheduling can be utilized to optimize the user experience of mobile systems with energy limit.

2.2 GPOS scheduling algorithms

2.2.1 Introduction

Scheduling is concerned with the optional allocation of system resources (e.g. CPU time, network bandwidth, and memory) to threads, processes or data flows in a multitasking environment. It plays a pivotal role in fairly sharing resources and providing performance guarantees. Resource sharing among tasks always results in contention, and scheduling disciplines are algorithms that are used to resolve the contention. In operating systems, scheduling algorithms are mainly applied to the CPU, network interface and storage devices. The scheduler is a special operating system module that implements one or several scheduling disciplines to share the local resources among different entities. A process scheduler selects from among the threads and processes in memory that are ready to execute, and allocates the CPU to one of them. While a network scheduler (or packet scheduler) handles the network packet traffic for fairly sharing the transmission bandwidth among packet flows that are associated with different CPU processes. In the storage side, I/O requests initiated by different processes are managed by the I/O scheduler (sometimes also called disk scheduler), it determines in which order the requested I/O operations will be submitted to the storage devices. Because one process can access to different hardware resources, the schedulers must work cooperatively to achieve a system-wide resource sharing among the processes.

The design of a scheduler may relate to different concerns such as the throughput, latency, CPU utilization, fairness and time-constraint meeting. In practice, there is no “one commonly true scheduling algorithm” that fulfills all these goals [38], which are usually contradictory (e.g. throughput versus latency, fairness versus time-constraint meeting). Preference is given to one or more of the concerns depending upon the specific purpose of the system. A batch processing system executes a series of programs that run to completion without manual intervention, thus maximizing the CPU utilization is more concerned than minimizing the latency. On the contrary, interactive systems should optimize the scheduling latency to avoid undermining the

user experience on system smoothness. In a hard real-time system, such as an automatic control system in transportation (e.g. high-speed train), the scheduler must ensure the crucial processes to meet the hard deadlines, which is pivotal to keep the system stable and avoid disastrous consequences. While in soft real-time systems, such as multimedia systems, processes are allowed to miss a few of deadlines on condition that the user-sensed quality of service is not significantly undermined.

The earliest computer systems are mainly used to process batch tasks for large scale computation, and the computational resources were very scarce and expensive at that time. For this reason, early scheduling algorithms are designed to maximize the CPU utilization with low scheduling overhead. The First-Come First-Serve (FCFS) is one of the oldest and most widely used batch scheduling algorithms in the process and I/O scheduling. In a FCFS scheduler, the ready-to-run tasks are stored in a First-In First-Out (FIFO) queue and tasks are scheduled in the order of their arriving time. A FCFS scheduler is simple to implement because only one FIFO queue is needed for all tasks. It is also of low-overhead because context switches solely occur when a task execution is terminated. Unfortunately, FCFS can lead to resource starvation if one task arrives first but never completes. For the same reason, the waiting time and response time of short interactive tasks can be high if the CPU is continuously occupied by long batch tasks.

Modern general-purpose operating systems (GPOSs) include personal computer operating systems used in desktop PCs or laptops, as well as mobile operating systems used in Smartphones, tablets, PDAs, or other digital handheld devices. Modern GPOSs often have a mix of batch, interactive and real-time processes. In a typical mobile phone operating system, there may be batch processes running in the background while interactive or multimedia processes running in the foreground; besides, certain system processes that run in the background may have hard deadlines to meet in order to handle emergency calls or keep the operating system stable. In addition to that, a user may have different preferences on the foreground applications. Therefore, a suitable compromise of the above mentioned criteria should be made in the design of a GPOS scheduler.

In the remaining of this section about GPOS scheduling, the design requirements of a GPOS scheduler will be firstly discussed from a general view, and then, three important GPOS scheduling disciplines will be separately surveyed.

2.2.2 GPOS scheduling requirements

In general purpose operating systems, basically the scheduler ensures that no process is starved of accessing to the system resources and, additionally, it provides fairness in the resource sharing among processes. In other words, each process should be allocated a certain fraction of the system resources depending upon the user preference as well as the actual resource demand of different processes. For this reason, general purpose operating systems are essentially time-sharing so that each task or user gets a chance on the CPU at regular intervals. To achieve that, early operating systems require processes to voluntarily cede the CPU and give control to other processes. This approach relies on the processes cooperating for time sharing to work; it is known today as cooperative scheduling (or cooperative multitasking). In a cooperatively-multitasked system, one poorly designed application can continuously occupy the CPU to starve other applications or even pivotal system processes; thus, this method is rarely adopted in modern general-purpose operating systems that are dealing with an increasing number of malicious third-party applications.

Modern GPOSs widely employ the preemptive scheduling (or preemptive multitasking) to ensure a more reliable sharing of the CPU time among processes. Preemptive scheduling takes advantage of clock interrupts or event interrupts (e.g. I/O return, operating system call) to forcibly suspend the currently executing process on the CPU and make a context switch to the next process that is selected according to the scheduling discipline. One of the simplest and most commonly used preemptive scheduling algorithms is the Round-Robin. Round-Robin relies upon a programmable interval timer to generate interrupts periodically; this determines the length of the time quantum (or time slot) after which a scheduling decision will at least have to be made. Processes are managed in a run queue and served with the equally-sized time quanta

in a circular order. A process may voluntarily quit the CPU if its CPU burst is smaller than the assigned time quantum, or otherwise be preempted and added to the back end of the ready queue once the time quantum is expired. Although the Round-Robin scheduling is starvation-free and easy to implement, the resource sharing among processes may be unfair. Because the length of CPU bursts may be smaller than the scheduling time quantum, processes with longer CPU bursts can receive more CPU allocation than processes with shorter CPU bursts. This limitation can be resolved by fair queuing that is to be further discussed in section 2.2.5.

Moreover, the process scheduler must also provide low scheduling latency to interactive processes and time-constraint compliance to real-time processes. Many operating systems employ the process priority to distinguish the importance or urgency of processes. A time-sensitive process (interactive or real-time process) is often given a higher priority over background batch processes to enable earlier dispatch from the run queue. This, however, may bring energy starvation risks. Because a high priority task is always scheduled ahead of a low priority task, the low priority task will be starved if the high-priority task never or rarely blocks. This problem is aggravated in modern GPOSs that usually run high workload real-time applications (e.g. Multimedia, Gaming and VoIP). Besides, a high priority task may also be starved by a medium priority task in the case of priority inversion. Potential solutions such as the aging technique may be used to give starving tasks the chance of execution by gradually increasing their priorities [38]. How the processor scheduler should be designed to support low-latency and real-time scheduling in GPOSs will be further discussed in section 2.2.3 and section 2.2.4.

Apart from fairness and latency, a practical scheduler should also not incur too much processing overhead. The scheduling overhead mainly comes from the computational complexity of the scheduler and the context switch between processes. The ideal computing complexity of a scheduler is *O(1)*, indicating that the scheduling overhead will not scale with the number of active tasks. In contrast, the overhead incurred by an *O(N)* scheduler can scale linearly to the number of active tasks. As far as the context switch overhead is concerned, it depends on how frequently a

scheduling decision must be made, or in other words, the length of the minimum-schedulable time quantum. In most modern systems, the time quantum is generally between 10 and 100 milliseconds, with respect to a relatively small context switch overhead that is on the order of 10 microseconds. Popular GPOSs, such as Windows, Linux and Mac OS, usually employ a hierarchical or multi-level design of the scheduler to support the multitasking of a variety of applications under tolerate overheads [38].

2.2.3 Priority scheduling

Modern operating systems widely employ numerical priorities for process scheduling. Different levels of priority are assigned to processes and the priority scheduler simply allocates the CPU to the process with the highest priority. Processes of equal priority are scheduled in FCFS or Round-Robin. A priority scheduler can be either preemptive or cooperative. In the preemptive version, the currently running process is preempted when a higher-priority process comes in; while in the cooperative version, the higher-priority process is added at the top of the queue to wait for the termination of the currently running process.

Priorities can be assigned according to OS internal criteria, such as CPU burst time and execution urgency, or external factors, such as process importance and user preference. One of the simplest and most intuitive methods is to assign the priority based on the process importance that is subject to user preference. An importance-based scheduling is quite straightforward in providing resource guarantees to the applications that are most important to the user. However, giving autocracy to the user on an operating system that he or she is unfamiliar with can lead to several significant problems. Since the process scheduling is completely based on user preference, there is no fairness among processes and thus, resource starvation is very likely. Also, the ignorance of system internal factors such as time limit and CPU burst time may yield poor performance to the time-sensitive processes. To improve the poor response time of interactive processes, the priority can also be assigned based on the

CPU burst time. Especially, the Shortest Job First (SJF) scheduling firstly selects the task with the shortest length of CPU burst in the run queue, and the Shortest Remaining Time First (SRTF) scheduling is a preemptive version of SJF that achieves the optimal average response time by selecting the task with the shortest remaining time to run next. Unfortunately, SJF and SRTF are only practical in systems where the length of future CPU bursts is predictable; while in real systems, the processing time of many applications, typically multimedia, is hard to predict, without even mentioning the additional prediction overheads. Moreover, in a system that is stressed by many tasks with short CPU bursts, resource starvation is very likely to happen on tasks with long CPU bursts due to their fixed lower priorities.

In general purpose operating systems where processes can be easily categorized into different groups based on their importance to the system and requirements on the scheduling latency, different levels of priority are statically assigned to separate queues for meeting different scheduling needs. Processes from a ready queue with higher priority will be scheduled definitely ahead of those from a lower-priority ready queue, and each queue can have its own scheduling algorithm. This is known as fixed-priority multilevel queue scheduling (MLQ). For example, a high priority can be assigned to a foreground queue that runs interactive processes in Round-Robin; and a low priority is associated with a background queue that schedules batch processes in FCFS. Therefore, interactive processes with a high response time requirement will be fairly scheduled ahead of batch processes with a low response time requirement.

The use of static priority in multilevel queue scheduling can lead to the resource starvation of low-priority processes if each process permanently remains in a queue. The multilevel feedback queue (MLFQ) allows processes to migrate between queues and applies priority aging to avoid starvation. Low priority processes that wait a long time will be upgraded to a higher-priority queue to increase its chance of CPU assignment and eventually complete its execution. In the meanwhile, processes with long CPU bursts may be demoted to a lower-priority queue to avoid overly use of the CPU. In this way, MLFQ favors interactive and I/O-bound processes over batch and

CPU-bound processes. Multilevel feedback queue scheduling is the most flexible and general scheduling algorithm because it can be tuned for any specific system with the dynamic priorities. It is widely employed in many popular operating systems, such as the Solaris, Windows and Mac OS X, to achieve time-sharing scheduling among interactive and batch processes [38]. However, MLFQ is also the most complex to implement, since the design of a MLFQ scheduler concerns many adjustable parameters, including the number of queues, the scheduling algorithm per queue, the methods to upgrade or demote processes and the queue selection rules. The Linux also employs dynamic priorities to favor interactive and I/O-bound processes in the time-sharing scheduling, but in a manner with less overheads. In Linux, the priority of a time-sharing process is upgraded or demoted by a certain value (normally 5) depending on whether it is I/O-bound or CPU-bound.

Real-time and multimedia applications require low waiting and response times to meet concrete temporal constraints. Since a high priority provides low scheduling latency and increases the chance of meeting deadlines with the best effort, it is intuitive to assign higher priority to real-time processes. To add support of soft real-time scheduling in a general purpose time-sharing system, modern operating systems usually separate priorities into real-time class and time-sharing class, and then assign static and strictly higher priority to processes that belong to the real-time class. Take the Windows NT-based operating system for example, higher priority levels that range from 16 to 31 are reserved for real-time processes in a static manner; while lower priority levels from 1 to 15 are assigned to interactive and batch processes, whose priorities are variable under the multilevel feedback queue scheduling [38]. The Linux is another example, in which higher priority levels that range from 0 to 99 are statically reserved for real-time processes, while lower priority levels from 100 to 139 are assigned to interactive and batch processes depending on their interactivity [39]. This is beneficial for a time-sensitive process to meet deadlines because it can immediately obtain resources with a high priority. On the other side, interactive and batch processes may risk to starvation because they have to wait as long as there are real-time processes awake in the system. The system may be

blocked for normal use in the presence of heavy-loaded or runaway real-time activities [40]. The Marc OS X also assigns the highest priority range to real-time processes; in particular, a real-time process will be demoted to the priority range of time-sharing processes if it overly requests CPU access in a compute-bound manner [41]. This may help to avoid starvation in the above specific situation, and in most cases, it is enough for a general purpose operating system in which few processes belong to the real-time class. However, demoting a real-time process is a tricky work, especially in modern systems where real-time applications such as multimedia applications are increasingly heavy-loaded and compute-bound. On one side, a normally-behaved but relatively compute-bound real-time process may be mistakenly demoted to the time-sharing priority range and time constraints are unnecessarily missed. On the other side, a malicious real-time process may continuously fork short processes to keep remained in the real-time priority range and starve time-sharing processes.

Traditionally, general purpose operating systems are mainly extended to support multimedia and soft real-time processes. They are not ready to support hard and strict real-time scheduling because the real-time priorities are statically assigned based on programmer- or user-expressed importance and preferences. To achieve strict time-constraint compliance, process properties such as the cycle duration and deadlines should be taken into consideration in the priority assignments. Recently, the Linux community has been working on the incorporation of hard real-time scheduling into the mainline Linux kernel [42]. Algorithms that target to achieve strict time-constraint compliance will be further discussed in the next section.

2.2.4 Real-time scheduling

Real-time scheduling aims to guarantee that all real-time processes must be served before their temporal constraints. To achieve this objective, it is mandatory to have a preemptive and priority-based scheduling algorithm so that the system can respond immediately to the requirements of a real-time process. The priorities in a real-time

scheduling algorithm can be either static or dynamic. A real-time scheduler working with static priority is simple to implement because the priority of each process is unchanged after the initial assignment; while a dynamic-priority scheduler is more complex because the priorities are adjusted according to the approaching deadlines. In both cases, the system load has to be controlled below a desired threshold in order to keep the processes schedulable. This is achieved by using a technique known as admission control, with which a new process will be admitted only if meeting its resource requirements will not violate the time-constraint compliance of other processes [43].

The rate-monotonic (RM) is one of the most used algorithms for real-time scheduling. Processes are assigned fixed priorities that are inversely proportional to their periods, so that a process with shorter period receives a higher priority. RM is easy to implement on top of generic operating systems and commercial kernels that are usually based on static priorities, and it doesn't require the system to support explicit temporal constraints [44]. Unfortunately, RM cannot fully utilize the CPU bandwidth for real-time scheduling because its CPU utilization is bounded. According to the schedulability test of Liu and Layland [45], the schedulable bound of RM tends to 0.69 when the number of process approaches infinity. It means that, with a large number of processes, the CPU utilization is better below 0.69 so that each process can meet deadlines under RM. A schedulable bound of 1 is achievable under the earliest-deadline-first (EDF) scheduling [46], which dynamically assigns priorities to the processes based on their deadlines: the highest priority is always assigned to the process that has the earliest deadline. In particular, EDF is theoretically optimal when the system is under-loaded because it is believed that "if any scheduling algorithm can meet all the deadlines then EDF can" [47]. Even though with the above mentioned appealing properties, EDF is not as widely used as RM due to the additional implementation complexity incurred in tracing the dynamic deadlines and mapping them into the priority levels [44]. The managing of dynamic priorities also introduces an additional runtime overhead that is not present in a fixed-priority scheme like RM. However, it is worth noting that the total runtime overhead introduced in EFQ may be

less than RM, because the context switching caused by EFQ is less frequent than RM. Another disadvantage of EDF is the poor predictability during overload conditions. When the system becomes overloaded, under EFQ, the set of tasks that miss their deadlines is largely unpredictable; while with fixed priority, the highest-priority process is at least guaranteed to meet its deadlines.

Under admission control schemes, an overload condition is usually transient and occurs when the resource requirements of admitted processes exceed their expected value [43]. This is known as execution overrun, and it is most likely to occur in applications with highly variable resource requirements, such as multimedia applications. To prevent task interference and achieve predictability during execution overruns, an approach known as resource reservation can be employed to enforce resource isolation among tasks [43, 48]. In fact, resource reservation mechanisms are commonly combined with real-time scheduling and admission control schemes to support multimedia applications in general purpose operating systems [43, 46, 48].

The idea behind resource reservation is to allow each real-time task to reserve a time-independent fraction of resource that is just enough to meet its deadlines. Once a task has reserved a certain share of resources, it is guaranteed to receive exactly that share in isolation, independently of the behavior of other tasks. To achieve this stringent resource protection, each task should be prevented from consuming more CPU cycles than its reservation. Therefore, a CPU usage monitoring mechanism is required to accurately measure the computation time consumed by each task [43]. Any task overruns its reservation budget will be immediately degraded to the background time-sharing level. The reservation rate is not necessarily static; it can be modified in a user process with the support of the CPU usage monitoring module and a rate-adaption interface between the kernel and user processes. The resource reservation can be applied both to RM and EDF, and it relies on the admission control to guarantee these already reserved shares: a new task will be rejected to enter the system if its reservation request exceeds the leftover CPU bandwidth that is available for reservation. Recalling the schedulable bound of EFQ and RM, EDF allows a total CPU reservation up to 100%, while RM has a lower reservation bound that is

dependent on the number of real-time tasks. In general purpose operating systems, however, certain unreserved CPU bandwidth is required to avoid resource starvation on time-sharing tasks.

Although resource reservation achieves predictability during execution overruns, “the overall system performance becomes quite dependent on a proper resource allocation” [49]. To satisfy the time constraints of a real-time application with variable computation times, the CPU bandwidth has to be over-reserved based on maximum requirements. However, in this case, the task execution rate will always fall behind the reserved rate, and the CPU cycles reserved for the real-time application are therefore not fully utilized. In the meantime, the slack times are neither available for reservation of other real-time tasks, nor for sharing by conventional time-sharing tasks. The scheduler becomes non-work-conserving in a way that when a task finishes its execution earlier than the reserved time, the CPU goes to idle even if there are other tasks waiting in the queue to be executed. Although the reservation rate can be adjusted in accordance to the feedback of task workloads, predicting the resource requirements is a difficult job in multimedia applications where the workloads are usually data and hardware dependent. Therefore, a hard resource reservation is better suited for hard real-time applications in which the resource requirements are usually constant or can be known beforehand [50].

To achieve a more efficient resource reservation for multimedia applications, the Constant Bandwidth Server (CBS) [51, 52] can be employed under EDF to reserve a proper and constant bandwidth share for each task. With CBS, a task remains in the real-time priority level when its reserved CPU cycles are exhausted; however, its next approaching deadline is postponed properly to guarantee that the execution rate never exceeds the reserved rate. This is also known as soft resource reservation because only soft deadlines are affordable to be postponed. A CBS-based scheduler can behave in a work-conserving fashion, so that available slack times can be efficiently utilized by other tasks [53]. Unfortunately, this method cannot be applied to static-priority algorithms because it requires the system to provide explicit deadlines. Besides, it can risk to resource starvation in a GPOS due to the fact that an

overrunning task can remain in the real-time priority level. Another disadvantage is that the use of deadline postponement mechanism poses restrictions on the behavior of real-time applications, which instead can choose to abandon the unfinished work of the current period and skip to the next period.

Because resource reservation relies on admission control to avoid system overload, any new resource request that cannot be satisfied by the leftover CPU capacity will be denied in order to not violate the rate guarantees of admitted tasks. This achieves fairly strict share guarantee, but at the cost of losing flexibility, fairness and efficiency [40]. With this scheme, a later arriving but more important application may be denied to be allocated resources. Even if the new application is not that important, it is a common situation that a user might be willing to degrade the performance of admitted applications to accommodate the new application [54]. Admission control policy allows a reservation system to shed the system load based on the reservation rate that is allocated at reservation time, thus the system is totally ignorant of the dynamical changing workloads during the program execution. Since the over-reserved CPU time cannot be shared by other tasks, a reservation system might be working in a lightly loaded state and goes to idle regularly, while rejecting the requests from newly launched applications.

2.2.5 Proportional share scheduling

Proportional share scheduling (PSS) is a type of work-conserving algorithm that aims to achieve a proportional and fair allocation of the system resource to each process or data flow [55]. In a proportional share system, each process is pre-assigned a weight and makes progress at a precise rate that is proportional to the weight. This is beneficial to provide differentiated services to various applications in general purpose operating systems. Because proportional share scheduling provides a natural mean to guarantee the performance of time-sensitive tasks and allows a graceful degradation of the task performance in overload situation, it is particularly well-suited to the problem of supporting multimedia and soft real-time applications in GPOSs [55, 56].

The remaining of this section is organized as follows. Firstly, the reference model and performance metrics of the proportional share scheduling are given as the fundamentals in section 2.2.5.1 and section 2.2.5.2, respectively; then section 2.2.5.3 is dedicated to the discussion on supporting real-time scheduling with PSS algorithms, and section 2.2.5.4 is focused on the mechanisms of protecting the resource shares of time-sensitive tasks. From section 2.2.5.5 to section 2.2.5.7, different types of proportional share scheduling algorithms are comparatively introduced, and finally in section 2.2.5.8, it is introduced several PSS algorithms that are particularly designed to support multimedia scheduling in GPOSs.

2.2.5.1 The reference model

PSS algorithms are implemented to approximate the Generalized Processor Sharing (GPS), an idealized fluid-flow model that assumes the resource can be arbitrarily split into infinitesimal units and simultaneously served to different processes or data flows [57]. The earliest comprehensive studies on referring to the GPS model for PSS algorithm design are carried out in the network scheduling domain [57, 59, 60, 61, 66, 68, 71]. In network scheduling, a proportional sharing scheduler should ensure that multiple packlized traffic flows (or session) can fairly share the linked network bandwidth to transmit data with guaranteed rates. This approach is later applied to the processor scheduling domain to achieve proportional sharing of the CPU bandwidth among competitive processes [55, 56, 63, 64, 69, 74]. In this document, the GPS model and PSS algorithms are introduced mainly in referring to the processor scheduling, however, some concepts and definitions from the network scheduling are also referred to provide a comprehensive knowledge on proportional share scheduling.

In the GPS model, each process is associated with a weight that determines the minimum share of the CPU bandwidth that the task is entitled to use. Let w_i denotes the weight assigned to task i, the bandwidth share $f_i(t)$ of task i at time t is defined as [55]:

$$f_i(t) = \frac{w_i}{\sum_{\forall j \epsilon A(t)} w_j}, \ \forall i \ \epsilon \ A\ (t) \tag{2.1}$$

where $A(t)$ denotes the set of active tasks at time t. For any interval $(\mathrm{t}_1, \mathrm{t}_2]$ during which the set of active tasks does not change, the bandwidth share should remain constant and the CPU time allocated to task i is $f_i(t)(t_2 - t_1)$. More generally, if the task share varies over the time, the allocated CPU time during any interval $(\mathrm{t}_1, \mathrm{t}_2]$ is [55]:

$$W_i(t_1, t_2) = \int_{t_1}^{t_2} f_i(\tau)\, d\tau \tag{2.2}$$

2.2.5.2 Performance metrics

The GPS scheduler can provide perfect fairness in resource allocation because the shared resources are assumed to be infinitely divisible and they are simultaneously accessed by multiple tasks. However, any practical implementation of a PSS algorithm must take into account that the minimum schedulable time quantum is bounded by the hardware restrictions and that the processor can be accessed only by one task at a time. Due to the quantization in the CPU time allocation, it is impossible that a task always receives exactly the same CPU time as it is entitled to in the GPS model. This difference between the CPU time that a task should consume in the ideal GPS model and the CPU time it actually consumes in the real system is called *allocation error* or *service time lag (or simply lag)* [55, 58]. The lag is considered as positive if the task in a real system has received less service time than the service time received in the ideal GPS system; otherwise, it is regarded as a zero lag or negative lag. An elaborately designed PSS algorithm should ensure that the sum of lag of all active tasks is zero at any time.

Because the performance of a PSS algorithm is measured by how close it can approximate the GPS model, it is necessary to design the algorithm with a bounded lag for each task. This is especially important when supporting real-time scheduling in general purpose operating systems, in which a proper share of the resource should be determined for periodical time-sensitive tasks [51, 55]. Besides of the specific

algorithm design, the allocation error bound is also dependent on the length of the standard time quantum, Q, which is defined as the maximum length of time that a task is allowed to continuously seize the processor before the next scheduling decision is made. The length of the standard time quantum, Q, is pre-defined by the scheduler designer or system user to achieve a balance between the allocation error and the context-switching overhead. In the processor scheduling, the optimal bound of allocation error that can be achieved by a PSS algorithm is Q; while, in the packet-based network scheduling, the optimal lag bound is L_{max}, the maximum packet size among all the data flows.

The *unfairness bound* (or *fairness bound*) and *latency* (or *delay bound*) are another two metrics that are commonly used to assess a PSS algorithm design, especially in the packet-based network scheduling [59, 60].

The unfairness bound is defined as the maximum difference between the normalized CPU time received by any two tasks i and j over all intervals of time $(t_1, t_2]$ during which both are continuously active, it is expressed as [59]:

$$Max\left\{\left|\frac{W_i(t_1,t_2)}{w_i} - \frac{W_j(t_1,t_2)}{w_j}\right|\right\} \tag{2.3}$$

The unfairness bound is usually used for theoretical analysis of the fairness of a PSS algorithm. In the idealized GPS model, the unfairness bound is zero because the normalized service time of any two tasks is always the same. Intuitively, PSS algorithms should be designed to achieve an unfairness bound that is as close to zero as possible. However, it has been proven that the optimal unfairness bound of a proportional share scheduling algorithm is $\frac{1}{2}\left(\frac{q_i^{max}}{w_i} + \frac{q_j^{max}}{w_j}\right)$, where q_i^{max} and q_j^{max} denote the maximum lengths of time quanta of task i and j, respectively [61].

Note that the maximum length of time quanta of any task in a PSS algorithm is no more than the length of the standard time quantum Q. Therefore, the optimal unfairness bound can be loosely computed as $\frac{Q}{2}\left(\frac{1}{w_i} + \frac{1}{w_j}\right)$. Depending on the algorithm design, the unfairness bound of a PSS algorithm may be close to or larger

than the optimal unfairness bound, but in whatever case, the bound should be $O(1)$, independent of the number of active tasks [61].

The latency (or *delay bound*) is defined to measure the maximum elapsed time from the instant that a time quantum is requested by a task to the instant that the requested time quantum is completely served to the task [60]. It represents the worst-case delay of a task, which is normally the delay seen by the first requested time quantum of a newly-rejoined or rejoined task. In general purpose operating systems, achieving a low latency is important to provide quick response time to the low throughput tasks with short CPU burst time.

Besides of the above metrics, proportional share scheduling algorithms are also evaluated by the implementation complexity and run-time efficiency [56].

2.2.5.3 Real-time scheduling support

The rationale of supporting real-time execution with proportional share scheduling is to allocate a proper share of the CPU bandwidth that is corresponding to the periodic workload of each task [55]. Ideally, the CPU share should be computed based on the task period and the actual execution time in each period. However, because the actual execution time in most multimedia and soft real-time applications can vary over the time and may be hard to predict, it is preferably to compute the CPU share based on the worst-case execution time so that to avoid the prediction overhead as well as any deadline miss caused by the prediction error.

Besides, determining the concrete share for each task in a real system requires taking the allocation error bound into account [56]. Let us assume the task period is p, the actual execution time in each period is a, and the worst-case execution time is c, to meet all the time constraints in a PSS scheduler with allocation error bound as lag, it requires a CPU share that is at least $(c + lag)/p$. This is an over-reserved share in comparison to the ideal share of a/p in the GPS model. As the total share of all tasks is one ($\sum_{i \in A(t)} f_i = 1$), the over-reservation causes a waste of the CPU time allocation in the sense that the over-reserved CPU time cannot be guaranteed to other tasks [56].

The over-reservation of a CPU share can be qualified as $(c + lag)/a$. Because the actual execution time a and the worst-case execution time c are both dependent on the given task, the only approach to directly reduce the over-reservation is to minimize the allocation error bound. According to Stoica *et al* [55], the optimal allocation error bound that can be achieved by PSS algorithms is equal to the size of the standard time quantum, Q. Therefore, the over-reservation is dependent on the size of the standard time quantum, which is typically 10-30 ms in GPOSs. Smaller time quantum can be employed as the user desires, however, it must be balanced with the scheduling overheads incurred by timer interrupts and context switches.

In despite of the allocation error, the level of over-reservation is acceptable in many real-time tasks whose workload is fluctuating slightly. Even for tasks whose worst-case execution is much larger than the actual execution time, a large over-reservation is not a big problem if the worst-case share is a small one. However, the CPU share for reservation will be badly wasted if the task workload is fluctuating significantly and the worst-case share is a considerable large one, which is very common in modern multimedia applications.

To better support multimedia scheduling, it has been proposed to combine extra real-time friendly mechanisms into the PSS algorithms [50]. With the combination of real-time friendly mechanisms, PSS algorithms can provide stronger guarantees on time-constraint compliance while allowing the CPU shares to be much smaller than the worst-case shares. Although the short-term fairness is worse due to the instantaneous priority given to the time-sensitive tasks, the long-term CPU shares are still maintained based on the weight of each task. Algorithms belong to this type will be further introduced in section 2.2.5.8.

2.2.5.4 Share protection

Once the minimum CPU share required by a time-sensitive task is determined, it is necessary to ensure the task at least that amount of share during the task execution. However, according to equation (2.1), the CPU share f_i of one task varies with the

number and the weight sum of the active tasks. Any new task that joins the competition with a large weight may reduce the CPU share of a time-sensitive task to an arbitrarily low level, thus leading to unstable real-time performance. To maintain a desired share for a target task, its weight must be adjusted in accordance with the dynamic activities in the system. This is known as the weight-assignment problem [62], and the approaches that can resolve the problem is called share protection or share insulation. Share protection aims to protect the resource right of one task by isolating its resource share from the activities of the other tasks. It is also employed in hierarchical scheduling models to protect the resource use of different groups of tasks or users [63, 64].

The concept of share protection firstly appears in the lottery scheduling [63], one of the earliest proportional share processor scheduling algorithms based on the probability theory. In the lottery scheduling, it is claimed that the resource use of different task groups can be isolated by applying a specific resource abstraction barrier on each group. However, the concrete implementation scheme of load insulation is not provided in the lottery scheduling, and therefore its computational complexity is unknown. This topic is later explored by Stoica et al., in [54]. It is proposed to build an equation for each time-sensitive task based on the duality of weights and shares. Upon dynamic activities, the weights of normal tasks are always fixed, while the weights of time-sensitive tasks are recomputed based on their desired shares and the weight sum of all tasks. This approach is straightforward and easy to understand, however, it requires resolving a group of simultaneous equations to obtain the proper weight for each time-sensitive task.

To reduce the computational complicity, Goddard and Tan [62] proposed to re-compute the weights of normal tasks instead of the weights of time-sensitive tasks upon dynamic activities. Specifically, the weight sum of all tasks is fixed to one ($\sum_{i\in A} w_i = 1$), and the weight of each time-sensitive task is fixed to its desired share by $w_i = f_i$ to achieve the share reservation; then the weights of the normal tasks are recomputed based on their initially-assigned weights and the share that is not yet reserved. In this way, the computational complexity of share protection is

significantly reduced and the overhead of weight recalculation is dependent on the number of normal tasks.

In the above methods, time-sensitive tasks are ensured a minimum share that is independent of the system dynamics, but they are not able to compete for the resource share that is released by normal tasks. A recently developed share protection approach [65] provides this support by explicitly separating the weight and share to be initial ones and effective ones. In this approach, time-sensitive tasks are assigned both the initial weight and the initial share, while normal tasks are only assigned the initial weight. The initial share allows the time-sensitive task to reserve the minimum required CPU share and the initial weight enables it to compute for the unreserved or released share. In practice, however, it should appear to the system that the time-sensitive task is competing for the resource with one effective weight. Therefore, this approach needs to compute the effective weight for each task. The effective weight of a normal task is simply equal to its initial weight. However, the computation of the effective weight of time-sensitive tasks is pretty cumbersome due to the dependency on the initial weights and shares of all active tasks. Therefore, this approach is better suited for systems with a small number of time-sensitive tasks.

2.2.5.5 Weighted Round-Robin

The weighted Round-Robin (WRR) [66] is one of the simplest proportional share scheduling algorithms that aims to emulate the Generalized Processor Sharing (GPS) model in a Round-Robin style. Like Round-Robin, it executes processes in a circular order and enforces preemption when the allocated time quantum (or time slot) is expired. However, to provide differentiated quality of service, weighted Round-Robin (WRR) assigns a time quantum that is equal to the weight of each process. A process with a large weight gets a larger time quantum than a process with a small weight. Then the CPU is expected to be proportionally shared among processes by scheduling them with the same frequency but different size of time quanta.

As a Round-Robin algorithm, WRR is simple to implement and schedules

processes with a time complexity of *O(1)*. However, it also inherits the unfairness in resource sharing from Round-Robin upon non-equally sized CPU bursts. In practice, to resolve the unfairness problem and achieve proportional resource sharing, WRR requires an estimation of the mean length of CPU bursts to normalize the weight factors. Unfortunately, efforts to resolve the problem have only been found in network packet scheduling under the name of deficit Round-Robin (DRR) [67]. The method is difficult to be transplanted to the processor scheduling domain due to the unpredictability of CPU bursts. Besides, since a process that misses its slot of time has to wait until the time slot of the next round, the maximum delay incurred in a WRR or DRR server is dependent on the number and weights of the active tasks that can be arbitrary. Therefore, the delay bound can be arbitrarily high if the weights are not appropriately selected [68].

2.2.5.6 Lottery scheduling

Lottery scheduling is a probabilistic scheduling algorithm that statistically guarantees proportional resource sharing among processes [63]. Each process is given some number of "lottery tickets", and the scheduler selects the next process by generating a random ticket number. A process with more tickets can have a higher chance of selection, and the statistic fraction of the resources allocated to one process is determined by the relative number of tickets that are held by the process.

Lottery scheduling is starvation-free because any process that is holding at least one lottery ticket has the probability of being selected. The algorithm is applicable to the management of diverse resources and provides a flexible control on the resource consumption rates of processes through a number of ticket mechanisms, such as the ticket transfer and ticket inflation [63]. However, being a randomized algorithm, lottery scheduling offers proportional fairness only over a large number of time quantum allocations. In addition, due to the randomized property, the allocation error in lottery scheduling is not tightly-bounded and can increase over the time [68]. Specifically, the allocation error after a series of n_a allocations is $O(\sqrt{n_a})$ [69]. Besides, the

implementation of lottery scheduling is also a tricky and challenging work. Firstly, it requires implementing a reliable random number generator, and secondly, it may incur a high overhead in managing the index of all tickets.

2.2.5.7 Fair queuing

Fair queuing is a classic algorithm that is widely used in processor and network scheduling to achieve a proportional allocation of resources. Different from the WRR and lottery scheduling, the implementation of a fair queuing algorithm relies upon the using of *virtual time* to track the service allocation in both the real system and the idealized GPS system. The concept of virtual time was firstly invented in network packet scheduling to resolve the unfairness of resource allocation that may be generated in WRR scheduling when the data packets are of variable length [70]. Naturally, this approach is employed to achieve a better unfairness bound in processor scheduling, in which the CPU burst time can also vary from one task to another.

The main purpose of the virtual time is to emulate the GPS system and use it as a reference model to the real system scheduling [57]. To achieve that, a non-decreasing function called *system virtual time* $V(t)$ is defined to track the normalized CPU time that is served to all the tasks. Ideally, the system virtual time $V(t)$ evolves at a rate that is inversely related to the weight sum of all active tasks, it is expressed as [57, 70]:

$$V(t) = \int_0^t \frac{1}{\sum_{\forall j \epsilon A(\tau)} w_j} d\tau \tag{2.4}$$

where $A(\tau)$ denotes the set of active tasks at time τ. In a practical implementation of PSS algorithms, the system virtual time can be computed in different methods with different computational complexity. The most expensive method is to concurrently emulate the GPS model and compute the system virtual time $V(t)$ based on the above integration equation. This incurs a computational complexity of $O(N)$, where N denotes the number of active tasks in the system. There are less expensive methods with which the system virtual time can be computed by self-referring to the

real-system scheduling model [59, 68]. However, the performance of the PSS algorithm may be affected. According to the discussion in [71], in order to provide a lag bound and delay bound within one standard time quantum or one maximum size packet, it is necessary for the system virtual time to have the minimum slope property, that is, the increasing slope of the system virtual time is at least one.

Besides of the system virtual time, the *task virtual time* $V_i(t)$ is defined to determine the scheduling order of tasks by tracking the normalized CPU time consumed by each task. More concretely, the task virtual starting time (or simply staring stamp) and the task virtual finishing time (or simply finishing stamp) are defined to separately record the normalized CPU time consumed by each task before and after it is allocated a time quantum. Depending upon the algorithm design, a fair queuing scheduler may select either the task with the lowest staring stamp or the one with the lowest finishing stamp to access the processor. The task virtual time $V_i(t)$ starts from zero, and it is updated when task i finishes the execution of a time quantum or when the task rejoins the resource competition after temporally leaving the run queue. To prevent any temporally-leaving task from continuously seizing the processor when they rejoin the competition with a smaller task virtual time, the system virtual time is used as a reference to update the task virtual time so that the latter can never suppress the former [55, 57].

Weighted fair queuing (WFQ) is one of the earliest fair queuing algorithms that strictly emulate the GPS model in the data packet scheduling [57, 72]. It is a well-studied algorithm and has been proven to have a bounded fairness and latency. Specifically, it is guaranteed that the WFQ system can never fall behind the GPS reference system by more than one maximum size packet, and the maximum delay of data flow f is bounded by $\frac{l_f^{max}}{r_f} + \frac{L_{max}}{C}$, where l_f^{max} denotes the maximum packet size of data flow f, r_f denotes the rate assigned to data flow f, L_{max} denotes the maximum size packet among all data flows, and C denotes the capacity of the network bandwidth. Unfortunately, WFQ fails to preserve the fairness when it is directly applied to the processor scheduling. This is because WFQ schedules tasks

based on the finishing stamp and it requires a priori knowledge of the length of each CPU time quantum. While in network scheduling, the length of each packet can be known from the header upon packet arrival; in processor scheduling, the length of CPU burst time is unknown and usually impossible or difficult to predict. This problem is resolved in the stride scheduling [69], the first fair queuing processor scheduling algorithm that supports fractional and non-uniform quanta and provides deterministic guarantees on resource shares. In stride scheduling, tasks are also scheduled in the increasing order of the finishing stamp (known as finishing pass). Although the finishing stamp of each task is computed based on the standard time quantum Q, a pre-defined time quantum with constant size, its value is modified to reflect the actual length of the execution time when the task relinquishes the CPU.

As stride scheduling [69] provides a time-quantum-based solution for fair queuing implementation, there is no theoretical proof of the unfairness bound and the allocation error. Actually, stride scheduling is believed to have an allocation error that can increase linearly to N, the number of active tasks [55]. The $O(N)$ lag is due to the continuously scheduling of the task with the lowest finishing stamp, which may cause the task to advance far ahead of the GPS model. Another problem experienced by fair queuing processor scheduling is the unfairness that may be introduced in dynamic task participation. In processor scheduling, if a task is allowed to dynamically join or leave the resource competition at any time, it may leave the competition before being allocated a requested time quantum (positive lag) or right after using up a CPU time quantum (negative lag). This introduces to the remaining tasks an unfairness that can be quantified by the non-zero lag of the leaving task. To address the problem, stride scheduling defines a state variable called *remain* to store the lag of each temporally-leaving task. When the task rejoins the system, the remain is used to compensate the updating of the task virtual time, so that the task with positive lag is favored to receive time quantum while the task with negative lag is punished [69]. This approach introduces a time complexity of $O(lgN)$ for both operations regarding the task leaving and rejoining. However, it is not quite effective to maintain the fairness because it is based on the assumption that a current time

quantum is equivalent to a future time quantum [55]. Extra unfairness will be incurred if the set of active tasks varies significantly between the instant of time that a task leaves and rejoins the system [58].

The earliest eligible virtual deadline first (EEVDF) [55, 58] is a fair queuing algorithm that improves the lag bound and unfairness bound of stride scheduling and provides a strict theoretical proof. Actually, EEVDF has been proven to have the optimal lag bound that equals to the length of the standard time quantum, Q, and the maximum delay of time quantum q_f^j is bounded by $\frac{q_f^j}{r_f} + \frac{Q}{C}$, where r_f denotes the rate assigned to task f, and C denotes the CPU bandwidth. In EEDVF, tasks are also scheduled according to the finishing stamp (known as virtual finishing deadline). However, only those tasks whose starting stamp is smaller than the system virtual time are considered as eligible to be scheduled. Since a task is only eligible when it receives less service time than it is entitled to in the GPS model, no task can advance ahead of the GPS model by more than one standard time quantum. To maintain the fairness upon dynamic task participation, EEVDF employs a more effective and systematical approach based on its interpretation of fairness in dynamic systems: any non-zero lag of leaving tasks should be proportionally distributed to the remaining tasks in the system [58]. The approach can be summarized in three steps. First, it delays whichever task with a negative lag to leave the system until the lag becomes zero, while allows tasks with a non-negative lag to immediately leave the system. Then, any positive lag is proportionally distributed to the remaining active tasks by compensating the system virtual time with a value that can be expressed as $\frac{lag_j(t)}{\sum_{i \in A(t) \setminus \{j\}} w_i}$, where $\sum_{i \in A(t) \setminus \{j\}} w_i$ denotes the weight sum of all active tasks after task j leaves the system. Finally, each task that newly joins or rejoins the competition is assigned a zero lag. With this approach, EEVDF maintains the fairness upon dynamic task participation and no extra unfairness is incurred as in the stride scheduling.

The implementation of any PSS algorithm should be practical and has reasonable computational complexity. Generally, the computational complexity of a fair queuing algorithm comes from three types of operations: virtual time stamp computation for

each task, task sorting in the run queue, and other operations that are specifically required to implement each algorithm, such as re-computing the task virtual time after the termination of a time quantum in stride scheduling and EEVDF. The complexity incurred by task sorting is $O(lgN)$ in all fair queuing algorithms, where N denotes the number of active tasks in the system. However, the complexity of time stamp computation depends on whether the ideal GPS system is required to compute the system virtual time $V(t)$. In both stride scheduling and EEVDF, the complexity of time stamp computation is $O(N)$ because the ideal GPS model must be emulated concurrently to support the computation of system virtual time through the integration equation (2.4). To reduce the expensive overhead of GPS emulation, it has been proposed to compute the system virtual time by self-referring to the task virtual time of the active tasks in the real system. This approach is free of the GPS model emulation and can reduce the complexity of time stamp computation to $O(1)$.

Self-clock fair queuing (SCFQ) [59] is one of the earliest and most well-known fair queuing algorithms that compute the system virtual time based on self-reference. Similar to stride scheduling, SCFQ schedules tasks in the increasing order of the finishing stamp. However, to reduce the scheduling overhead, the system virtual time is defined equal to the finishing stamp of the task under execution. In other words, the system virtual time is defined to track the lowest finishing stamp of all active tasks. SCFQ is simple to implement and its unfairness bound is near optimal in the sense that it is only twice the optimal bound [59]. However, because the system virtual time in SCFQ does not increase with a minimum slope of one, the allocation error and delay bound under SCFQ are not guaranteed within a standard time quantum, but increase with the growth of the number of active tasks. More specifically, the maximum delay of time quantum q_f^j is bounded as $\frac{q_f^j}{r_f} + \sum_{1 \le i \le N \wedge i \ne f} \frac{q_i^{max}}{C}$, where r_f denotes the rate assigned to task f, q_i^{max} denotes the maximum time quantum of task i, and C denotes the CPU bandwidth.

Starting-time fair queuing (SFQ) [68] is another fair queuing algorithm that is free of the GPS model. Unlike SCFQ, it defines the system virtual time equal to the

starting stamp of the task under execution and schedules tasks in the increasing order of the starting stamp. Compared to SCFQ, SFQ is even more computationally efficient because it does not need to modify the task virtual time after the termination of a time quantum; yet, the delay bound in SFQ is considerably smaller than in SCFQ while both have the same unfairness bound that is near-optimal. Unfortunately, like SCFQ, because the system virtual time of SFQ does not meet the minimum slope property, the lag bound and delay bound of SFQ also increase with the growth of the number of active tasks in the system. Specifically, the maximum delay of time quantum q_f^j is bounded as $\frac{q_f^j}{C} + \sum_{1 \le i \le N \wedge i \ne f} \frac{q_i^{max}}{C}$. However, recall that the delay bound of in WFQ and EEVDF is $\frac{q_f^j}{r_f} + \frac{Q}{C}$, in comparison, SFQ provides a better delay guarantee to low throughput tasks (such as interactive tasks) when the number of tasks is small.

Until now, all the above discussion and comparison of the fair queuing algorithms are based on the assumption that the system server bandwidth, such as the network bandwidth or CPU bandwidth, is constant. However, the capacity of a real-system server can be variable over the time and this situation is not considered in many fair queuing algorithms that assumes a constant rate server, such as the WFQ, stride scheduling and EEVDF. In particular, it is illustrated that WFQ fails to achieve the claimed fairness over a variable capacity server [68]. Implicitly, any fair queuing algorithm (including the stride scheduling and EEVDF) that computes the system virtual time under the assumption of a constant CPU bandwidth may potentially fail to provide its claimed unfairness bound and delay bound over variable rate servers. Conversely, fair queuing algorithms (including SFQ and SCFQ) that compute the system virtual time by self-referring to the real system can preserve their claimed fairness over variable rate servers. Actually, it has been proven that SFQ can provide a near-optimal unfairness bound under any variable rate server [68] and that it can provide bounds on maximum delay and minimum throughput (or share) for each task under a fluctuating bandwidth that can be modeled as a Fluctuation Constrained (FC) server or Exponentially Bounded Fluctuation (EBF) server [68]. The FC server [73] is

defined as a lower-bounded server that can be expressed as $(C, \delta(C))$, where C denotes the long-term average capacity and $\delta(C)$ denotes the maximum delay of resource serving in the FC server. In other words, a FC server serves at most $\delta(C)$ less resource than a server with a constant capacity of C in any interval of a busy period. The EBF server [73] is simply a stochastic relaxation of the FC server.

2.2.5.8 Multimedia scheduling

To better support soft real-time and multimedia applications in GPOSs with proportional share scheduling, several algorithms that combine certain real-time friendly mechanism to proportional share scheduling have been proposed [50, 74, 75]. Basically, those algorithms break the short-term fairness to give scheduling priority to time-sensitive tasks, but maintain the proportional power sharing from the long-term.

SMART [50] is one of earliest PSS algorithms that are specifically enhanced to support multimedia scheduling. The crux of SMART is to distinguish between urgency and importance when making scheduling decisions. While urgency is specific to real-time applications and measured by the time constraints, importance is common to all applications. The importance is measured by a value-tuple consisting of a static priority and a biased virtual finishing time (BVFT) that reflects the normalized energy received by a task. In the computation of BVFT, a bias is added to the virtual finishing time (or finishing tag) of regular tasks when completing a quantum, so that regular tasks are deferred to let real-time tasks get scheduled earlier to meet their time constraints. The bias affects instantaneous proportional allocations and worsens fairness bound, but does not change the long-term proportional share of resources. Generally, SMART makes the scheduling decision in two steps. In the first step, it identifies the most important task (the one with the highest value-tuple), if the most important task is a regular one, the task is scheduled immediately; if not, it enters the second step. In the second step, all real-time tasks which are more important than the most important regular task are organized in a queue named working schedule, and among them, the most urgent one is selected to execute.

In comparison with traditional PSS algorithms like EEVDF, SMART introduces time-constraint awareness to the scheduler, and thus, provides better real-time performance while allowing proportional sharing of resource based on user-desired share ratio. In addition, SMART integrates static priority into the proportional share scheduling and allows prioritizing tasks across real-time and non-real-time classes. Besides, SMART provides dynamic feedback to real-time applications to allow them adapting properly to the current load. However, SMART experiences several practical issues in the implementation. Firstly, SMART incurs computing overhead in managing both the value-tuple list and the working schedule queue. Since the system virtual time is updated in reference to a GPS system and tasks are ordered based on the virtual finishing time (or finishing tag), the cost of managing the value-tuple list is similar to WFQ. It requires a time complexity of $O(n_c)$ for finishing tag computations and $O(lgn_c)$ for service quanta sorting. The complexity of managing the working schedule queue is $O(n_R^2)$, where n_R is the number of real-time tasks in the candidate set. This complexity can be further reduced to $O(n_R)$, but it requires a more complicated implementation scheme. Secondly, SMART predicts the service time required by a periodical real-time task in its future periods, tests if the required service time can be served before its deadline, and based on that decides whether insert a task into a working schedule queue or abandon its service request. This method is sensitive to the prediction error, thus, introduces risks of abandoning a service request that can actually meet its deadline. Finally, SMART introduces a bias to the virtual finishing time of regular tasks to preferentially schedule time-sensitive tasks, but does not provide any user interface to support a flexible control on the bias.

BERT [74] is another real-time friendly PSS algorithm that was developed in reference to SMART. Since it shares several common features with SMART, such as the requirement of simulating the GPS system and the need to compute the virtual finishing time based on predicted length of time quanta, it suffers the high computing overhead and implementation complexity as in SMART.

Borrowed-virtual-time (BVT) [75] is an effective yet low-complexity PSS algorithm targets on the support of real-time and multimedia scheduling in GPOSs.

BVT employs a real-time friendly mechanism named warping to support time-sensitive tasks in proportional share scheduling. The warping mechanism involves the following state variables of each task: A_i, the actual virtual time (AVT); E_i, the effective virtual time (EVT); W_i, the virtual time warp; and $warpBack_i$, a bool sets whether the warp is enabled or not. The effective virtual time E_i is computed as $A_i - W_i$ if the warp is enabled ($warpBack_i = 1$) for task T_i; otherwise, if the warp is not enabled ($warpBack_i = 0$), the effective virtual time E_i equals to the actual virtual time A_i. BVT monitors the task execution progress with the actual virtual time A_i, but schedules tasks in the increasing order of the effective virtual time E_i. By enabling the warp to warp back the virtual time stamp, a time-sensitive task appears earlier in the scheduling queue and gains dispatch preference. The fairness worsens due to the dispatch preference given to time-sensitive tasks; however, the long-term CPU bandwidth share is still constrained by the weighted fair sharing of BVT because the actual virtual time A_i is advanced based on its actual CPU usage. Warping a task can introduce latency to other lower-priority tasks. Thus, BVT introduces two additional parameters to warping: the warp time limit L_i that limits the maximum time one task can run warped; and the unwarp time requirement U_i that governs the time a task must wait before warping again. These two warp parameters should be properly set to limit the CPU occupation of higher-priority tasks and thus avoid adding too much latency to other tasks. BVT provides a user interface to support a flexible setting of these two parameters.

The warping mechanism is simple and straightforward to implement in any WFQ-like fair-queuing algorithm. With a proper choice of the warp parameters, BVT can reduce the dispatching latency for real-time and interactive tasks while providing weighted proportional sharing of the CPU bandwidth across time-sensitive and regular tasks. However, since the interaction of multiple warped tasks with multiple levels of warp values has not been investigated and it is still an open question how various warp parameters should be set to produce a desired overall system behavior, more research on the warping mechanism are required [74].

2.3 Summary and discussion

In this chapter, the related works of energy-centric processor scheduling have been surveyed from two domains, OS-level power management and GPOS scheduling algorithms.

Different from traditional OS-level PM schemes which are either best-effort in energy saving or dependent on application self-adaptation, energy-centric PM schemes can provide a strict guarantee on the user-specified battery lifetime by globally managing energy as the first-class resource in mobile systems with general applications. While the guarantee of a specific battery lifetime meets the basic requirement of a mobile system user, the energy-centric processor scheduling is pivotal in user experience optimization on top of the guaranteed battery lifetime. Specifically, to optimize the user experience of an energy-limited mobile system, energy-centric processor scheduling is required to provide supports in proportional power sharing, time-constraint compliance, and a flexible trade-off between them. Until now, no energy-centric processor scheduling algorithm that has all the above-mentioned properties has been proposed. Although certain former work has been done on energy-centric processor scheduling and it has been claimed that proportional power sharing is achievable, it is based on the assumption of general purpose workload with no information on the time constraints and neither formal description nor concrete implementation method of the algorithm is provided.

Develop energy-centric processor scheduling algorithms need to first consider the general scheduling requirements in GPOSs. As a large number of processor scheduling algorithms have been developed for GPOS scheduling, it is natural to add energy-awareness to the existing GPOS scheduling algorithms and extend them to the energy-centric scheduling domain to form energy-centric processor scheduling algorithms. Basically, a GPOS processor scheduler should provide differential quality of service (QoS) to the applications while avoid any application being starved for resource accessing; in addition, the scheduler should be practical in implementation as well as low overhead in computing and scheduling. Priority scheduling based on static

parameters like the user preferences and application importance is a natural way to provide differential QoS to applications, but there are two problems: first, it does not provide fairness in resource sharing, low-priority applications may experience resource starvation if high-priority tasks are never or rarely blocked for execution; second, when there are multiple real-time applications that are simultaneously executed with high priorities (in comparison with the low priority of regular and interactive applications), if the priority assignment does not take into account application properties such as the cycle duration and approaching deadlines, the scheduler may fail to achieve strict time-constraint compliance for all real-time applications. Although approaches like priority aging and priority demoting are available for avoiding resource starvation on low-priority applications, the implementation complexity is greatly increased and more risks are brought to the time-constrain compliance of real-time applications. To ensure strict time-constraint compliance and avoid resource starvation in GPOSs, real-time scheduling algorithms, such as rate-monotonic (RM) and earliest-deadline-first (EDF), are commonly combined with admission control and resource reservation mechanisms to support real-time and multimedia scheduling in GPOSs. While the resource reservation allows reserving certain amount of processor time to each real-time application, the admission control can set the maximum share of resource that is reservable for real-time applications, thus leaving certain amount of unreserved computational time to other regular and interactive applications. However, the scheduler based on resource reservation is non-working-conserving in a way that any over-reserved and unused resource will neither be available for reservation by other real-time applications nor for sharing with regular applications. Make it worse, to protect the already-reserved resource, the admission control may deny a later coming but more important application while the system is working in a lightly loaded state.

Fair queueing-based proportional share scheduling (PSS) can achieve a proportional sharing of the common resource upon non-uniform service quanta in accordance with the pre-defined weight of each application. Because each application can be guaranteed certain share of resource that is specified by the user or determined

by the application workload, fair queuing-based PSS is able to provide a differential yet fair sharing of the schedulable resource. In addition, the time-constraint compliance of real-time applications can be ensured under PSS as long as an adequate resource share is allocated to each real-time application, and real-time friendly mechanisms can be further combined into the fair queuing-based PSS to improve its support on real-time and multimedia scheduling. Besides, different from the resource reservation in RM and EDF scheduling, PSS is work-conserving in the sense that the over-allocated resource share can be reallocated to other regular applications. Considering all these properties, the fair queuing-based proportional share scheduling is regarded as a proper reference candidate to be extended to the energy domain for the development of energy-centric processor scheduling algorithms.

Fair queuing scheduling algorithms are implemented to approximate the ideal Generalized Processor Sharing (GPS) model to achieve a proportional and fair sharing of the common resource. The GPS model has been applied in both the network and CPU scheduling domains, and based on that, fair queuing algorithms have been developed to achieve a proportional sharing of the network bandwidth or CPU bandwidth among the different schedulable entities. Similar to the network bandwidth and CPU bandwidth, energy is a limited resource that is commonly shared by different tasks with various energy requirements. Equivalence can be built between the (network or CPU) bandwidth allocation and energy allocation. In packet-based network scheduling, the service received by each session is data measured in bits, each session is allocated a share of the network bandwidth that is defined as its rate of data transmission and measured in bits per seconds (bps); in time-quantum-based CPU scheduling, the service received by each task is time measured in CPU clock cycles, each task is allocated a share of the CPU bandwidth that is defined as the rate of its execution on CPU and measured in cycles per second or in Hz; similarly, we can develop fair queuing algorithms in the energy domain by considering the energy (measured in Joules) as the service, then each task will be allocated a share of the system power that is defined as its rate of energy consumption and measured in Watt. Therefore, the GPS model can also be applied to model the energy sharing among

different tasks, and based on that, energy-based fair queuing (EFQ) algorithms can be developed to achieve a proportional and fair sharing of the system power. The extension of the fair queuing and the GPS model for supporting energy-centric processor scheduling will be presented in the next chapter.

Chapter 3

Energy-Centric Processor Scheduling

Fair queuing-based proportional share scheduling is considered as a proper reference algorithm for the development of energy-centric processor scheduling. In this chapter, the fair queuing algorithm and the ideal Generalized Processor Sharing (GPS) model are implemented in the energy sharing domain, and the energy-based fair queuing (EFQ) algorithm is proposed for achieving proportional power sharing and time-constraint compliance in energy-centric processor scheduling. To simplify the work and focus on the processor scheduling, this chapter starts with a set of assumptions and conditions on the energy accounting, the energy allocation, and the schedulable entities. Then, in reference to the Generalized Processor Sharing (GPS) model and the traditional fair queuing scheduling models of network and CPU sharing, the energy-centric scheduling model is proposed as the basis of algorithm design for energy-centric processor scheduling. After that, an insight of the power share management under the GPS-based energy-centric scheduling model is provided, and specific mechanisms are proposed for the protection and reallocation of the power shares of time-sensitive tasks. Finally, energy-based fair queuing (EFQ) algorithms are designed for energy-centric processor scheduling to achieve the proportional power sharing and the time-constraint compliance.

3.1 Assumptions and Conditions

The CPU scheduler is a core component in the operating system that can interact with a variety of different system components. Therefore, before the presentation of the scheduling algorithm design, a series of conditions and assumptions should be built on the scheduling surroundings. This section provides a statement of the conditions and assumptions that our scheduling proposal is based on.

In a typical energy-centric system, the proper functioning of an energy-centric CPU scheduler relies on the cooperation and interaction with a number of hardware and software components. Their relationships are abstracted in Figure 3.1.

As can be seen in Figure 3.1, the energy-centric CPU scheduler interacts with the energy consumptions not only on the CPU but also on the other hardware components; it relies on the cooperation with the energy allocation module and especially the energy accounting module to achieve the above interactions.

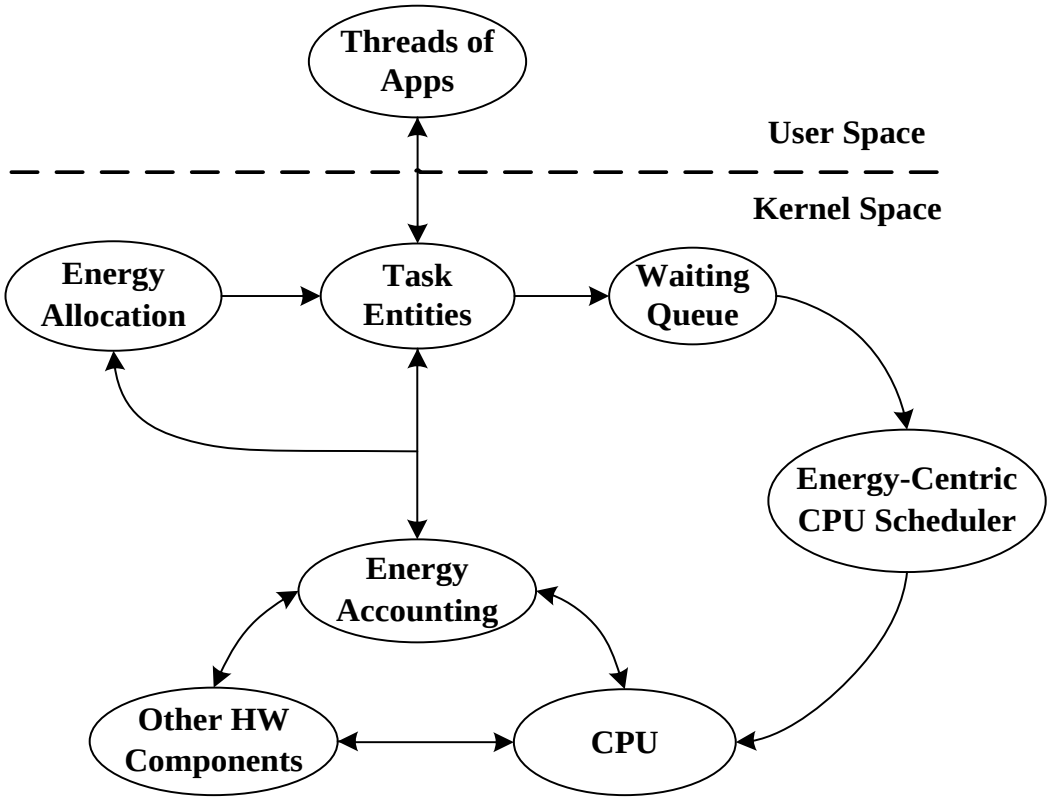


Figure 3. 1: Energy-centric CPU Scheduling Surroundings

In a modern mobile system that is running multiple applications, the cooperation and interactions between the different components can be considerably complex. For instance, each application can create multiple threads in the user space, and in the kernel space, there will be one task entity structure maintained for each user-space thread and thus the applications are scheduled in the unit of thread; however, to achieve a proportional share of energy and power between different applications, the battery energy should be proportionally allocated to each application instead of thread,

while the energy consumptions caused by thread activities on various hardware devices should be correctly mapped to different applications. The above situation will be further complicated if inter-thread and inter-process communications are considered between different threads and applications.

To simplify the energy-centric scheduling surroundings and focus on the CPU scheduler design, a series of conditions and assumptions have been built up for the applications, the energy allocation module, and the energy accounting module. They will be stated in the following sub-sections.

3.1.1 Applications, threads, and tasks

For simplicity, the first assumption made in this work is that each application is a single threaded process that is independent to other applications. That is to say each application has only one thread in the user space and is correlated to a unique task entity structure in the kernel space. Therefore, the terms application, process, thread, and task mean the same thing in this dissertation, and they are used interactively in different parts of the dissertation. With this assumption, the energy allocation, energy accounting, and energy scheduling are greatly simplified because they can all be considered based on the task entity. Note that, in the future, our energy-centric CPU scheduling algorithm can be enhanced to schedule multithreaded applications by organizing tasks in groups and applying group scheduling strategies; also, the scheduling algorithm can be extended to multi-core processors by applying additional load balancing policies. However, in this dissertation work, we only consider the individual task as the scheduling entity and focus on the development and verification of the very fundamental energy-centric scheduling algorithms for single-core processor.

Even single threaded applications can have a variety of different energy requesting patterns, so the second assumption made in this dissertation is related to the characterization of the task working patterns. In general, the tasks are separated into three categories: batch tasks, interactive tasks and real-time (both soft and hard) tasks. It is assumed that batch tasks can continuously consume energy and never go

idle before their termination, while interactive and real-time tasks are periodic ones that can go idle between the periods. However, the period of real-time tasks is assumed to be a fixed value all the time, while the period of interactive tasks are considered as variable to reflect the aperiodic of user requests. Besides, different metrics are applied to interactive and real-time tasks to measure the performance. While the performance of interactive tasks is measured by the response time of the energy request in each period, the performance of real-time tasks are measured by the percentage of deadline misses in all periods. In total, real-time tasks and interactive tasks are collectively called periodic time-sensitive tasks in this work.

3.1.2 Energy accounting

Energy accounting is fundamental to an energy-centric power management scheme because it enables a tracking of the amount of energy consumed by each task, only by referring to the energy consumption information of each task can the CPU scheduler make energy-centric scheduling decisions. However, achieve accurate energy accounting is a complex and challenging work, as shown in Figure 3.1, it deals with both the hardware devices in the bottom level and the tasks in the high-level. Therefore, to focus on the high-level scheduling algorithm, it is assumed in this dissertation work that energy consumptions on the different hardware devices can be correctly mapped to the specific tasks.

With the above assumption, an energy model that abstracts the power consumptions on low-level hardware devices is necessary for the design of high-level energy-centric CPU scheduling algorithms. While the CPU scheduler is unwaring of the task operational time on other devices, it does have the information of each task’s CPU occupation time. To build the abstracted energy-centric scheduling model, the system-wide power consumption caused by one task should be in certain way correlated to the CPU occupation time of the task. However, as we observe the system-wide power changing when executing a task, it can be found that the system power can vary greatly over the time and the task can still consume energy even if it is not occupying the CPU any more.

Figure 3.2 demonstrates the current change of an embedded system when a program is executed alone on the system. The program firstly executes CPU-intensive computations and later writes the computation results into the SD card of the system. The voltage is a stable value during the execution so it is not specifically given. As can be observed, the system power increases significantly when there are extra I/O operations caused by the program; besides, even after the program has quitted the CPU, extra power consumptions are still caused by the I/O operations to the SD card. While the fluctuating system power can be approximately modelled by certain random functions, the asynchrony between the system power and the CPU activity can significantly complicate the energy-centric CPU scheduling model. Therefore, to simplify the model, another assumption in made assuming that the system power is synchronized to the CPU activities, and when computing the system power of a task, the asynchronous energy consumptions are accounted onto the CPU occupation time. Under this assumption, we have developed an energy-centric scheduling model that will be presented in detail in the next section (section 3.2).

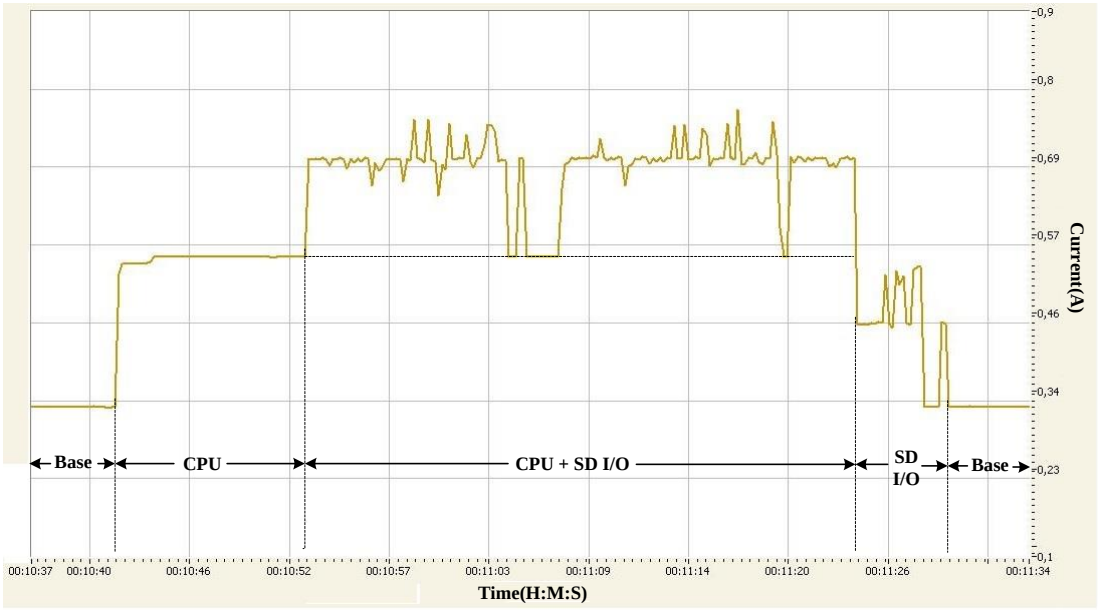


Figure 3. 2: Example of System-wide Power in Reference to CPU Execution Time

Note that the above assumption on energy accounting will not preclude the author from properly designing and assessing the behavior of the energy-centric scheduling

algorithm from the high level, because the scheduler just does not care about low-level details such as in which hardware device the energy is spent. What really matters to the scheduler is how much energy is consumed by which task at a specific time. In the future when practical and accurate energy accounting mechanisms are available and easily applicable for modern OS-based mobile systems, these mechanisms can be combined with the high-level energy-centric scheduling algorithms to achieve a complete system cycle of energy-centric power management over a variety of different hardware devices.

3.1.3 Energy allocation

Although the energy allocation module is not necessarily needed for the normal functioning of the energy-centric CPU scheduler, it plays a pivotal role in guaranteeing a user-specified battery lifetime to the user-preferred applications. In other words, a proper energy allocation over the time and among the applications is the prerequisite of a meaningful energy-centric scheduling and PM scheme. Therefore, in this dissertation work, the energy-centric CPU scheduling algorithm is researched based on the condition and assumption that, the idea of *epoch* [4] is adopted to restrict the battery discharge rate and guarantee a target lifetime, and in the meantime, the idea of *reserve* [19] is adopted in each *epoch* to reserve or limit a certain amount of battery energy for an application.

3.1.4 Whole view

Under the above conditions and assumptions, a mobile operating system is considered as a set of tasks $\{T_1, \dots, T_n\}$ (real-time, interactive, and batch) competing for the total amount of energy $E_{battery}$ that is available in the battery. The system has a target lifetime T_{target}, which is divided into m periods of time, termed epochs, with length T_{epoch}^{i}. In each epoch, the energy available for consumption, known as E_{epoch}^{i}, is a limited value. Then, E_{epoch}^{i} is further allocated to different tasks in two

ways: for a user-preferred task, a certain amount of energy is reserved to ensure its normal execution during the whole epoch; and for a less-favored task, a maximum limit of energy consumption is imposed in each epoch. The combinational use of the *epoch* and *reserve* mechanisms in the energy allocation provides a flexible yet strict guarantee of battery lifetime to the user-preferred tasks. This builds the prerequisite of user experience optimization for battery-limited mobile systems.

For the sake of simplicity and without loss of generality, in this thesis work, the research on the optimization of the user experience is focused on one epoch out of the whole target lifetime. Within one epoch, the optimization of the user experience relies on the fulfillment of two conditions. First, all user-preferred applications should be guaranteed to run with the user-acceptable performance during the whole epoch; and second, the remaining energy at the end of one epoch should be minimized to ensure the maximization of the epoch energy E_{epoch}^{i} and so that the total system performance.

To fulfill the first condition, the scheduler has to be well designed to achieve proportional power sharing among tasks and in the meanwhile provide a strong support to the time-sensitive tasks; the scheduler design will be presented in detail in the following sections.

And to fulfill the second condition, the energy allocation among tasks should be properly and dynamically set in each epoch to achieve that the epoch energy E_{epoch}^{i} is exhausted right at the end of the expected epoch time. If the energy quota of user-preferred tasks is conservatively reserved while the energy quota of those less-favored tasks is overly restricted, the E_{epoch}^{i} will not be exhausted during the expected epoch time, which indicates a loss of the opportunity to improve the whole system performance during the epoch. In contrast, if the energy restriction is too loose, the E_{epoch}^{i} may be drained before the expected epoch time by those less-favored but energy-greedy tasks.

Since the work of this dissertation is focused on the energy-centric scheduler design, the mechanisms of optimizing the energy usage in each epoch are not specifically investigated here; it is simply assumed that a proper energy allocation has been implemented for the tasks in each epoch.

3.2 Energy-centric scheduling model

In this section, the Generalized Processor Sharing (GPS) [57] model is extended to the energy sharing domain, and based on that, the energy-centric scheduling model is introduced as the basis of the energy-based fair queuing algorithm design for energy-centric processor scheduling.

Based on the assumptions and conditions in section 3.1, the energy-centric scheduling problem can be formulated considering the set of tasks $\{T_1, \dots, T_n\}$ (real-time, interactive, and batch) competing for a limited amount of energy E_{epoch}^{i} during the i^{th} epoch. Each task T_i is assigned a weight w_i that determines its minimally guaranteed share of the system power. Then, in the ideal Generalized Processor Sharing (GPS) model, which assumes energy as an infinitely divisible resource that can be simultaneously served to multiple tasks, the power of task T_i at time t, $P_i(t)$, is at least:

$$P_i(t) = P(t) \cdot \frac{w_i}{\sum_{\forall j \epsilon A(t)} w_j} \tag{3.1}$$

where $P(t)$ denotes the system power and $A(t)$ denotes the set of active tasks at time t.

The ideal GPS model considers energy sharing as a continuous fluid model with perfect fairness. In a real mobile system, however, it is required to consider a discrete energy distribution both over time and across the different hardware components. Firstly, multiple tasks cannot simultaneously gain access to a single component; therefore, energy is served to tasks along with the assigned discrete time quanta. Secondly, one task can consume energy by accessing different components.

While it is a complex work to exactly model the energy sharing and the system power in a real system, by assuming that the system power $P(t)$ is synchronized to the CPU activities (refer to section 3.1.2), the energy consumption of a real system can be abstracted to a model in which the system power is a piecewise constant function of the CPU time. Their relationship is illustrated in Figure 3.3. During different intervals, the system power value is dependent on the specific hardware activities caused by task execution. Note again that, the above assumption will not preclude the author from properly designing and assessing the energy-centric CPU scheduling algorithm, because it is not the responsibility of the high-level scheduler to account for where the task has spent its energy.

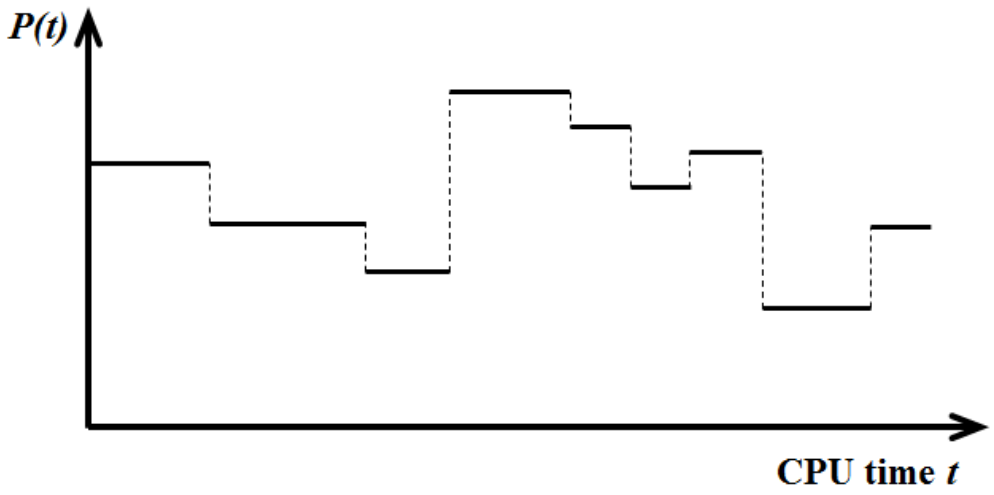


Figure 3. 3: Illustration of the relationship between the system power and CPU time

Therefore, the energy sharing in a real system can be modelled considering that energy is allocated to the active tasks along with discrete CPU time quanta. For each CPU time quantum allocated to a task, there is a corresponding amount of energy consumption that is synchronously spent on the task. A task is selected to receive energy at the beginning of a time quantum; it may run on the CPU for the entire time quantum or release it before the end of the time quantum. This is realized by dividing the service time of task T_i into a number of time quanta q_i^k with maximum length Q, and defining the energy consumption of task T_i during the time quantum q_i^k as an

energy packet e_i^k. Specifically, a CPU time quantum with the maximum length of Q is called the standard time quantum, and the energy consumption during a standard time quantum is called the standard energy packet. Both time quantum q_i^k and energy packet e_i^k are regarded as service in this context, and thus, they are collectively called as service quantum, marked as sq_i^k.

While one task can be allocated two service quanta with the same length of CPU time quanta, the size of the two energy packets may vary because of the different hardware activities caused by specific task functions. Due to the quantization of the system service, it is impossible for a task to always consume exactly the same amount of energy as in the ideal GPS model. The difference between the energy a task consumes in the GPS model and the energy it actually consumes in the real system is defined as the energy allocation error.

For clarity, in this dissertation work, the basic unit of CPU time quantum is uniformly defined as one CPU time unit (Tu), and the basic unit for measuring energy consumption is defined as one energy unit (Eu), as such, the basic unit for measuring power is Eu/Tu, defined as one power unit (Pu). Now, recall the general comparative discussion on applying the fair queuing scheduling in the domain of network, CPU and, energy (in section 2.3), we can reach to a detailed comparison of them as shown in Table 3.1.

Table 3. 1: Comparison of the Proportional Share Scheduling Models for Network, CPU and Energy

	Network	**CPU**	**Energy**
Scheduling resource	Data in bits	CPU in cycles	Energy in Eus
Allocation share	Bandwidth in bps	Bandwidth in Hz	Power in Pus
Scheduling entities	Sessions	Tasks	Tasks
Schedulable units	Packet in bits	Time quantum in cycles	Time quantum in Tus &Energy packet in Eus

In comparison to the proportional share scheduling model in the network and CPU domain, applying proportional share scheduling (PSS) algorithms in the energy domain is a more complex and challenging work considering the uncertain relationship between the CPU time quantum and the energy packet. Based on the energy-centric scheduling model built in this section, the design of energy-centric proportional power share scheduling algorithms will be presented in section 3.4.

3.3 Power share management

This section provides an insight into the power share management under the GPS-based energy-centric scheduling model before the energy-centric scheduling algorithm is presented in the next section (section 3.4).

3.3.1 Maximum long-term and worst-case power shares

3.3.1.1 Definitions

Batch tasks can continuously receive energy until their total energy demands are all met, and then, they finish the work and become completely passive. Therefore, increase the weight of a batch task will always increase its power share. However, this is not true for periodic time-sensitive tasks. Once a time-sensitive task receives its demanded energy in one period, it stops receiving energy and becomes passive until next period begins. Thus, different from batch tasks, increase the weight of a periodic time-sensitive task may not always increase its power share; there will be bounds in the power share allocation. Generally, in periodic time-sensitive tasks, both the number of service quanta per period and the size of each energy packet are not constant, leading to a variable energy load in each period. In this work, the maximum long-term power share and the worst-case power share are defined to describe the power allocation bounds of the periodic time-sensitive tasks with variable energy loads over the periods.

The maximum long-term power share is defined as the average power share of the system power that guarantees meeting the total energy demands of a periodic task

in a long-term static interval. Note that the term “long-term” in this work is a statistical concept with respect to the task period. The average power of one task during a long-term interval is called the long-term average power, or simply long-term power, while the average power during a task period is called the instantaneous average power, or simply instantaneous power. A long-term static interval is a certain number of task periods during which the set of tasks and the average power of each task are determined. The maximum long-term power share sets the upper bound of the long-term power share that can be allocated to a periodic time-sensitive task. Increase the weight of a periodic task can only increase its long-term power share up to the value of the maximum long-term power share. During the idle time of a periodic task, any over-assigned instantaneous power share will be re-allocated to the other active tasks that are able to receive a higher long-term power share; this point will be expanded in section 3.3.3. The value of the maximum long-term power share depends on many parameters, such as the long-term power of each task, the workloads of periodic tasks, and the weights of batch tasks. At the later part of this section, one example will be provided to demonstrate the computation of the maximum long-term power share.

The maximum long-term power share guarantees the serving of all energy that is demanded by a periodic time-sensitive task in a long-term interval. Therefore, assigning one time-sensitive task a weight that can guarantee a maximum long-term power share is the prerequisite of meeting the majority of the time constraints. In contrast, if any time-sensitive task is allocated a power share that is lower than its maximum long-term power share, it is likely to miss the most of its time constraints if the energy demand is not degraded. Since it makes little sense to keep executing a task that misses the most of its time constraints, periodic time-sensitive tasks, especially real-time tasks, should be assigned a weight to guarantee an instantaneous power share that at least equals to the maximum long-term power share. How a weight can be determined to guarantee a specific and constant power share to time-sensitive tasks will be expanded in the next section (section 3.3.2).

The worst-case power share is defined as the lowest instantaneous power share that is required to guarantee the meeting of the maximum energy demand among all periods of a time-sensitive task. For a periodic time-sensitive task, the maximum energy demand is also known as the worst-case energy load, it appears in the period during which the task has the worst-case execution time (WCET) and the maximum average size of energy packets at the same time. Therefore, the worst-case power share sets the lower bound of the instantaneous power share that is required by the periodic time-sensitive tasks to guarantee the meeting of the energy demand in all periods. By averaging the worst-case energy load with the task period length, we can obtain an average power that is called the worst-case instantaneous power. The value of the worst-case power share depends not only on the worst-case instantaneous power of the concerned task, but also on the power of other tasks. Theoretically, the worst-case power share is computed when the concerned task has the worst-case instantaneous power and all other tasks have the lowest instantaneous power. However, in real systems, the probability of having the above combinational situation is very low. Therefore, a practical method is employed to compute the worst-case power share considering the looser situation in which the concerned task has the worst-case instantaneous power and all other tasks have the long-term average power.

3.3.1.2 Example demonstration

To demonstrate the computation of the maximum long-term power share and the worst-case power share, table 3.2 shows an example of the power allocation based on 4 tasks. For simplicity, the length of all service time quanta is uniformly set as 1 Tu. R1 and R2 are periodic time-sensitive tasks with fixed length of period, while B1 and B2 are batch tasks with continuous energy demand. R1 has a period of 10 Tus; the service time in each period varies from 2 Tus to 6 Tus and averagely is 4 Tus, and the energy packet size varies from 6 Eus to 10 Eus with the average size as 8 Eus. R2 has a period of 15 Tus; the service time in each period varies from 2 Tus to 4 Tus and averagely is 3 Tus, and the energy packet size varies from 12 Eus to 18 Eus with the

average size as 15 Eus. The energy packets of batch tasks are also with variable sizes. Specifically, the energy packet size of B1 varies from 4 Eus to 8 Eus, with the average size as 6 Eus; the energy packet size of B2 varies from 10 Eus to 14 Eus, with the average size as 12 Eus. To compute the maximum long-term power share and worst-case power share of R1 and R2, the average size of the energy packets requested by all the batch tasks should be computed in advance. In this example, based on the weight ratio of 1:2 of B1 and B2, the average energy packet size of the two batch tasks is $(6 \times 1 + 12 \times 2)/3 = 10\ Eu$. In a real dynamic system where the weights of batch tasks are variable over time or not known in advance, the lower energy packet size can be directly referred to compute the maximum long-term and worst-case power shares; in this case, the obtained power shares are conservative ones with a larger value than the actual ones.

Table 3. 2: Example of Computing the Maximum Long-term and Worst-case Power Shares

<table>
<tr><th>Task</th><th>R1</th><th>R2</th><th>B1</th><th>B2</th></tr>
<tr><td>Period (Tus)</td><td>10</td><td>15</td><td colspan="2" rowspan="3">N/A</td></tr>
<tr><td>Number of service quantum / period</td><td>2-6</td><td>2-4</td></tr>
<tr><td>Average number of service quantum / period</td><td>4</td><td>3</td></tr>
<tr><td>Size of energy packets (Eus)</td><td>6-10</td><td>12-18</td><td>4-8</td><td>10-14</td></tr>
<tr><td rowspan="2">Average size of energy packets (Eus)</td><td rowspan="2">8</td><td rowspan="2">15</td><td>6</td><td>12</td></tr>
<tr><td colspan="2">10
(weight ratio = 1:2)</td></tr>
<tr><td>Maximum long-term power (Pus)</td><td>3.2</td><td>3</td><td colspan="2">lim(x → 10)</td></tr>
<tr><td>Maximum long-term power share</td><td>0.314</td><td>0.294</td><td colspan="2">lim(x → 1)</td></tr>
<tr><td>Worst-case instantaneous power (Pus)</td><td>6</td><td>4.8</td><td colspan="2" rowspan="2">N/A</td></tr>
<tr><td>Worst-case power share</td><td>0.545</td><td>0.424</td></tr>
</table>

First, we compute the maximum long-term power share. In the long-term, the average CPU occupation rate of R1 is at most $4/10 = 0.4$, even if R1 is assigned an infinitely high weight to allow it consuming energy in an infinitely high instantaneous power; therefore, the maximum long-term power of R1 is $0.4 \times 8 = 3.2\ Pus$.

Similarly, the average CPU occupation rate of R2 is at most $3/15 = 0.2$ in the long-term, and the maximum long-term power of R2 over the periods is $0.2 \times 15 = 3\ Pus$. The two batch tasks can achieve a maximum long-term power close to 10 Pus if their weights are infinitely higher than those of R1 and R2, correspondingly, the maximum long-term power share of the two batch tasks can be close to 1. However, to compute the maximum long-term power share and the worst-case power share of R1 and R2, both R1 and R2 are assumed to have a weight that allows them taking the maximum long-term power. In this case, the remaining CPU occupation rate for the two batch tasks is $1 - 0.4 - 0.2 = 0.4$, and the long-term power of the two batch tasks is $0.4 \times 10 = 4\ Pus$. Finally, based on the maximum long-term power of periodic real-time tasks and the long-term power of the two batch tasks, the maximum long-term power share is computed. For R1, the maximum long-term power share is $3.2/(3.2 + 3 + 4) = 0.314$; and for R2, the maximum long-term power share is $3/(3.2 + 3 + 4) = 0.294$.

Next, we compute the worst-case power share. Because R1 has a worst-case execution time of 6 Tus and a maximum energy packet size of 10 Eus, the maximum amount of energy that can be consumed by R1 in one period is 60 Eus, and the worst-case instantaneous power of R1 in one period is 6 Pus. When R1 has the worst-case CPU occupation rate of $6/10 = 0.6$, the maximum average CPU occupation rate of R2 is $3/15 = 0.2$ and the maximum long-term power of R2 is 3 Pus; therefore, the remaining CPU occupation rate for the two batch tasks is 0.2, and the long-term power for the two batch tasks is $0.2 \times 10 = 2\ Pus$. Finally, based on the worst-case instantaneous power of R1, and the long-term power of the rest tasks, the worst-case power share of R1 is $6/(6 + 3 + 2) = 0.545$. Similarly, the worst-case power share of R2 can be computed based on the worst-case instantaneous power of R2 and the long-term power of other tasks, the computation process is not repeated here, but the result can be found in Table 3.2.

As can be seen from the above example, the maximum long-term power share and the worst-case power share can vary with the system environment. These two parameters only keep constant in a long-term static interval where the set of active

tasks, their weights and average power do not change. In a real dynamic system, tasks may frequently join or leave the system, change their weights or significantly vary their energy loads. Therefore, both the maximum long-term power share and the worst-case power share are variable in different long-term static intervals.

3.3.2 Power share protection

3.3.2.1 Mechanism

As in the traditional fair queuing scheduling of network and CPU, the weight-assignment problem [62] also arises in the energy-centric scheduling model in which the power share of time-sensitive tasks should be protected from the dynamic activities of other tasks to ensure a stable real-time performance. Basically, the power share of a time-sensitive task should be no lower than its worst-case power share if a strict compliance of all deadlines is required; in another case, the assigned power share can be between the maximum long-term power share and the worst-case power share if a certain degree of response time and deadline miss ratio is acceptable. However, according to equation (3.1), the power share P_i of any task can vary with the number and weight of the active tasks in the system. Any new task that joins the competition with a large weight may reduce the power share of a time-sensitive task to a level that is lower than its maximum long-term power share, leading to the miss of the majority of its deadlines. Therefore, certain power share mechanisms should be employed to achieve a desired level of power share for time-sensitive tasks.

In this work, the approach to achieve power share protection is a combination of several CPU share protection proposals [62, 54, 65]. First, as in [65], each task is assigned an initial weight $\bar{w}_i$ and an initial power share $\bar{f}_i$ that ranges from zero to one ($\bar{f}_i \in [0,1]$). This allows supporting a mix of tasks in three categories:

- $\bar{f}_i = 0$ and $\bar{w}_i > 0$, for batch tasks that do not require a guaranteed power share for meeting time constraints. This type of tasks compete for the unreserved and released power share with the non-zero initial weight $\bar{w}_i$.

- $\bar{f_i} > 0$ and $\bar{w_i} = 0$, for time-sensitive tasks that have a specific power share requirement. This type of tasks can reserve a certain level of power share with the non-zero initial share, but do not compete for the unreserved and released power share.
- $\bar{f_i} > 0$ and $\bar{w_i} > 0$, for time-sensitive tasks that have a minimum power share requirement. In addition to the reserved power share, this type of tasks also compete for the unreserved and released power share with its non-zero initial weight $\bar{w_i}$.

The above three categories are further separated into two classes, the first category with $\bar{f_i} = 0$ and $\bar{w_i} > 0$ belongs to the best-effort class (BC), while the latter two categories with $\bar{f_i} > 0$ belong to the reserved class (RC). Since the power share of the category with $\bar{f_i} > 0$ and $\bar{w_i} > 0$ can also be achieved by adaptively adjusting the initial share $\bar{f_i}$ in the second category, for simplicity and easy analysis, only the category with $\bar{f_i} > 0$ and $\bar{w_i} = 0$ is considered in the simulation and experiments of this work.

Second, each task is assigned an effective power share f_i and an effective weight w_i, with the effective weight sum of all tasks fixed to one ($\sum_{\forall i \epsilon A} w_i = 1$). As the name indicates, the power share allocation is expressed with the effective power share and is finally based on the effective weight of each task. Let F denotes the total initial power share that has been reserved to the reserved class (RC) tasks, the effective power share and the effective weight are computed as follows:

$$f_i = \bar{f_i} + \left(\frac{\bar{w_i}}{\sum_{\forall j \epsilon A(t)} \bar{w_j}}\right) \cdot (1 - F),\ F = \sum_{\forall j \epsilon A(t)} \bar{f_j} \tag{3.2}$$

$$w_i = f_i \tag{3.3}$$

In a dynamic system, each time the task number changes or any task adjusts its power share requirement (by modifying the initial share $\bar{f_i}$ or the initial weight $\bar{w_i}$), equation (3.2) and (3.3) are employed to re-compute the effective weight. Specifically, for a time-sensitive task with the initial weight as zero ($\bar{w_i} = 0$), no

re-computation of the effective power share is required; the effective weight can be simply fixed to the initial power share or the desired power share by $w_i = \bar{f}_i$.

The above mechanism of power share protection is simple yet can support a wider variety of energy requirements than the mechanism in [62]. Besides, in comparison with the share protection mechanism in [65], the effective weight recalculation is significantly simplified; consequently, this mechanism is suitable for systems with arbitrary combination of batch tasks and time-sensitive tasks.

3.3.2.2 Example demonstration

To demonstrate the idea of the power share protection, an example is provided in Table 3.3 regarding to the effective weight recalculation upon the join of new tasks.

Table 3. 3: Example of Recalculating the Effective Weight for Power Share Protection

a)

Tasks	$\bar{f}_i$	$\bar{w}_i$	w_i **and** f_i
1	0.2	0	0.2
2	0.2	1	0.3
3	0	2	0.2
4	0	3	0.3

b)

Tasks	$\bar{f}_i$	$\bar{w}_i$	w_i **and** f_i
1	0.2	0	0.2
2	0.2	1	0.25
3	0	2	0.1
4	0	3	0.15
➢ **5**	0	6	0.3

c)

Tasks	$\bar{f}_i$	$\bar{w}_i$	w_i **and** f_i
1	0.2	0	0.2
2	0.2	1	0.225
3	0	2	0.05
4	0	3	0.075
5	0	6	0.15
➢ **6**	0.3	0	0.3

In Table 3.3-a), there are four active tasks in the system. Task 1 and 2 are RC tasks that require a specific guarantee of power share. Especially, task 2 can compete for unreserved or released power share with its non-zero initial weight. Task 3 and 4 are batch tasks which do not require a guaranteed power share. Because task 1 and task 2 reserve a total power share of 0.4, only a remaining power share of 0.6 is allocated to task 2, task 3, and task 4 with their initial weights being 1, 2, and 3, respectively. Then, based on the equation (3.2) and (3.3), the effective power share and the effective weight of each task are computed as follows:

$$w_1 = f_1 = 0.2 + 0 = 0.2,$$

$$w_2 = f_2 = 0.2 + \frac{1}{6} \times 0.6 = 0.3,$$

$$w_3 = f_3 = 0 + \frac{2}{6} \times 0.6 = 0.2,$$

$$w_4 = f_4 = 0 + \frac{3}{6} \times 0.6 = 0.3.$$

Later on, as shown in Table 3.3-b), a BC task (task 5) joins the energy competition with the initial power share as zero and the initial weight as 6. Then, the effective power share and the effective weight are updated as follows:

$$w_1 = f_1 = 0.2 + 0 = 0.2,$$

$$w_2 = f_2 = 0.2 + \frac{1}{12} \times 0.6 = 0.25,$$

$$w_3 = f_3 = 0 + \frac{2}{12} \times 0.6 = 0.1,$$

$$w_4 = f_4 = 0 + \frac{3}{12} \times 0.6 = 0.15,$$

$$w_5 = f_5 = 0 + \frac{6}{12} \times 0.6 = 0.3.$$

As can be seen, task 5 can only compete for the remaining power share of 0.6; the reserved power share of task 1 and task 2 are protected from the competition of task 5.

Finally, Table 3.3-c) shows the effective weight recalculation upon the join of a RC task (task 6) whose initial power share is 0.3 and initial weight is 0. The effective

power share and effective weight of each task are computed as:

$$w_1 = f_1 = 0.2 + 0 = 0.2,$$

$$w_2 = f_2 = 0.2 + \frac{1}{12} \times 0.3 = 0.225,$$

$$w_3 = f_3 = 0 + \frac{2}{12} \times 0.3 = 0.05,$$

$$w_4 = f_4 = 0 + \frac{3}{12} \times 0.3 = 0.075,$$

$$w_5 = f_5 = 0 + \frac{6}{12} \times 0.3 = 0.15,$$

$$w_6 = f_6 = 0.3 + 0 = 0.3.$$

In this case, a total power share of 0.7 is reserved for task 1, task 2, and task 6, and then, only a remaining power share of 0.3 is allocated to task 2, task 3, task 4, and task 5 in proportional to their initial weights.

3.3.3 Power share reallocation

3.3.3.1 Problem and solution

When a task finishes its work and leaves the energy competition, its power share is released and is available for reallocation among the other active tasks in the system. Ideally, the released power share should be fairly reallocated to the other active tasks according to their initial weights. However, in a real system, unfair share reallocation may occur if the temporarily released power shares of periodic time-sensitive tasks are immediately reallocated to the other active tasks. This is because, a periodic task releases its power share after the work of one period is completed and regains the power share at the beginning of its next period; when there are multiple periodic time-sensitive tasks in the system, depending on which periodic task or which set of periodic tasks have released the power share as well as which task is selected to access the CPU at which time, the remaining power share for effective weight re-computation is different. To demonstrate the unfair power share reallocation that

may occur in a real system, an example is provided in Table 3.4-a) that will be explained later in this section.

Note that, this problem of unfair power share reallocation has not yet been reported and dealt with in former share protection mechanisms [62, 54, 65]. In this dissertation work, as a solution to the problem, the total reserved power share F of RC periodic time-sensitive tasks is fixed when any RC task temporarily leaves the system after finishing the work of one period; however, the total reserved power share F will be updated if any RC task has finishes all periods of work and completely left the system. With the above solution introduced, the method of computing the effective weight and the effective power share changes slightly. This is because the sum of all effective power shares is always one ($\sum_{\forall i \epsilon A} f_i = 1$) while the sum of all effective weights is now not necessary to be one ($\sum_{\forall i \epsilon A} w_i \leq 1$). Let $\overline{F}$ denotes the total initial power share of the RC tasks that have not completely finished the work, the effective weight is directly computed as follows:

$$w_i = \bar{f}_i + \left(\frac{\bar{w}_i}{\sum_{\forall j \epsilon A(t)} \bar{w}_j}\right) \cdot \left(1 - \overline{F}\right),\ \overline{F} = \sum_{\forall j \epsilon U(t)} \bar{f}_j \tag{3.4}$$

where $U(t)$ denotes the tasks that have not completely finished all the work, it also contains periodic tasks that have temporarily left the system. And, since the sum of all effective weights is not always one any more ($\sum_{\forall i \epsilon A} w_i \leq 1$), the effective power share allocated to each task is now computed as:

$$f_i = \frac{w_i}{\sum_{\forall j \epsilon A(t)} w_j} \tag{3.5}$$

According to equation (3.4) and (3.5), since the remaining power share for allocation is constant upon the temporary releasing of power shares by RC tasks, it can support a fair and proportional energy allocation among BC tasks. Although the solution favors the remaining active RC tasks with a larger allocation of the instantaneous power share when the effective weight sum is less than one ($\sum_{\forall i \epsilon A} w_i < 1$), the maximum long-term power share that can be served to a RC periodic time-sensitive task is the

same; the over-allocated instantaneous power share will be finally re-allocated to the BC tasks after a RC task finishes its work in each period.

3.3.3.2 Example demonstration

To demonstrate the unfair power share reallocation and our solution to the problem, Table 3.4 provides an example of the power share reallocation upon the temporary releasing of the reserved power shares.

Table 3. 4: Example of Power Share Reallocation upon the Temporary Releasing of RC Power Shares

a)

Tasks	$\bar{f}_i$	$\bar{w}_i$	w_i	f_i
1	0.1	0	0.1	0.1
2	0.2	0	0.2	0.2
3	0.4	0	0.4	0.4
4	0	1	0.1	0.1
5	0	2	0.2	0.2

b)

Tasks	$\bar{f}_i$	$\bar{w}_i$	w_i	f_i
1	0.1	0	0.1	0.167
2	0.2	0	0.2	0.333
4	0	1	0.1	0.167
5	0	2	0.2	0.333

c)

Tasks	$\bar{f}_i$	$\bar{w}_i$	w_i	f_i
1	0.1	0	0.1	0.25
4	0	1	0.1	0.25
5	0	2	0.2	0.5

In Table 3.4-a), task 1, task 2 and task 3 are RC periodic time-sensitive tasks with non-zero initial power shares, while tasks 4 and task 5 are BC batch tasks with non-zero initial weights. The RC tasks reserve a total power share of 0.7, and the remaining power share of 0.3 is allocated to task 4 and task 5 in proportional to their initial weights. Therefore, when the five tasks are all active in the system, the

effective weights of task 4 and 5 are 0.1 and 0.2, respectively. Now, assume that at certain moment, task 1 finishes its work of one period, temporarily leaves the system, and releases its power share of 0.1. Then, if we immediately update the total reserved power share to 0.6 and task 4 is the next task to be executed, the effective weight of task 4 is $\frac{(1-0.6)\times 1}{3} = 0.13$. At a later moment, task 5 is executed after both task 1 and task 2 have left the system and released a total power share of 0.3. Then, after the total reserved power share is updated to 0.4, the effective weight of task 5 is $\frac{(1-0.4)\times 2}{3} = 0.4$. As can be seen, since the effective weight ratio of task 4 and task 5 is no longer in accordance with their initial weight ratio of 1:2, they will fail to receive the energy fairly and proportionally in the long-term.

Table 3.4-b) and 3.4-c) show the power share reallocation results based on our solution that is formulated in equation (3.4) and (3.5). In table 3.4-b), after task 3 has temporarily left the system, the total reserved power share $\overline{F}$ is fixed as 0.7, and according to equation (3.4), the effective weights of the rest active tasks are the same as the original ones in Table 3.4-a). However, since the effective weight sum of all active tasks is no longer one, the effective power shares of the active tasks change proportionally to their effective weights, and the values are computed based on equation (3.5). As can be seen in Table 3.4-b), task 4 and task 5 receive power shares that are proportional to their initial weights, and an instantaneous power share that is larger than the reserved one is respectively allocated to task 1 and task 2. If the new instantaneous power share is larger than the maximum long-term power share, task 1 and task 2 will finish their work before the end of each period and release the overly-allocated instantaneous power share to task 4 and task 5 that are continuously active in the system. To see how that works, we move to Table 3.4-c), in which task 2 has finished its work of one period and temporarily left the system, leaving only task 1, task 4, and task 5 still active in energy competition. We keep fixing the total reserved power share $\overline{F}$ as 0.7, and compute the effective weights and effective power shares of all active tasks based on equation (3.4) and (3.5).

As can be observed, task 4 and task 5 keep receiving their power shares in proportional to the initial weight ratio of 1:2. If task 1 temporarily leaves the energy computation in a later moment, its instantaneous power share of 0.25 will also be reallocated to task 4 and 5 in proportional to their initial weights. Actually, in the long-term, if the maximum long-term power shares of task 1, task 2, and task 3 are P_{max}^{1}, P_{max}^{2} and P_{max}^{3} , respectively, then, the long-term power shares of task 4 and task 5 are $\frac{1}{3} \times (1 - \sum_{i=1}^{3} P_{max}^{i})$ and $\frac{2}{3} \times (1 - \sum_{i=1}^{3} P_{max}^{i})$, respectively. Therefore, our solution avoids the unfair power share reallocation among BC batch tasks while in the meantime favors the time-sensitive tasks with temporary higher instantaneous power shares.

3.4 Energy-based fair queuing (EFQ)

This section presents the energy-based fair queuing (EFQ) algorithm design for energy-centric processor scheduling. It starts with a challenge analysis of developing EFQ algorithms based on the traditional fair queuing algorithms in network and CPU scheduling, and then, proposes a practical and low-time-complexity EFQ algorithm named starting-energy fair queuing (SEFQ). After that, the issue of time-constraint compliance under general EFQ and SEFQ scheduling is analyzed and the requirement of combining time-friendly mechanisms into EFQ scheduling is emphasized. Finally, the borrowed starting-energy fair queueing (BSEFQ) is proposed to support real-time and multimedia scheduling in EFQ with the combination of a real-time friendly mechanism.

3.4.1 Challenges of developing energy-based fair queuing

Fair queuing has been proven as the proper candidate algorithm for achieving proportional resource sharing among competing entities. To achieve a proportional share of the energy and system power, it is natural to combine the fair queuing algorithm with our energy-centric scheduling model so that to develop an

energy-based fair queuing (EFQ) algorithm that schedule tasks according to the energy consumption as well as the effective weight of each task. However, extending the fair queuing to the energy domain is not an easy work; several issues can arise if an existing fair queuing algorithm of network or CPU scheduling is directly applied to energy-centric scheduling. We will discuss the challenges of energy-based fair queuing from three aspects.

The first challenge of developing energy-based fair queuing is the unpredictability of the energy packet size. Unlike the fair queuing network scheduling in which the data packet size can be known upon arrival, the energy packet size of a service quantum can only be known after the service quantum is completely served. Making it worse, the energy packet size is hard to predict considering the fact that it can vary with different tasks and within the same task it can also vary depending on which piece of code is executed, especially in the case of multimedia applications. Even if energy prediction mechanisms are available, how the prediction accuracy can affect the performance of these algorithms is another open problem. Therefore, fair queuing network scheduling algorithms which rely on a prior knowledge of the packet size, such as WFQ [57], WF^2Q [71], and SCFQ [59], are not applicable to energy-centric scheduling. On the other side, time-quantum-based fair queuing CPU scheduling algorithms like the stride scheduling [69] and EEVDF [55] do not require the length and size of service quanta to be known in advance. However, these algorithms can incur a lot of extra overhead, because the finishing tag has to be recomputed to reflect the actual size of a service quantum after its completion.

The second challenge is the volatility of the system power. While the network or CPU bandwidth in most traditional fair queuing algorithms is assumed to be constant for allocation, energy-based fair queuing has to deal with a system power that is variable over the time. It is because the system power is dependent on the tasks being executed at specific times. Therefore, those fair queuing algorithms that update the system virtual time with the assumption of a constant commonly-shared bandwidth, such as stride scheduling [69], EEVDF [55], and SMART [40], may fail to provide the claimed fairness and delay bound when applied to energy-centric scheduling.

The third challenge is the time complexity of the scheduling algorithm. A practical energy-based fair queuing algorithm should avoid the introduction of high computing-time complexity. Therefore, traditional fair queuing algorithms that require a simultaneous emulation of the ideal fluid-flow system are not suitable for energy-based fair queuing scheduling.

3.4.2 Starting-energy fair queueing (SEFQ)

The above challenges have posed many restrictions on the development of energy-based fair queuing. These restrictions can greatly narrow down the possible methods of designing fair queuing algorithms in the energy domain in comparison to the domains of network and CPU scheduling. With the above challenges and restrictions in mind, we consider the starting-time fair queuing (SFQ) [64, 68] as the proper candidate algorithm for energy-centric scheduling. Based on SFQ, an energy-based fair queuing scheduling algorithm called starting-energy fair queuing (SEFQ) is developed by combining the SFQ algorithm with our energy-centric scheduling model.

The starting-energy fair queuing (SEFQ) algorithm is designed to schedule tasks in the increasing order of the starting energy tag (or simply starting tag) S_i, a variable that traces the normalized energy consumption of each task. To compute the starting tag S_i, a time function named virtual energy is defined to track both the received and missed normalized energy of each task, similar to the concept of the virtual time in traditional fair queuing algorithms. This is realized by defining a task virtual energy $VE_i(t)$ to track the normalized energy received by each task, and a system virtual energy $VE(t)$ to track the normalized energy consumed in the system. The system virtual energy $VE(t)$ works as a reference to update the task virtual energy $VE_i(t)$ whenever a new task joins the system for the first time or an old task rejoins the system after leaving temporarily. To avoid incurring the complexity of simulating the fluid-flow model, the system virtual energy $VE(t)$ traces the lowest starting tag among all active tasks, and therefore, is defined as being equal to the starting tag of

the task currently in service. Then, the starting tag, S_i^k, of the k^{th} service quantum sq_i^k, can be defined as the value of the task virtual energy $VE_i(t)$ at the instant of time when the service quantum sq_i^k of task T_i begins the execution. The starting tag S_i^k can be computed as follows:

$$S_i^k = max\{VE(AR(sq_i^k)), F_i^{k-1}\} \tag{3.6}$$

where $AR(sq_i^k)$ denotes the time at which the service quantum sq_i^k is requested, and F_i^{k-1} is the finishing tag of the previous service quantum sq_i^{k-1}. The finishing tag F_i^k is incremented as follows:

$$F_i^k = S_i^k + \frac{e_i^k}{w_{i,}}, F_i^0 = 0 \tag{3.7}$$

where e_i^k is the energy packet size of the service quantum sq_i^k.

As the SEFQ algorithm is an extension of the SFQ algorithm in the energy domain, it inherits from SFQ the following properties that are highly valued in energy-centric scheduling [64, 68]:

1. Low time complexity. SEFQ is computationally efficient with time complexity of O(1) for starting tag computation and O(log N) for service quantum selection. In SEFQ, since the system virtual energy is updated by self-referring to the starting tags, there is no need to simultaneously run the ideal fluid-flow model for continuously recalculating the system virtual energy, which will lead to a computing time complexity of O(N).

2. No prior knowledge of the service quanta required. Since SEFQ makes the scheduling decisions based on the starting energy tags, both the time-quantum length and the energy packet size of the service quanta are not required to be known in prior. This property is highly appreciated in multimedia task scheduling, in which the energy loads may vary dramatically over periods and are hard to predict.

3. Near-optimal fairness bound under variable system power. Like SFQ, SEFQ can achieve a fairness bound that is near optimal in comparison to WFQ, regardless of the variation of the system power. The near-optimal fairness bound of SEFQ ensures a fair and proportional sharing of the system power among the candidate tasks.

As can be seen, SEFQ is able to achieve near-optimal fairness bound under variable system power with low computing time and implementation complexity. However, as in the SFQ algorithm, the energy allocation error bound under SEFQ can increase with the growth of the number of active tasks due to the failure to meet the minimum slope property of system virtual energy updating. The increasing energy allocation error will cause the growth of the maximum dispatch latency (or delay bound) as well, making the time-constraint compliance of latency-sensitive tasks a difficult work under SEFQ. A detailed analysis of the time-constraint compliance problem under the energy-based fair queuing is provided in the next section.

3.4.3 Time-constraint compliance under EFQ

Under the energy-based fair queuing (EFQ) scheduling, meeting the time constraints of periodic time-sensitive tasks is equivalent to meeting the energy demand in each of the periods. In many periodic time-sensitive tasks, because the energy load varies in different periods, the minimum instantaneous power share that is required for meeting the energy demand in each period is also variable. In traditional fair queuing CPU scheduling algorithms, it has been proposed to dynamically adapt the share to deal with the variable workload. Unfortunately, there are several practical issues to apply adaptive power shares in the energy-based fair queuing. First, adaptive power share relies on a knowledge of the historical energy demands of previous periods to predictively compute the proper power share for the coming periods, therefore, the time-constraint compliance is sensitive to the energy load prediction accuracy. Since energy load prediction is a much more difficult work than CPU workload prediction,

the prediction overhead may be very large and the prediction accuracy is hard to guarantee. Second, unlike fair queuing CPU scheduling under which the proper bandwidth share can be computed simply based on the predicted CPU workload, the proper power share under EFQ is more complex to compute because it depends not only on the energy load of the concerned task but also on the energy loads and average powers of other active tasks in the system. The issues from the above two aspects have prevented the applyment of adaptive power shares in the energy-based fair queuing.

One obvious method to guarantee time-constraint compliance under EFQ is to assign each periodic time-sensitive task an instantaneous power share that is no lower than the worst-case power share of the task. However, as in the fair queuing CPU scheduling domain, this method can lead to an over-reservation of power share because the actual power share required by one periodic time-sensitive task is usually lower than the worst-case power share. The over-reservation causes a waste of the power share in the sense that the over-reserved power share is no more reservable for other time-sensitive tasks. In periodic time-sensitive tasks whose energy load is fluctuating slightly or whose worst-case power share is a small percentage, the degree of power share over-reservation is still acceptable under EFQ. However, the power share for reservation will be badly wasted if the task energy load is fluctuating significantly and the worst-case power share is a considerable large percentage, which is very common in many soft real-time applications such as multimedia applications.

Furthermore, even if the degree of power share over-reservation is acceptable, determine a proper instantaneous power share for time-constraint compliance is a difficult work. The difficulties mainly come from two aspects. First, since the worst-case power share of a periodic task may change frequently over different long-term static intervals in a dynamic system, the instantaneous power share reserved for a periodic time-sensitive task have to be adjusted timely and frequently in accordance to the variable worst-case power share. This brings challenge to the time-constraint compliance and increases the computing overhead on the EFQ scheduler. If the instantaneous power share reserved for each time-sensitive task is not

timely and correctly adjusted in accordance to the current worst-case power share, the time-sensitive tasks will fail to timely receive the periodic energy demands and begin to miss the deadlines. Second, although, theoretically, a worst-case power share can guarantee the compliance of all time constraints, in real systems, an instantaneous power share that is higher than the worst-case power share is required considering the energy allocation error caused by service time and energy quantization. Unfortunately, the energy allocation error bound under SEFQ is not a constant value but can increase with the growth of the active task number; without knowing the average system power, the system power burstiness, and the active task number, which are randomly variable over time, the energy allocation error bound is impossible to compute, and consequently, the instantaneous power share that is minimally required to meet the worst-case energy load in a real system cannot be determined. This means, in real system scheduling, a coarse and conservative power share that may be much larger than the worst-case power share has to be reserved to each periodic time-sensitive task for ensuring the time-constraint compliance.

In fair queuing scheduling, maintaining a strict fairness is a double-edged sword for supporting time-sensitive tasks. On one side, achieving the optimal fairness bound and allocation error bound can minimize the share over-reservation for time-sensitive tasks, and therefore, conserve the share reservation space for the support of more time-sensitive tasks. On the other side, to enforce a strict fairness among the tasks, the scheduler may preclude a time-sensitive task from using the CPU at an inopportune time, leading to a close deadline being missed. Unfortunately, in SEFQ, the near-optimal fairness bound does not guarantee a stable allocation error that is unaffected by the number of active tasks. As have been mentioned before, because of the energy allocation error bound that can increase with the task number, the minimum instantaneous power share that is required by each time-sensitive task to meet the time constraints cannot be determined under SEFQ. Making it worse, the energy allocation error of SEFQ causes a delay bound that can increase with the task number; this will amplify the negative effect of a strict fairness on the time-constraint compliance as the active task number increases in the system. A time-sensitive task

may be continuously precluded from accessing the CPU for a long period of time due to a large number of tasks waiting ahead to fairly receive their share of the energy.

Considering the above facts, maintaining a near-optimal fairness bound becomes less attractive in supporting time-sensitive tasks in energy-based fair queuing scheduling. Instead, as in the fair queuing CPU scheduling domain, certain real-time friendly mechanism can be combined into the SEFQ algorithm for a better support of time-sensitive tasks. A real-time friendly mechanism breaks the short-term fairness between time-sensitive tasks and batch tasks by giving dispatch preference to the former; however, the long-term proportional power sharing should still be maintained among tasks so that any energy starvation on batch tasks can be avoided. This point will be extended in the next section.

3.4.4 Borrowed starting-energy fair queuing (BSEFQ)

To provide a better support for time-sensitive tasks in energy-based fair queuing (EFQ), this work combines one low-complexity and low-overhead real-time friendly mechanism named warping into the SEFQ algorithm and, proposes a new EFQ algorithm named borrowed starting-energy fair queuing (BSEFQ). The warping mechanism was originally proposed in the borrowed-virtual-time (BVT) scheduling [74], it has been adapted to support priority-based real-time scheduling in energy-based fair queuing. The adaptations will be introduced along with the presentation of the BSEFQ algorithm.

The warping mechanism introduces a new variable called effective starting tag for each task, the BSEFQ traces the normalized energy consumption of each task with the starting tag S_i but schedules the tasks in the increasing order of the effective starting tag, ES_i. For distinction with the effective starting tag ES_i, the original starting tag S_i is called the actual starting tag from this point forward. The effective starting tag ES_i is computed in such a way that, for a periodic time-sensitive task within a certain time limit, it is the actual starting tag S_i minus a certain value named warp, and the task is regarded as warped; for a batch task or a time-sensitive task that

is out of the time limit, it equals to the value of the actual starting tag S_i, and the task is regarded as un-warped.

The time limit imposed on periodic time-sensitive tasks is called the warp time limit. This limit is applied on a periodic basis to restrict the maximum length of time that one task can run warped in each period. One periodic time-sensitive task is allowed to run warped at the beginning of each period; it can run warped continuously until the cumulative warp time of the current period reaches the warp time limit; after that, the time-sensitive task is forced to run un-warped until the next period begins. Note that, in comparison to the warp mechanism in BVT, where an additional parameter is required to specify the minimum time that one task has to wait before warping again, the way how the warp time limit works is simplified and easier to control in this work, especially in the situation when multiple periodic time-sensitive tasks are simultaneously running with warp.

By warping back the starting tag and borrowing virtual energy from its future energy allocation, a time-sensitive task moves forward in the run queue and receives its share of energy earlier to meet its time constraints [74]. The short-term fairness among tasks is broken with the staring tag warping to give time-limited dispatch preference to time-sensitive tasks; however, the long-term proportional power sharing is still maintained by updating the actual starting tag based on the assigned effective weights. If a time-sensitive task becomes un-warped before finishing the work of one period, since its actual starting tag S_i has been advanced with the normalized energy consumption during the warped period of time, the task will be placed at the very end of the run queue to let other tasks have the chance to catch up with the energy consumption.

In BSEFQ, the warping mechanism brings a problem to the updating of the system virtual energy that is not reported in the BVT algorithm [74]. Recall that, in SEFQ, tasks are scheduled in the increasing order of the actual starting tag S_i and the current task in execution is the one with the smallest actual starting tag. Thus, the system virtual energy $VE(t)$ is updated to the actual starting tag of the task currently in execution to trace the smallest starting tag among all active tasks. This guarantees

the system virtual energy to be a monotonically non-decreasing function that is no greater than the minimum virtual energy of all active tasks. However, under BSEFQ, once a time-sensitive task is warped, it will be executed even if its actual starting tag S_i is not the smallest one. If the system virtual energy $VE(t)$ is still updated to the actual starting tag S_i of the current task in service, it is no longer a non-decreasing function that always keeps pace with the smallest actual starting tag. In this case, when a new batch task joins the system at time τ and is assigned an actual starting tag that equals the system virtual energy $V(\tau)$, the task will be delayed in execution if the system virtual energy $V(\tau)$ is larger than the lowest actual starting tag in the system. The delay time can be considerably large if there is a big difference between the system virtual energy and the lowest actual starting tag of all active tasks.

To resolve the above problem, BSEFQ only updates the system virtual energy to the actual starting tag of the current task in service if the current task is not warped. Otherwise, the system virtual energy remains unchanged. With this approach, the system virtual energy $VE(t)$ of BSEFQ is guaranteed to be a monotonically non-decreasing function that traces the lowest virtual energy of all active tasks.

The scenario of multiple time-sensitive tasks with multiple levels of warp values is also not investigated in the BVT algorithm [74]. When there are multiple periodic time-sensitive tasks active in the system, different levels of warp values can be either statically or dynamically assigned to the tasks according to their periods, user preferences, or urgencies. Statically, warp values can be assigned in a rate-monotonic (RM) manner with which a larger warp value is given to the task with shorter period; while dynamically, warp values can be assigned in an earliest-deadline-first (EDF) manner with which a larger warp value is assigned to the task with the most urgent deadline. For simplicity and to focus on GPOS in which a certain share of CPU bandwidth should be left to batch tasks, this work focuses on static warp values. Besides of assigning warp values in the RM manner, more important, user-preferred, or latency-sensitive tasks are usually given larger warp values so that they can be dispatched earlier to meet their time constraints. For instance, a time-critical system task or a hard real-time task can be assigned the largest warp value, and a soft

real-time task or an interactive task can be assigned a smaller warp value. The task that holds the largest warp value is dispatched immediately after its new period begins and can continuously occupy the CPU until finishing its work or reaching its warp time limit, while tasks holding smaller warp values have to wait until more important tasks finish their work or become un-warped. For time-sensitive tasks of equal importance, the same warp values are assigned so that the tasks can be scheduled in turns; the dispatching frequency of each task depends on the energy packet size and the effective weight, and their real-time performance can be traded off by adjusting the effective weights.

With the warp values of different levels, the warping mechanism actually combines the priority-based real-time scheduling into the energy-based fair queuing scheduling. While the warp-based priorities enable a flexible and effective support of various types of time-sensitive tasks in energy-based fair queuing (EFQ), the warp time limit of each time-sensitive task restricts the maximum time that one task can run with priority and avoids any energy starvation on batch tasks. By adjusting the warp time limit automatically based on the CPU workload feedback or manually through a user interface, BSEFQ allows for a flexible tradeoff between power control and the time-constraint compliance of time-sensitive tasks. Therefore, on one hand, power pulses caused by high-priority tasks with excessive energy demand can be restricted to avoid interfering with the normal energy use of other tasks; on the other hand, stringent time-constraint compliance can be achieved when tasks are consuming energy at a normal rate. As in the traditional real-time processor scheduling, admission control can be applied in combination with the warping mechanisms to control the CPU utilization rate under a safe threshold to keep all admitted time-sensitive tasks schedulable in EFQ scheduling.

3.5 Summary

In this chapter, the traditional fair queuing algorithm is extended to the energy sharing domain, and the energy-based fair queuing (EFQ) scheduling algorithm is proposed for energy-centric processor scheduling to provide a support on proportional power sharing, time-constraint compliance and a tradeoff between them.

Applying the fair queuing algorithm for energy-centric processor scheduling is equivalent to implementing the ideal Generalized Processor Sharing (GPS) model in the energy sharing domain. Therefore, the first step of developing energy-based fair queuing (EFQ) algorithms is to build a realistic energy-centric scheduling model in reference to the GPS model. Considering that an accurate modelling of the asynchronous energy consumption caused by hardware devices is almost impossible and high-level scheduling policies do not concern in which device a specific amount of energy is consumed, this chapter introduces a practical energy model in which the energy consumption is assumed to be synchronously related to the CPU time quantum. Thus, in the energy model, each task that receives one CPU time quantum will be served a corresponding amount of energy that is defined as energy packet.

In comparison with the CPU bandwidth management in the fair queuing processor scheduling model, the power share management in the energy-centric scheduling model is a much more complex work. Therefore, this chapter provides an insight into the power share management under the energy-centric scheduling. The maximum long-term power share is defined to describe the upper bound of the average power share that one periodic time-sensitive task can receive in the long-term, and the worst-case power share is defined to describe the minimum theoretical power share that is required to meet the worst-case energy demanding of one periodic time-sensitive task. It is found that the maximum long-term power share and the worst-case power share are sensitive to the system dynamics because their values are also dependent on the energy loads of other active tasks. To ensure an acceptable and stable performance, the power share of one time-sensitive task should be no less than the maximum long-term power share and its instantaneous value needs to be protected

from the competition of other tasks. Upon the change of the scheduling environment, one simple yet effective way to achieve the power share protection is to fix the weight of one time-sensitive task to its desired power share and re-compute the weight of regular tasks based on their initial weights and the overall reserved (or protected) power share of the time-sensitive tasks. Because the instantaneous power share of one periodic time-sensitive task is usually over-reserved in comparison with its maximum long-term power share, the over-reserved power share should be reallocated to the other tasks when the time-sensitive task temporarily leaves the energy competition. To avoid the occurrence of unfair power share reallocation upon the temporary releasing of power shares, the overall reserved power share of time-sensitive tasks can be fixed as a constant value for the task weight re-computation of other regular tasks.

The design of energy-based fair queuing (EFQ) algorithms is restricted by the unpredictability of the energy packet size, the volatility of the system power and the practical issues such as the implementation complexity and the scheduling overhead. The proposed starting-energy fair queuing (SEFQ) algorithm schedules tasks in the increasing order of the starting energy tag, thus, the size of the energy packet as well as the length of the CPU time quantum are not required to be known in advance. In addition, because the system virtual energy of SEFQ can be updated by self-referring to the lowest starting energy tag of all tasks, no simulation of the ideal GPS model and no assumption of a constant system power are required for the functioning of SEFQ. Therefore, SEFQ is simple to implement and it can achieve near-optimal fairness bound for proportional power sharing under variable system power with low computing-time complexity. Unfortunately, because of the failure to meet the minimum slope property in the system virtual energy updating, the energy allocation error and the scheduling latency bound under SEFQ can increase with the growth of the task number. This increases the difficulty of determining a proper power share for the time-constraint compliance of time-sensitive tasks, and a conservative power share should therefore be overly reserved to each time-sensitive task. Especially, when the number of tasks is large, the effort of maintaining a fair sharing of the energy will lead to a considerable large scheduling latency to time-sensitive tasks and

make the time-constraint compliance a difficult work under SEFQ. To improve the support of real-time and multimedia scheduling under EFQ, a real-time friendly mechanism named warping is combined into the SEFQ, and based on that, the borrowed starting-energy fair queuing (BSEFQ) is proposed. The warping mechanism can periodically reduce the starting energy tag of one time-sensitive task to allow it being scheduled earlier to receive its share of energy, but the maximum time that one task can be warped in each period is restricted. This gives time-limited priority to the time-sensitive task and improves its scheduling latency; although the short-term fairness among tasks is broken, the long-term proportional power sharing can still be maintained by advancing the starting energy tag based on the task weight. When multiple time-sensitive tasks are active in the system, different levels of warp values can be employed to prioritize the time-sensitive tasks based on their importance or time urgency. The warping mechanism actually combines the priority scheduling into the proportional share scheduling, which allows the BSEFQ being able achieve proportional power sharing, effective time-constraint compliance, and a flexible tradeoff between them.

Chapter 4

High-Level Modelling and Simulation

Before moving to the complex implementation of the energy-based fair queuing (EFQ) scheduling algorithm on a concrete OS, a high-level modelling and simulation of the scheduling algorithm can provide a convenient and flexible pre-evaluation of the scheduling behavior. In this chapter, a high-level EFQ modelling and simulation based on the SystemC language is provided for early-phase assessment of the EFQ algorithm. The chapter starts with a brief introduction on the SystemC modelling language, and then continues with a high-level modelling of the EFQ algorithm in SystemC. After that, a simulation test-bench is developed and a task set is characterized to execute EFQ simulation experiments. Finally, the simulation results are analyzed according to the different EFQ properties.

4.1 SystemC

SystemC [76] is a system-level modelling language that is widely used for hardware and software co-design. It is a C++ class library which provides an event-driven simulation interface. Unlike the C and C++ languages which have no notion of time and concurrency, SystemC can simulate concurrent processes and support a cycle-accurate model for software algorithms.

Modules are the basic building blocks in SystemC design. A SystemC model usually consists of a number of modules that are connected via channels. Within the module, there are a variety of elements, including processes, ports, internal signals and data instances, etc. Processes are the basic computation elements which can be executed currently by the SystemC simulator. A process should be registered after its definition. The simplest registration method is *SC_THREAD*. The underlying functions and codes of a *SC_THREAD* process are executed only once. SystemC provides the *sc_time* data type to measure the process execution time. Particularly, the

wait(sc_time) method can be employed to suspend the execution of *SC_THREAD* processes for specified periods of time. Ports are in charge of the communication between modules. They use specified type of interfaces to communicate with channels. While interfaces define a set of synchronization and communication mechanisms, such as mutex, semaphore and FIFO, channels implement the interfaces. Channels in SystemC can be viewed as a special type of modules that implement the communication mechanisms. The readers are advised to refer the book [77] for more details on SystemC facilities and their usage.

In this thesis work, the SystemC development environment is set up on the Eclipse (Juno) IDE for C/C++ Developers. The codes are compiled using the GCC 4.6.3 compiler with the support of the SystemC 2.2.0 library.

4.2 EFQ modelling in SystemC

4.2.1 High-level abstraction

This section describes the high-level modelling of the EFQ scheduling algorithm in SystemC. The CPU scheduling abstraction that is referred for EFQ modelling is shown in Figure 4.1.

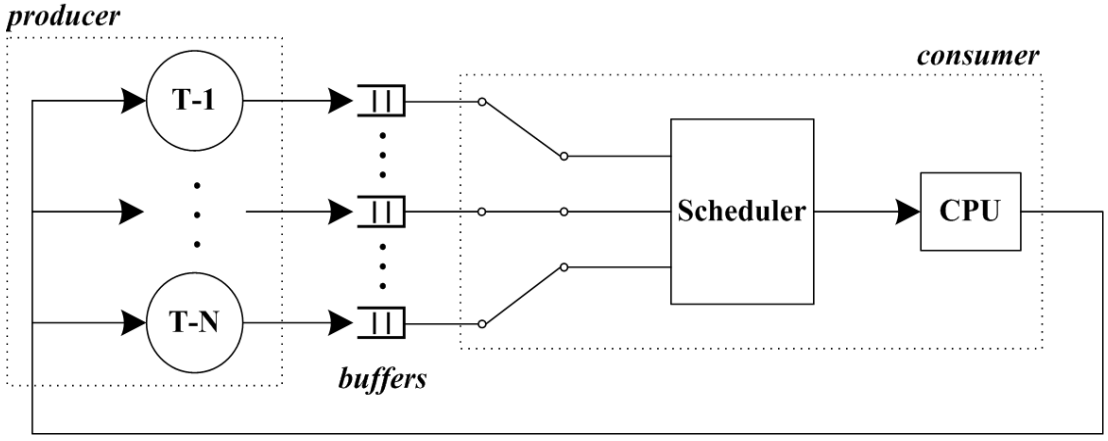


Figure 4. 1: High-level Abstraction of the CPU Scheduling

To reduce the simulation complexity, the run queue that uniformly manages all active tasks is not exclusively modelled in the abstraction. As an alternative, each task

is correlated to a data buffer which records its resource requests in the form of energy packet and the task queuing operations of the run queue are combined into the scheduler model. Therefore, it appears that the tasks are working as a producer that generates energy packet requests, and the scheduler and CPU are working as a consumer that consumes the requested energy packets in the data buffers. In the remaining of this section, we will focus on the modelling of the EFQ scheduler and the CPU. The modelling of the tasks and their request buffers will be later introduced when the simulation test-bench is presented.

4.2.2 EFQ modelling: the consumer

In this simulation design, the EFQ scheduler and the CPU have been modelled together in one SystemC module, which is known as the consumer module. The flow chart of the SystemC-based modelling of the EFQ algorithm is shown in Figure 4.2. Note that the EFQ algorithm has been modelled with the simulation control and results generation in mind. Therefore, some extra functions have been added in the consumer module for this purpose. Also, only the flow chart of BSEFQ is provided in this work because SEFQ can be seen as a special case of EFQ in which all tasks are un-warped.

To model the EFQ scheduler and the CPU, the length of the maximum schedulable time quantum Q as well as the scheduling time tick must be determined in advance. For clarity and without loss of generality, the CPU time unit (Tu) is employed to measure the time in the simulation, one Tu is defined as one millisecond (MS) in the *sc_time* data type of SystemC and is set as the minimum schedulable time quantum for the simulation. For simplicity, all time quanta requested by the tasks are assumed to have the maximum length Q, which is also set as 1 Tu. That is to say, in each scheduling tick of the simulation, a time quantum of 1 Tu is assigned to the selected task and a standard energy packet of the task is consumed. The length of the scheduling tick is thus set as 1 CPU time unit as well. To achieve the 1 Tu scheduling tick and simulate the CPU execution, we simply employ the *wait(sc_time)* method to suspend the consumer simulation for 1 Tu at certain point inside the scheduling loop.

Within each scheduling loop, it is firstly checked whether there is energy available for consumption. This is achieved by updating the total energy consumption at the end of each scheduling loop and comparing its value with the pre-set energy budget. If the energy is exhausted, the simulation program will output the scheduling results in text files and quit the execution; otherwise, we further check whether all the tasks under test are idling or not. If all the tasks have been continuously idled for a certain time that is larger than the biggest period of periodic tasks, the scheduling results are exported and the simulation is terminated; otherwise, we move to the initialization of the total initial weight and total reserved share. Note that a task is regarded as idle only if its corresponding request buffer is empty. To detect that, the scheduler reads a request buffer and returns the remaining number of time quanta requests in it; if the returned value is zero, the request buffer is empty and its corresponding task is regarded as idle.

The EFQ algorithm schedules tasks in the increasing order of the effective starting (energy) tag. It is thus important to determine the task with the lowest effective starting tag in the scheduling loop. To achieve this goal, first, a task needs to be set by default as the one for dispatching (Figure 4.2-①), and then, a tag comparison loop (Figure 4.2-②) is executed to compare the effective starting tag of all candidate tasks and select the one with the lowest tag for dispatching. To support this procedure, one variable named $minTag$ is exclusively defined to store the ID of the task with the smallest effective starting energy tag. Initially, the ID of the default dispatching task is assigned to the variable $minTag$.

The method of setting the default dispatching task determines the way how the scheduler breaks the tie when two tasks have equal energy tag. Because a tie breaking that favors time-sensitive tasks can support a better real-time performance without affecting the fairness, it first select the default dispatching task randomly from the time-sensitive tasks; a non-real-time task is randomly selected only in case all time-sensitive tasks are idling. Note that, while it has no time cost to execute these operations in a simulation, the overhead of implementing them in a real system can be very high if the algorithm is not properly designed.

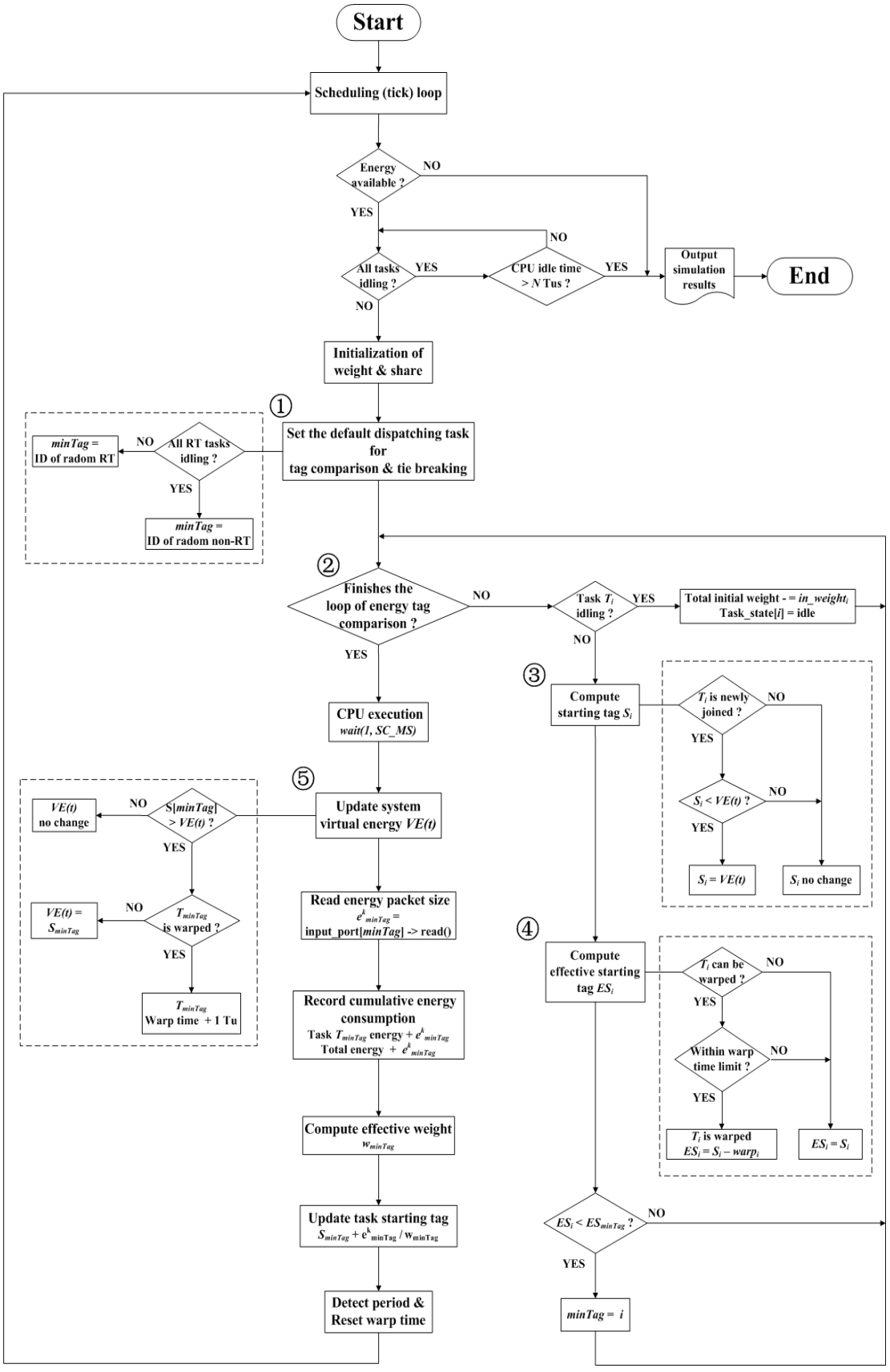


Figure 4. 2: Flow Chart of SystemC-based EFQ Modelling in the Consumer Module

After the default dispatching task is randomly determined, the tag comparison loop (Figure 4.2-②) is executed to compare the effective starting tag of all candidate tasks. The purpose is to select the task with the lowest effective energy tag and assign the task ID to the variable $minTag$. Within the loop, it first checks the request buffer of one task to find out whether the task is idling or not. Once an idle task is detected, the initial weight value of the idle task is subtracted from the total initial weight and the task state is marked as idle. For an active task T_i, it will further compute its starting energy tag S_i (Figure 4.2-③). First, it needs to detect whether the task is a newly joined one that has just made its transition from idle state to active state. If a newly joined task is detected and its starting energy tag S_i is smaller than current value of the system virtual energy $VE(t)$, the starting energy tag S_i is updated to the value of the system virtual energy $VE(t)$. Otherwise, the starting energy tag S_i keeps unchanged.

Next, it further computes the effective starting energy tag ES_i based on the warp parameters (Figure 4.2-④). For a warp-enabled task that is within its warp time limit, the effective starting energy tag ES_i is computed by subtracting its warp value $warp_i$ from the starting energy tag S_i. In other cases, the effective starting energy tag ES_i equals the starting energy tag S_i. Based on the effective starting energy tag ES_i, the current task T_i is compared with the task whose ID is stored in the variable $minTag$. If the current task T_i has a lower effective starting energy tag ($ES_i < ES_{minTag}$), its ID is assigned to the variable $minTag$; otherwise, the $minTag$ value keeps unchanged. After the above works are finished, the tag comparison loop will be executed another time for the next candidate task. When the loop finishes the tag comparison for all candidate tasks, the task with the smallest effective starting energy tag is determined and its ID is stored in the variable $minTag$.

Once the task to dispatch has been selected, the *wait(sc_time)* function is called to stop the consumer for one CPU time unit to simulate the task's execution on the CPU. After the CPU execution, the system virtual energy $VE(t)$ has to be firstly updated (Figure 4.2-⑤) before the update of other variables. Because the system virtual

energy $VE(t)$ is non-decreasing and always traces the smallest starting energy tag of all the tasks, it is updated to the starting energy tag of the current task T_{minTag} only in case that the task T_{minTag} is not warped and its starting energy tag S_{minTag} has a larger value. In other cases, the system virtual energy $VE(t)$ keeps unchanged. And in case the current task T_{minTag} is warped, its cumulative warp time is increased by one CPU time unit.

After the system virtual energy is updated, we need to further update the cumulative energy consumptions and the starting energy tag of the current task T_{minTag}. To achieve that, we need first read the size of the selected energy packet e_{minTag}^{k} from the request buffer of task T_{minTag}. Then, both the cumulative energy consumption of task T_{minTag} and the total energy consumption are increased by adding the size of the selected energy packet e_{minTag}^{k}. While the cumulative energy consumption of each task is recorded for energy statistics and scheduling results analysis, the total energy consumption is used to check if the energy budget is reached at the beginning of each scheduling loop. To update the starting energy tag of task T_{minTag}, we also need to compute in advance its effective weight w_{minTag} according to the share protection mechanism. Finally, after the size of the energy packet e_{minTag}^{k} and the effective weight w_{minTag} have been both obtained, the starting energy tag S_{minTag} is updated by increasing the current starting energy tag with the normalized energy consumption, $\frac{e_{minTag}^{k}}{w_{minTag}}$.

The warp mechanism should reset the cumulative warp time of a periodic task at the start of each period. At the end of the scheduling loop, the *sc_time_stamp( )* function is called to obtain the current time of the simulation and use it to detect whether a new period of any periodic task is started. If a new period is detected, the cumulative warp time of the task is reset to zero so that the task can be warped again for the coming period.

4.3 Simulation test-bench design

In this section, the simulation test-bench developed to assess the SystemC-based EFQ design is described. It starts with an overview of the test-bench architecture, and continues with the SystemC modelling of the simulation task set. Finally, the extra functions to generate scheduling results are presented.

4.3.1 Test-bench architecture

Based on the CPU scheduling abstraction and the producer-consumer model in Figure 4.1, a simulation test-bench has been developed. Its architecture is shown in Figure 4.3. There are three SystemC modules in general: the producer, the consumer and the FIFO channel. Each module is defined in a corresponding *C++* header file. The *main.cpp* file is in charge of setting simulation parameters (task number, initial shares and weights, task period and workload etc.,), instantiating the modules and controlling the test-bench execution.

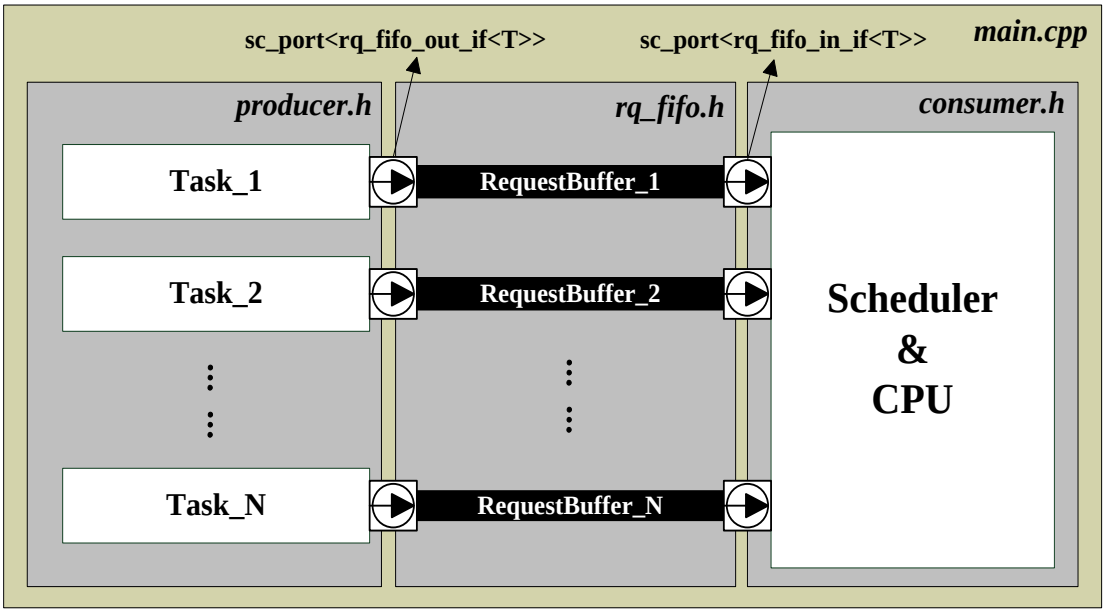


Figure 4. 3: Architecture of the SystemC-based Simulation Test-bench

In the simulation test-bench, one task is modelled as an instantiation of the producer module; depending on the task type, it either periodically or continuously

generates service requests in the form of energy packet. The request buffers are modelled as instantiations of the FIFO channel with sufficient length; the energy requests generated by the producer module are stored into the corresponding request buffers through the entitled ports. Then, buffering the energy packets is as easy as writing the size of each energy packet into the request buffers. This writing is fulfilled through the *write( )* function, which is declared in an interface class named *rq_fifo_out_if* and defined in the header file *rq_fifo.h*. The scheduler and CPU are modelled as an instantiation of the consumer module, which is also connected to the request buffers via the entitled ports. Based on the EFQ scheduling algorithm, the consumer module selects the next energy packet to consume from the candidate request buffers. Similarly, it is fulfilled by reading the size of a selected energy packet through the *read( )* function, which is declared in an interface class named *rq_fifo_in_if* and defined in the header file *rq_fifo.h*. Finally, in the *main.cpp*, the modules of producer, consumer, and FIFO channel are instantiated and connected to form a complete simulation test-bench.

4.3.2 Task modelling: the producer

To drive the EFQ scheduling simulation in the consumer module of the test-bench, the tasks should be properly modelled in the producer module to generate a variety of different energy requesting patterns. The flow chart of the producer module is shown in Figure 4.4.

At the beginning of the producer, it is firstly checked whether the task is set to be delayed in the starting up. This is to simulate the situation in which a new task joins the resource competition at certain instant of time; in case of that, the *wait(sc_time)* function is called to delay the task for a certain length of time before it enters the energy requesting loop. Then, once we enter the loop where the energy requests are issued, depending on the task type, different patterns are followed to generate the service quanta and energy packets.

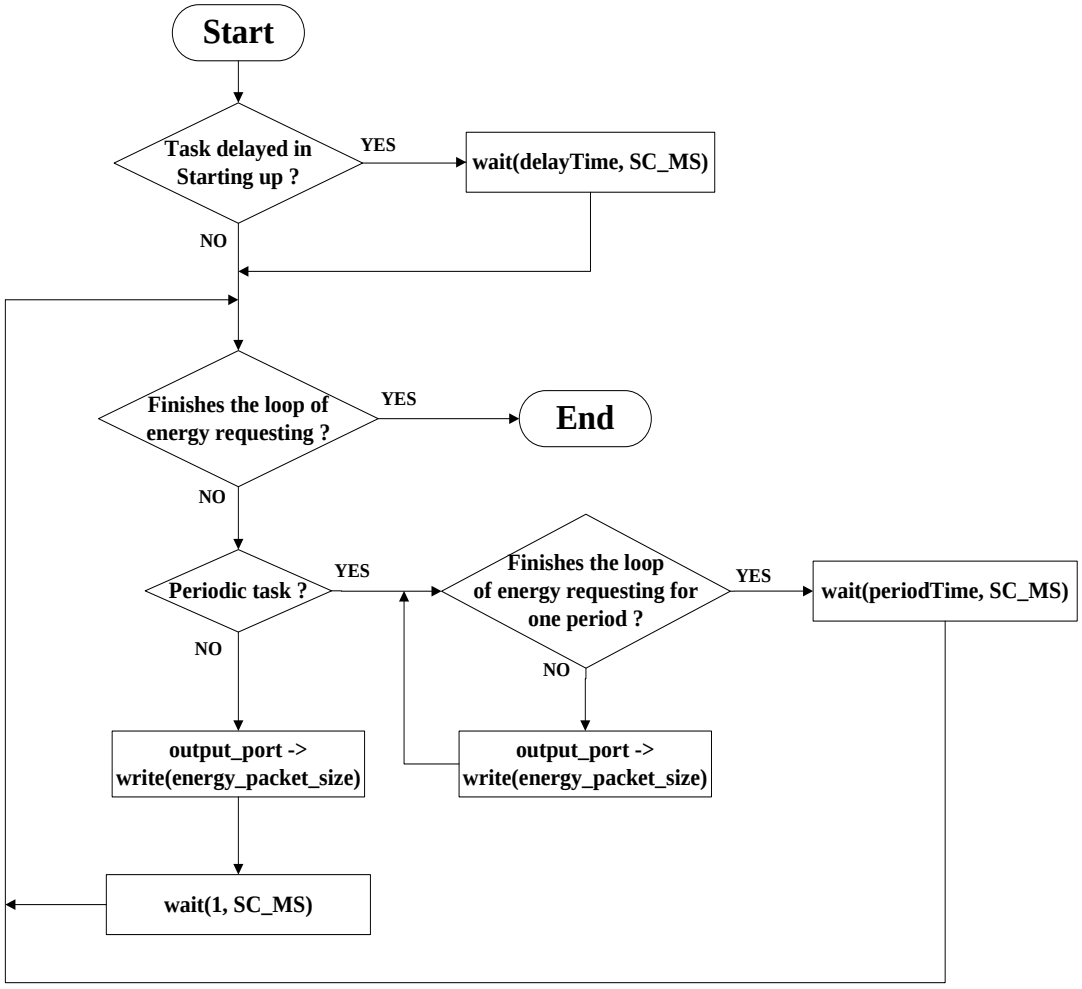


Figure 4. 4: Flow Chart of the SystemC-based Task Modelling in the Producer Module

For time-sensitive tasks that generate energy requests periodically, the producer module controls the period of service quanta generation, the number of service quanta per period, and the energy packet size. This will determine the workload and the average power of a periodic task. For instance, if a task has a period of 10 Tus, in each period, it request 3 service quanta that lasts 3 Tus and the energy packet size of each service quantum is 10 Eus, then, the workload is 30% and the average power over one period is 3 Pus. Specifically, in this simulation, real-time tasks are modelled with fixed-length periods while interactive tasks are modelled with random-length periods; in both cases, the number of service quanta per period and the energy packet size are variable over the time. At the beginning of each period, a loop is executed to request the service quanta of one period; the energy packet size value of each requested service quanta is stored into the corresponding FIFO request buffer through the *write( )*

function and the output port of the producer module. Once the above loop finishes, the *wait(sc_time)* function is called to stop the task until its next period starts. At the side of the consumer module, if the service quanta of a periodic task are consumed in a speed that is faster than their generation speed, then, the corresponding request buffer will become empty before the next period begins, and during this time the task is regarded as idle.

For batch tasks that request energy continuously, in each CPU time unit, the producer generates a service quantum and writes the energy packet size value into the corresponding FIFO request buffer. It turns out to the scheduler that a new service quantum is requested once the previous service quantum is finished. Therefore, the request buffer will never become empty until all requested service quanta are finished by the consumer. As in the case of periodic tasks, the energy packet size of batch tasks is also variable over the time.

Note that, all variable numbers of the test-bench are generated from the C++ random function with a discrete uniform distribution. Thus, their average values can be easily obtained for computing the maximum long-term power share of periodic time-sensitive tasks.

4.3.3 Obtaining scheduling results

To obtain the experimental results from the simulation test-bench, three extra pieces of SystemC code have been inserted into the consumer module; these codes are mainly added at the end of the scheduling loop in Figure 4.2 and should be checked for execution in each CPU time unit.

The first piece of SystemC code is employed to check the long-term task power shares under the EFQ scheduling algorithm. To achieve this goal, the cumulative energy consumption of all tasks are sampled into a text file every 100 Tus. The 100 Tus period is detected by checking whether the current CPU time returned by the *sc_time_stamp( )* function is divisible by 100.

The second piece of SystemC code is used to check the time-constraint

compliance of periodic real-time tasks. First, we use the same method as above to detect the deadline of periodic tasks. Once a deadline of any task is detected, the actual cumulative service time of the task and its expected cumulative service time before the deadline are compared. If the former is smaller than the latter, it means the task is not served the requested service time before the deadline, then, a deadline miss is recorded.

The third piece of SystemC code is employed to measure the response time of interactive tasks. To achieve the goal, we need to firstly detect whether the service quanta of one period are all served to an interactive task by comparing the number of service quanta that the interactive task has actually received with the number of service quanta that are requested in the current period. This requires passing the period length and workload of interactive tasks from the *main.cpp* to the consumer module. Then, if the service quanta of one period have all been served, the response time is computed based on the current CPU time returned by the *sc_time_stamp( )* function and the actual time when the current period starts requesting service quanta. Finally, after the scheduling loop is finished, we compute the maximum and average response time of the interactive task based on the recorded response time of all periods.

4.4 Task characterizations for simulation

This section provides the characterization of the tasks for the simulation. An overview of the task characterization is shown in Table 4.1. Batch 1 and Batch 2 are regular batch tasks with continuous energy demand, while Real-time and Interactive are time-sensitive tasks with periodic energy demand. Real-time has a fixed-length period of 10 time units (Tus), while Interactive is simulated with a random-length period that ranges from 40 to 60 Tus. Both time-sensitive tasks request a variable number of service quanta per period. In addition, their energy packets have variable size. All variable numbers in Table 4.1 are drawn from the C++ random function with a discrete uniform distribution. Thus, their average values can be easily obtained.

Real-time has an average of 3 service quanta per period and the average size of the energy packet is 10 Eus. Interactive has an average period of 50 Tus, every period has an average of 10 service quanta and the average size of the energy packet is 5 Eus. For a convenient analysis of the scheduling results, the average size of the energy packet in both batch tasks is set as 8 Eus to ensure a constant value for the maximum long-term power share and the worst-case power share.

Table 4. 1: Characterization of Tasks in the Simulation

	Real-time	**Interactive**	**Batch 1**	**Batch 2**
Period (Tus)	10	50±10	N/A	N/A
Num. of service quanta / period	3±1	10±4	N/A	N/A
Energy packet size (Eus)	10±3	5±2	8±1	8±3
Max. long-term power share	0.375	0.125	$\lim(x \to 1)$	$\lim(x \to 1)$
Worst-case power share	0.553	0.297	N/A	N/A

Based on the task characterization in Table 4.1, the maximum long-term power share and the worst-case power share (definitions refer to section 3.3.1) are computed. In the long-term case, the average power of Real-time and Interactive is 3 power units (Pus) and 1 Pu (in maximum), respectively. Then, the two batch tasks have an average power (in minimum) of:

$$\left(1 - \frac{3}{10} - \frac{10}{50}\right) \times 8 = 4\ Pus.$$

Therefore, the maximum long-term power shares of Real-time and Interactive are 0.375 and 0.125, respectively. When Real-time has a worst-case power of:

$$\frac{4\ \times 13}{10} = 5.2\ Pus,$$

the average power of Interactive is 1 Pu, and the average power of the two batch tasks is:

$$\left(1 - \frac{4}{10} - \frac{10}{50}\right) \times 8 = 3.2\ Pus.$$

Thus, the worst-case power share of Real-time is:

$$\frac{5.2}{5.2 + 1 + 3.2} = 0.553.$$

When Interactive has the worst-case power of:

$$\frac{14 \times 7}{40} = 2.45\ Pus,$$

the average power of Real-time is 3 Pus, and the average power of both batch tasks is:

$$\left(1 - \frac{3}{10} - \frac{14}{40}\right) \times 8 = 2.8.$$

Thus, the worst-case power share of Interactive is 0.297.

4.5 Simulation results

Based on the simulation test-bench and the task characterizations, simulation experiments are designed to evaluate the different EFQ properties in this section. To meet the requirements of energy-centric processor scheduling, the EFQ algorithm should be able to achieve proportional power sharing, time-constraint compliance, and when necessary, a tradeoff between them. In the remaining of this section, the EFQ algorithm will be evaluated from these three aspects through specifically-designed simulation experiments. Because a simulation of the whole operational time consists of a series of epoch simulations, the simulation is run for only one epoch and a system in which the epoch energy E_{epoch}^{i} equals 50,000 energy units (Eus) is considered.

4.5.1 Maintaining proportional power sharing

Figure 4.5 shows how the system power is proportionally shared under SEFQ when the task characterizations in Table 4.1 are employed as the simulation input and both Real-time and Interactive are reserved a power share that equals the maximum long-term power share. The solid lines are the (reference) power shares when the average parameters in Table 4.1 are employed, whereas the dotted lines are those obtained by averaging the results of 10 repetitive simulations when the random

parameters (generated by 10 different random seeds) specified in Table 4.1 are used. As can be observed in Figure 4.5, proportional power sharing among tasks is long-term guaranteed even under a variable energy load. Similar results are achieved under the BSEFQ [78]. However, the share fluctuations under BSEFQ are larger than those under SEFQ due to its fairness deficiency.

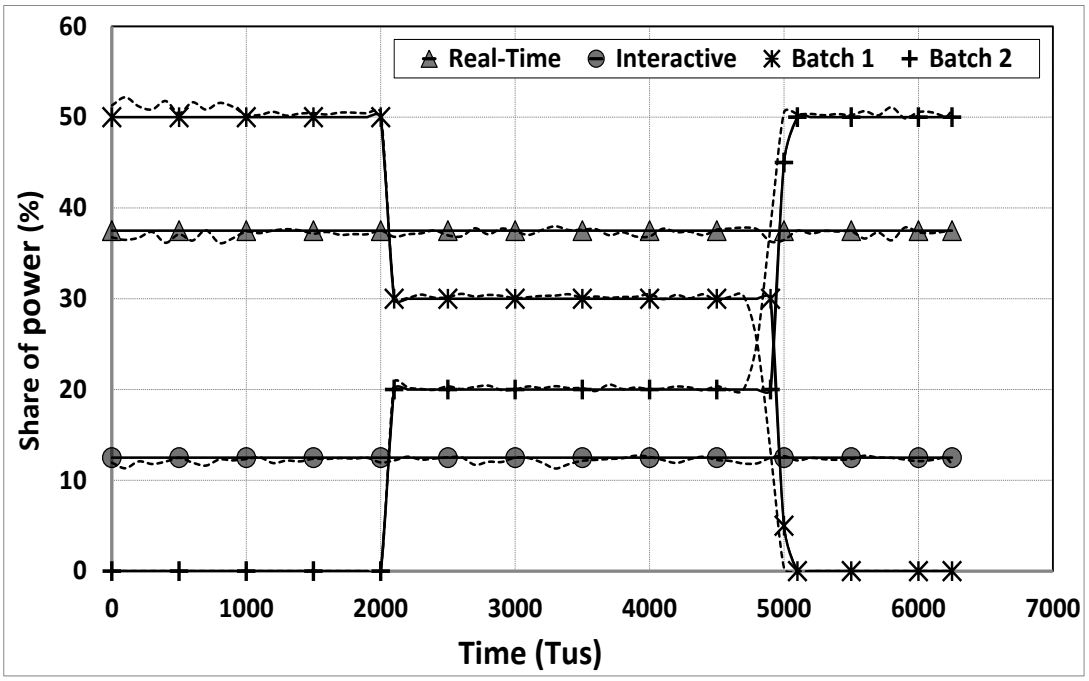


Figure 4. 5: Proportional Power Sharing under SEFQ

To introduce system dynamics, the launch of Batch 2 is advisedly delayed 2,000 Tus, and the energy allocation of Batch 1 is limited to 15,000 Eus; all other tasks start at time 0 and run until exhausting the 50,000 Eus epoch energy E_{epoch}^{i}. The sharp slopes in Figure 4.5 are due to the join event of Batch 2 (at 2,000 Tus) and the exit event of Batch 1 (at 4,800 Tus). As can be observed, the long-term power shares of Real-time and Interactive are not affected by these changes. This is because both Real-time and Interactive belong to the reserved class; they are both guaranteed reserved power shares that equal their corresponding maximum long-term power shares of 0.375 for Real-time and 0.125 for Interactive. The remaining power share is allocated to Batch 1 and Batch 2 according to their initial weights, 3 and 2,

respectively. Initially, Batch 1 receives the entire 0.5 power share left by the reserved class. At time 2,000 Tus, Batch 2 joins the competition and takes its power share of 0.2, while the power share of Batch 1 drops to 0.3. At a time near 4,800 Tus, Batch 1 becomes completely passive and leaves the system after receiving 15,000 energy units; therefore, Batch 2 takes the entire 0.5 power share. At approximately 6,250 Tus, the CPU goes to idle due to the exhausting of the epoch energy E_{epoch}^{i}. The above results illustrate that the proposed algorithm can properly control the power sharing in dynamic systems where the task weight and number are variable over the time.

In the above simulation, the length of the user-desired epoch time is not specified and no mechanism is available to drive the simulation until the end of one epoch. Therefore, the tasks and the CPU may be put into idle at certain time before the end of one epoch when the epoch energy E_{epoch}^{i} is exhausted. The CPU idle interval before the end of one epoch is not appreciated by those time-sensitive tasks or applications that need to provide a smooth user experience over epochs. To extend their execution to a user-specified epoch time, the energy allocations should be properly set to ensure that the epoch energy E_{epoch}^{i} is rightly exhausted at the end of one epoch. If not, implicitly we lose the opportunity to improve the whole system performance within the epoch. Let us assume Real-time and Interactive are the user-preferred tasks, and they are expecting an epoch time of 8,000 Tus, that is, 800 periods for Real-time and averagely 160 periods for Interactive. As a result, 24,000 Eus energy has to be reserved to Real-time and 8,000 Eus energy has to be reserved to Interactive, considering their average powers being 3 Pus and 1 Pus, respectively. On the other side, the total energy allocation of the two batch tasks has to be restricted in 18,000 Eus to avoid them draining the epoch energy E_{epoch}^{i} before the expected 8,000 Tus epoch time. This issue of energy allocation and epoch time achievement will be further investigated in our later experiments based on a concrete platform.

4.5.2 Time-constraint compliance

Table 4.2 statistically compares the performance of Real-time and Interactive under SEFQ and BSEFQ when they are run against Batch 1 and Batch 2. The results are based on the statistics of 20 sets of simulation data. Real-time has a total number of 625 deadlines on average. Any service quantum that misses its deadline is postponed to the later periods. Thus, the deadline of one period may be missed even if the energy demands in that period are met. Interactive starts a new period if and only if the quanta service from the previous period has been finished.

Table 4. 2: Comparison in Performance of Time-sensitive Tasks

	Reserved power share	**Warp value**	***Real-time* Num. deadline missed***	***Interactive* Mean response time***	***Interactive* Max. response time***
$SEFQ_1$	Max. long-term	0	403±20	46.9±0.7	74.5±1.3
$SEFQ_2$	Worst-case	0	1.4±0.4	18.4±0.2	29.8±0.3
$BSEFQ_1$	Max. long-term	RT > Inter.	0	14.1±0.1	23.8±0.7
$BSEFQ_2$	Max. long-term	RT < Inter.	62.1±2.9	10±0.1	14

* The interval around the mean is based on a 95% confidence interval of Student's t-distribution.

In $SEFQ_1$, although a reservation of the maximum long-term power share for both Real-time and Interactive guarantees that all their energy demands are met in the long-term, the scheduler fails to serve energy timely to meet the time constraints. Therefore, the real-time performance is very poor: Real-time misses more than half of its deadlines and Interactive experiences a mean response time that is close to its average period and a maximum response time that is larger than the average period. For real-time tasks that are able to abandon the unfinished work and jump to the next period, a smaller yet significant number of deadlines will be missed.

To improve the real-time performance, $SEFQ_2$ reserves the worst-case power share for both Real-time and Interactive. In this case, the response time of Interactive and the deadline compliance of Real-time are significantly improved in comparison

with $SEFQ_1$. However, this improvement is achieved at the cost of an overly-reserved total power share that is at least 0.85 in the worst cases, while the total power share of Real-time and Interactive is only 0.5 in average. Because of the over-reservation in power share, only a power share of 0.15 is available for reservation for later-joined time-sensitive tasks. Besides, Real-time still has the risk to miss a few deadlines; the number of deadline misses ranges from 0 to 3 in the 20 sets of scheduling results. The reason is that the energy allocation error in a real scheduling scenario is not considered when computing the worst-case power share for this simulation. Since the exact value of the energy allocation error is difficult to determine in real systems, to ensure the strict time-constraint compliance under SEFQ, a coarse and conservative power share that is larger than 0.85 is required for reservation.

In comparison with SEFQ, BSEFQ is more flexible and effective in supporting various types of time-sensitive tasks. It provides stringent deadline compliance and quick response time when Real-time is given a higher priority (indicated by the warp value) in $BSEFQ_1$, and achieves the optimal response time for Interactive when it is favored over Real-time in $BSEFQ_2$. This is because warped time-sensitive tasks are always scheduled ahead of the normal batch tasks, and the warped task with the highest priority is scheduled ahead of other warped tasks. In $BSEFQ_1$, the requested service quanta of Real-time are immediately served when it begins a new period, and therefore, no deadline is missed; the response time of Interactive is also improved in comparison with SEFQ, because its service quanta are served right after the ones of Real-time. In $BSEFQ_2$, since the service quanta of Interactive are always served before the ones of Real-time, Interactive achieves the best response time while Real-time misses around 10% of the deadlines. Under BSEFQ, both Real-time and Interactive are assigned the maximum long-term power share instead of the worst-case power share; therefore, it avoids the inconvenience and downside of considering the energy allocation error as in SEFQ. Besides, since a total power share of only 0.5 is required for power share reservation in BSEFQ, a remaining power share of 0.5 is available for other tasks.

4.5.3 Trading off power share and time-constraint compliance

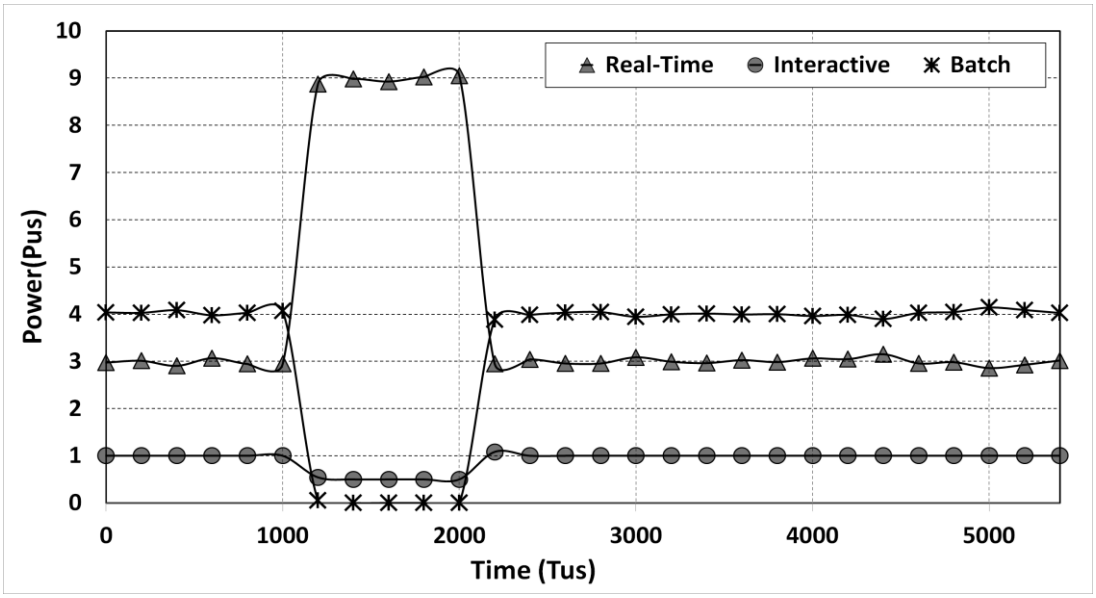
In systems where soft real-time tasks have highly variable workloads over the time, high power pulses may appear under the BSEFQ scheduling algorithm when certain high-priority real-time task is heavily loaded or abnormally behaving during some of its periods. The high power pulses may violate the power shares of other tasks (including time-sensitive tasks) and cause the deterioration of their performances. The warp time limit can be utilized to trade off the power share and the time-constraint compliance of real-time tasks, so that, on the one hand, the time-constraint compliance can be guaranteed when the normal energy load of the real-time task does not impair the power of other user-preferred tasks, and on the other hand, high power pulses can be restrained when the real-time task is over-requesting energy.

In the following simulation, we show how the warp time limit of BSEFQ can be properly adjusted to reach a trade-off between the power share and the time-constraint compliance of real-time tasks. Real-time and Interactive are assumed as user-preferred tasks; both of them are allowed to warp their starting energy tag to gain scheduling priority. Besides, Real-time is assigned a higher warp value (indicating priority) than Interactive. To generate high energy loads in task Real-time, its workload is deliberately increased to 9 service quanta per period during time 1,000 to 2,000 Tus. Besides, for clarity and without loss of generality, only one batch task (Batch 1) is considered for energy competition in the simulation.

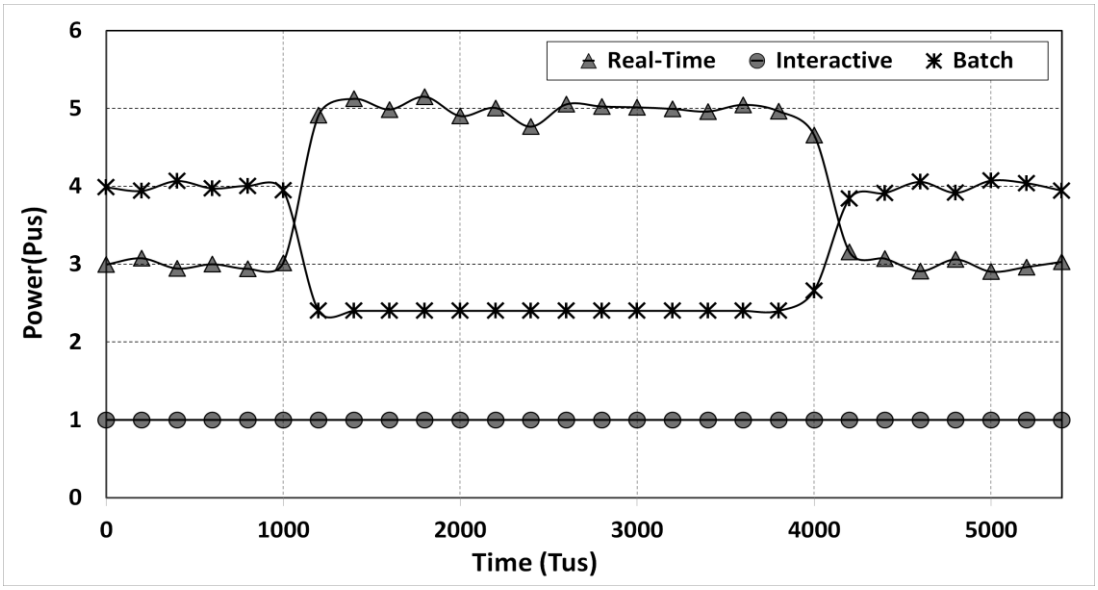
Figure 4.6 shows the power change of tasks under EFQ when Real-time is abnormally behaving with high energy loads during time 1,000 to 2,000 Tus. The graphs are obtained by averaging the scheduling results of 10 repetitive simulations with different random seeds. To focus on the power changes during the abnormally-behaving periods, only the scheduling results until time 5,400 Tus are shown in the graphs.

Figure 4.6-a) shows the task powers under BSEFQ without warp time limit or SEFQ with Real-time being assigned the worst-case power share. In both cases, Real-time is allowed to consume a power as high as 9 Pus, which starves Interactive

to a power level of 0.5 Pus and causes it suffering a maximum response time of 100 Tus during time 1,000 Tus to 2,000 Tus. Because the performance of the user-preferred task Interactive is not protected from the abnormal behaviors from task Real-time, the user experience of the system is greatly degraded by the misbehaving of Real-time. Besides, the power of Batch 1 is unfairly starved to zero during the abnormal periods of Real-time.



a) BSEFQ without warp time limit or SEFQ with worst-case power share for Real-time



b) BSEFQ with a 5 Tus warp time limit

Figure 4. 6: Trading off Power Share and Time-constraint Compliance with EFQ

Figure 4.6-b) shows the task powers under BSEFQ when the warp time limit of Real-time is set as 5 Tus. In this case, Real-time is scheduled as the highest-priority task only within the allowed 5 Tus period of warp time. Once the allowed warp time expires, Real-time is un-warped; and before its next period begins, it has to compete with other tasks based on its weight. For this reason, the maximum power of Real-time is properly controlled under 5 Tus; the 1 Pus power of Interactive and its performance are protected from the misbehaving of Real-time, and the power of Batch 1 is preserved to some extent. Note that, because of the power restraining on Real-time during 1,000 Tus and 2,000 Tus, the unserved service quanta of Real-time are postponed to the later periods, appears in Figure 4.6 as a 5 Pus power that lasts a time longer than 1,000 Tus.

According to the above simulation results, we can see that the adjustment of the warp time limit in BSEFQ plays an important role in protecting and isolating the power as well as the performance of different tasks.

4.6 Summary

A high-level modelling and simulation of the EFQ scheduling algorithm is implemented in this chapter. The system-level modelling language SystemC is employed to build the model because it can simulate concurrent processes and support cycle-accurate model for software algorithms. In the high-level abstraction, each task is modelled as a producer that continuously or periodically generates service quantum requests into one privately-owned FIFO buffer, while the EFQ scheduler and the CPU are modelled as a consumer that selects from the different FIFO buffers the next service quantum to serve in each scheduling tick. This producer-consumer model avoids incurring the complexity of modelling a realistic CPU scheduler, but allows a convenient characterization of the task energy loads and a flexible adjustment of the EFQ scheduling algorithm.

Based on the producer-consumer model, a simulation test-bench is developed to evaluate the EFQ scheduling behaviors from three aspects: power share management,

time-constraint compliance, and the ability to balance the power share and the time-constraint compliance.

The first simulation on power share management shows that EFQ can successfully share the power among tasks in proportion to the user-specified ratio, and in addition, EFQ is effective in protecting the power share of target tasks upon the change of the task number.

The second simulation on time-constraint compliance compares the real-time scheduling performance of SEFQ and BSEFQ. The results show that the time-constraint compliance under SEFQ can only be guaranteed when a power share that is larger than the worst-case power share is assigned, while under BSEFQ, the real-time performance of time-sensitive tasks can be effectively and flexibly guaranteed by assigning a power share that is no less than the maximum long-term power share.

The final simulation demonstrates that by adjusting the warp time limit to restrict the maximum time that one time-sensitive task can run with high priority, BSEFQ can balance the power share and the time-constraint compliance of the time-sensitive task, and avoid potential high power pulses that may be caused by the highly-fluctuating energy loads of the task. In the next chapter, the proposed EFQ algorithms will be implemented in the Linux kernel.

Chapter 5

Linux-based Implementation

This chapter presents the implementation of the energy-based fair queuing algorithm in the Linux. In the first part of this chapter, we introduce the Linux scheduling subsystem and demonstrate how its architecture can be maximally utilized to implement the EFQ algorithm. In the second part of the chapter, we develop a simulation test-bench based on a Linux scheduler simulator. This simulation test-bench serves two purposes. First, it helps to learn the Linux scheduling subsystem and debug the Linux-based EFQ implementation from the user space; second, it allows an assessment of the EFQ scheduling algorithm through simulated energy loads as in the SystemC-based test-bench. In this dissertation work, the main purpose of the simulation test-bench is for user-space debugging of the EFQ implementation in the Linux kernel. The simulation results under this test-bench are not specifically analyzed in this chapter because they are very similar with those of the SystemC-based test-bench when the same task characterizations are used as the simulation input.

5.1 EFQ implementation in the Linux kernel

Linux implements different scheduling polices for real-time and conventional non-real-time processes. Since the kernel version 2.6.23, the completely fair scheduler (CFS) has been adopted as the default scheduler for non-real-time process scheduling in Linux. Because the CFS algorithm is a variant of fair queuing and it shares the same basic principles with the EFQ algorithm, the organizational structure of the CFS scheduler has been maximally utilized to ease the code modification of EFQ implementation. The work is carried out in the Linux kernel V3.3 with around 150 lines of code; it can be ported to the latest kernel versions with minor modifications.

The Linux-based EFQ implementation is abstractly described in Figure 5.1. As can be seen, the implementation of EFQ requires works from the bottom data structures and reference tables, to core EFQ scheduling and share management functions and the system call interface. The different implementation steps will be introduced in detail in the next five sub-sections.

As shown in Figure 5.1, the default CFS run queue, which is in fact a red-black tree, is utilized to organize the tasks. The EFQ scheduler operates on the run queue according to the effective starting energy tag of each task. In the Linux kernel, the effective starting energy tag is computed based on the kernel load weight of each task, rather than the effective weight. The parameter flow in Figure 5.1 demonstrates how the effective starting energy tag in the kernel space is affected by the reserved share and initial weight in the user space. As can been seen, the reserved share and initial weight are passed into the kernel space via the extended system call interface to compute the effective weight; then, by referring to a conversion table with the effective weight, the corresponding kernel load weight is obtained for computing the effective starting energy tag.

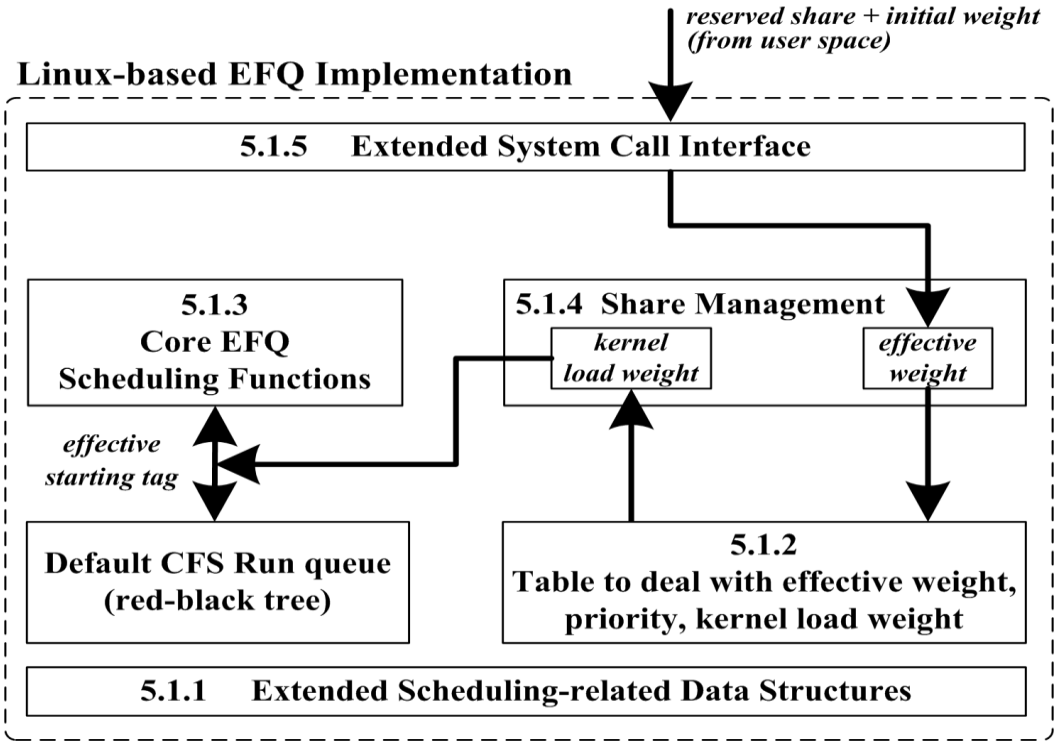


Figure 5. 1: Abstract Description of the Linux-based EFQ Implementation

The remaining of this section is organized as follows. Section 5.1.1 introduces the extension of the data structures that is required to support EFQ scheduling in Linux. Section 5.1.2 presents the modified table that is utilized to convert among the effective weight, priority and kernel load weight. In section 5.1.3, the core scheduling functions of Linux is introduced and the EFQ algorithm is implemented within those functions. In section 5.1.4, the power share protection and reallocation mechanism is implemented to compute the effective weight and kernel load weight of each task. Finally, in section 5.1.5, the system call interface is extended to support the creation and specification of EFQ threads in the user space.

5.1.1 Extend the scheduling-related data structures

Linux employs a series of data structures to sort and manage the processes, these data structures are mainly defined in *include/linux/sched.h* and *kernel/sched/sched.h*, and their relationships are shown in Figure 5.2.

The *task_struct* (Figure 5.2-①) is the central data structure that is used to represent all Linux processes. It contains a large number of elements, including the task state, scheduling class, priority, process identifier (PID), scheduling policy, and much more. Many elements are pointers to another data structure. However, because not all processes are always runnable and schedulable, the *task_struct* is not directly managed by the Linux scheduler. Instead, new data structures for scheduling entities are created to interact with the scheduler. In Linux, based on the *sched_class* element in *task_struct*, processes are generally classified into two classes, the real-time class and the completely fair share (CFS) class (sometimes also called normal class). Therefore, the data structure *sched_entity* (Figure 5.2-②) is created to track the scheduling information of CFS class tasks and the structure *sched_rt_entity* (Figure 5.2-③) is to track real-time class tasks.

Correspondingly, Linux implements two types of run queues and schedulers to manage the scheduling entities of the ready-to-run tasks. Real-time tasks are managed by the run queue *rt_rq* (Figure 5.2-⑤); the tasks are queued either in a First-In First-Out manner (if the *policy* element in *task_struct* is SCHED_FIFO) or in a

Round-Robin manner (with the *policy* set as SCHED_RR). CFS class tasks (with *policy* as SCHED_NORMAL, SCHED_BATCH or SCHED_IDLE) are managed by the run queue *cfs_rq* (Figure 5.2-⑥); this run queue is in fact a time-ordered red-black tree where tasks can be inserted or deleted as nodes. The root of the read-black tree is referred via the *rb_root* element (Figure 5.2-⑦) within the *cfs_rq* data structure, and each node in the tree is represented by the *rb_node* element (Figure 5.2-⑧) that is included in the *sched_entity* structure.

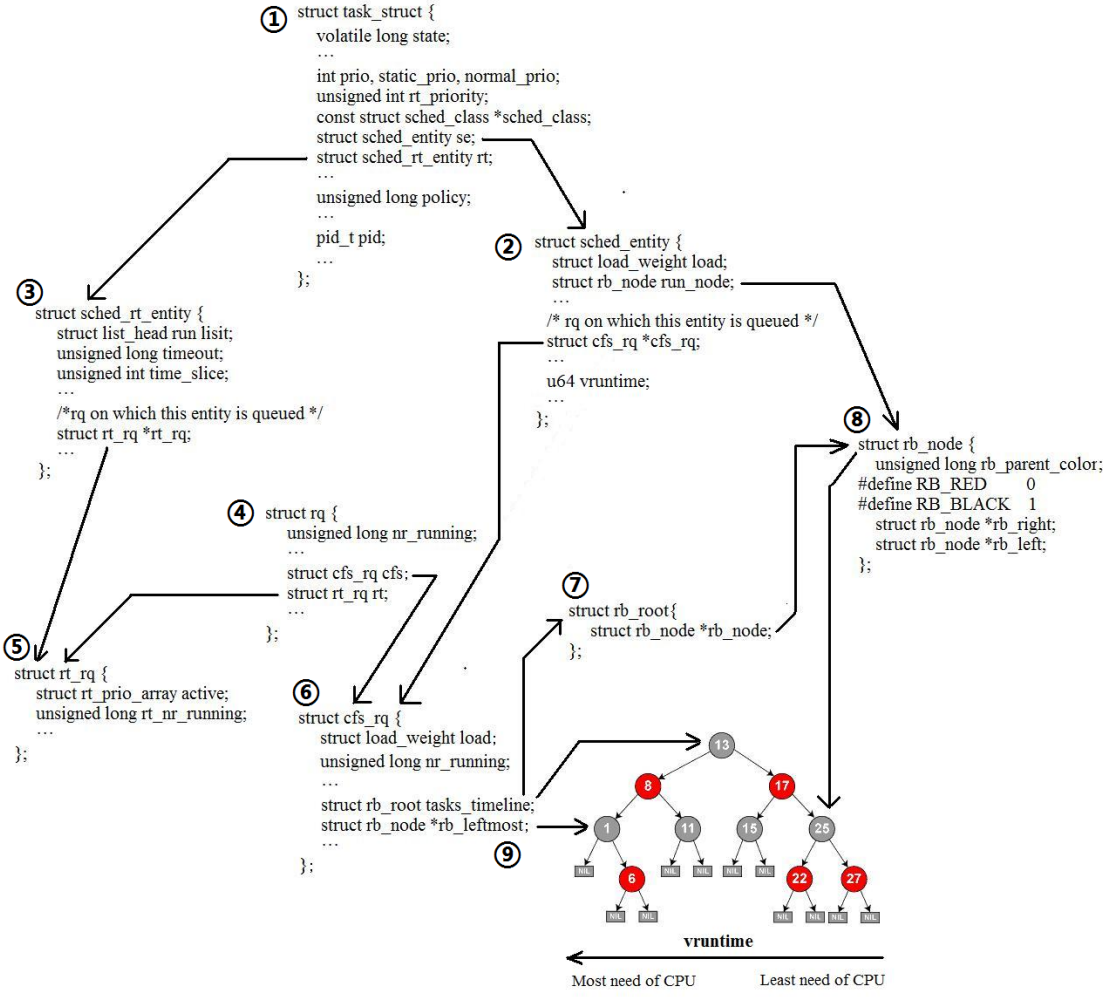


Figure 5. 2: Hierarchy of the Scheduling-related Data Structures in Linux

The most important element that is included in the *sched_entity* structure is the virtual runtime *vruntime*. It traces the amount of time one task has run in reference to the virtual clock and serves as the index of the red-black tree. One task with the smallest *vruntime* is inserted into the leftmost position of the tree, and, in the data

structure *cfs_rq*, a pointer called *rb_leftmost* (Figure 5.2-⑨) is defined to point to the leftmost node of the red-black tree. Within the data structure of both *rt_rq* and *cfs_rq*, there is a variable that records the number of active tasks in each queue. The two data structures are included in the data structure *rq* (Figure 5.2-④), so that it appears to the core Linux scheduler that the different types of tasks are managed by only one run queue. Linux maintains one *rq* for each processor.

To support the EFQ algorithm in Linux, the first step is to define a new scheduling policy macro named SCHED_EFQ in the header file *include/linux/sched.h* and correlate it to the CFS class so that any process that belongs to this policy will be handled by the CFS scheduler. Later on, the CFS scheduler should be modified to implement the EFQ algorithm. But before doing that, the data structure *sched_entity* is extended to include EFQ-related elements, such as initial weight, reserved share, effective weight, standard energy packet size, (effective) starting energy tag and warp parameters. The extended data structure of *sched_entity* is shown in Figure 5.3-a).

```
struct sched_entity {
    struct load_weight load;
    struct rb_node run_node;
    . . .
    u64 initial_weight;
    u64 reserved_share;
    u64 effective_weight;
    u64 standrad_energy_packet_size;
    u64 starting_energy_tag;
    u64 starting_energy_tag_effective;
    . . .
    u64 warp_value;
    u64 warp_time_limit;
    u64 cumulative_warp_time;
    bool warp_enabled;
    . . .
    u64 sum_exec_runtime;
    u64 prev_sum_exec_runtime;
    . . .
① u64 sum_consumed_energy;
② u64 prev_sum_consumed_energy;
    . . .
};
```

a)

```
struct cfs_rq {
    struct load_weight load;
    unsigned long nr_running;
    . . .
    u64 total_initial_weight;
    u64 total_reserved_share;
    u64 min_starting_energy_tag;
    . . .
    struct rb_root tasks_timeline;
    struct rb_node *rb_leftmost;
    . . .
③ u64 total_energy_consumption;
    . . .
};
```

b)

```
struct sched_param {
    int sched_priroity;
    u64 warp_value;
    u64 warp_time_limit;
    u64 standard_energy_packet;
};
```

c)

Figure 5. 3: Extension of the Main Data Structures

Also, the data structure *cfs_rq* is extended to include elements like *total_initial_weight*, *total_reserved_share* and *min_starting_energy_tag*. The elements *total_initial_weight* and *total_reserved_share* are utilized for power share protection and re-allocation that will be presented in section 5.1.4. The element *min_starting_energy_tag* is used to update the system virtual time by tracing the lowest starting energy tag among all active tasks. The extended data structure of *cfs_rq* is shown in Figure 5.3-b).

Note that, extra elements like *sum_consumed_energy* (Figure 5.3-①) and *prev_sum_consumed_energy* (Figure 5.3-②) in data structure *sched_entity* and *total_energy_consumption* (Figure 5.3-③) in data structure *cfs_rq* are exclusively added to trace the energy consumption and generate statistic results. Similar elements are also added into the data structure *sched_rt_entity* and *rt_rq* for comparing the scheduling results under the EFQ scheduler and the default Linux scheduler.

The red-black tree is an efficient and practical implementation of the run queue for normal tasks. To utilize it in the EFQ implementation, all of its data structures are kept unchanged. The new element *starting_energy_tag_effective* of structure *sched_entity* is now referred as the index for task insertion on the red-black tree.

When a new process is created, the EFQ-related elements are all initialized to zero within the *do_fork( )* function; to enable the change of their values after the task creation, the data structure *sched_param* in *include/linux/sched.h* is also extended to include some of the EFQ-related elements, as shown in Figure 5.3-c). Note that, the original data structure *sched_param* only contains one element, *sched_priority*. The user-specified element values of *sched_param* and the scheduling policy macro are passed from the user space to the kernel space via the system call function *sched_setscheduler( )*, which should be modified to recognize the SCHED_EFQ policy and assign values for EFQ-related elements.

5.1.2 Deal with priority and kernel load weight

In the Linux kernel, task priorities are expressed in two forms, the normal priority that is non-zero and globally uniform to all tasks, and the niceness that is specific to CFS class tasks. The Linux kernel uses a scale that ranges from 0 to 139 to represent the process priorities. Larger values indicate lower priorities. Real-time processes always have higher priorities than CFS class processes; they are reserved the priority range from 0 to 99. The range from 100 to 139 is allocated to normal processes, this priority range is mapped to the niceness values in the range [-20, +19]. The Linux kernel priority scale is shown in Figure 5.4.

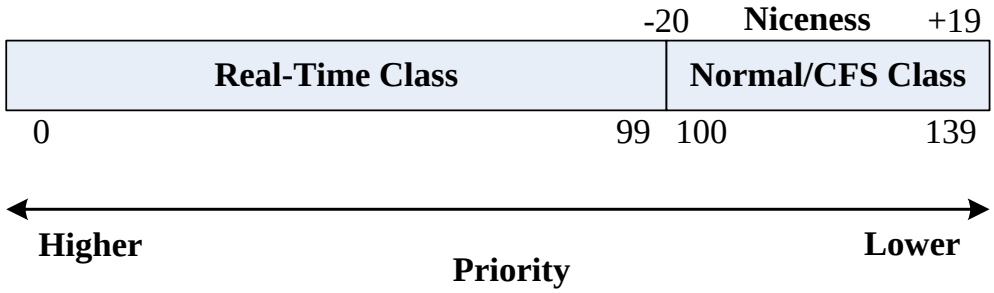


Figure 5. 4: The Linux Kernel Priority Scale

```
include/linux/sched.h
#define  MAX_USER_RT_PRIO      100
#define  MAX_RT_PRIO            MAX_USER_RT_PRIO
#define  MAX_PRIO               (MAX_RT_PRIO + 40)
#define  DEFAULT_PRIO           (MAX_RT_PRIO + 20)

kernel/sched/sched.h
#define NICE_TO_PRIO(nice)     (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio)     ((prio) - MAX_RT_PRIO - 20)
#define TASK_NICE(p)           PRIO_TO_NICE((p)->static_prio)
```

Figure 5. 5: Linux Macros for Priority Conversion

To convert between the different forms of priority representation, the following macros (as shown in Figure 5.5) are defined in *include/linux/sched.h* and *kernel/sched/sched.h.* The MAX_RT_PRIO macro specifies the maximum priority

value of real-time processes, and the MAX_PRIO macro is the maximum priority value of CFS class processes. Upon the creation of one normal process, the default niceness value is 0 with the DEFAULT_PRIO macro set as 120; the process priority can be adjusted via system call function *nice( )* or *set_priority( )* at any time during the execution. Similarly, we can change the priority of a real-time task via the system call function *sched_setparam( )* or *sched_setscheduler( )*.

Linux computes the virtual runtime *vruntime* and the timeslice length of a CFS class task based on the kernel load weight, which is an element of the data structure *load_weight* that is contained in the data structure *sched_entity*. The load weight is determined by the process priority, thus, a niceness table is provided in the *kernel/sched/sched.h* to convert between the niceness priority and the load weight, as shown in Figure 5.6. The table contains one weight entry for each nice level in the range [-20, 19]. For instance, the load weight of a task with a default niceness value of zero is 1024. The weight values in the array are carefully set so that every process that decreases the priority by one nice level can get 10 percent more CPU time [39].

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

Figure 5. 6: The Default Niceness Table of Linux-CFS

To implement the EFQ algorithm based on the CFS architecture, the niceness table in Figure 5.6 has been modified to allow a proper mapping between the kernel load weight and the effective weight. The modified niceness table contains 100 elements, with the load weight values ranging from 100 to 10,000 and increasing in a step of 100, as shown in Figure 5.7. Correspondingly, the nice levels are now extended to a range of [-50, 49] and mapped to the *prio_to_weight* array index with a

difference of 50. With the new niceness table, decreasing the priority by one nice level will increase the load weight by 100; this allows the power share to be reserved in a resolution of 0.01 from the user space. In addition, the new niceness table enables a quick search and matching between the effective weight and the kernel load weight, because the load weight values are all multiples of 100 and have a simple arithmetic relationship with their array indexes. This point will be illustrated in section 5.1.4.

```
static const int prio_to_weight[100] = {
 /* -50 */   10000,      9900,      9800,      9700,      9600,
 /* -45 */    9500,      9400,      9300,      9200,      9100,
 /* -40 */    9000,      8900,      8800,      8700,      8600,
 /* -35 */    8500,      8400,      8300,      8200,      8100,
 /* -30 */    8000,      7900,      7800,      7700,      7600,
 /* -25 */    7500,      7400,      7300,      7200,      7100,
 /* -20 */    7000,      6900,      6800,      6700,      6600,
 /* -15 */    6500,      6400,      6300,      6200,      6100,
 /* -10 */    6000,      5900,      5800,      5700,      5600,
 /*  -5 */    5500,      5400,      5300,      5200,      5100,
 /*   0 */    5000,      4900,      4800,      4700,      4600,
 /*   5 */    4500,      4400,      4300,      4200,      4100,
 /*  10 */    4000,      3900,      3800,      3700,      3600,
 /*  15 */    3500,      2900,      2800,      2700,      2600,
 /*  20 */    3000,      2900,      2800,      2700,      2600,
 /*  25 */    2500,      2400,      2300,      2200,      2100,
 /*  30 */    2000,      1900,      1800,      1700,      1600,
 /*  35 */    1500,      1400,      1300,      1200,      1100,
 /*  40 */    1000,       900,       800,       700,       600,
 /*  45 */     500,       400,       300,       200,       100,
};
```

Figure 5. 7: The Modified Niceness Table for Linux-EFQ

To reflect the range changing of nice levels, the Linux priority scale and the Linux macros for priority conversion have to be modified accordingly. We extend the priority scale to the range from 0 to 199, and map the priority range from 100 to 199 to the nice levels of range [-50, 49], as shown in Figure 5.8 and Figure 5.9.

For a newly-created task, because the DEFAULT_PRIO macro is set as 150, its default niceness value is 0 and the load weight is 5,000 in according to the priority conversion in Figure 5.8 and the new niceness table in Figure 5.6. Note that, under the EFQ scheduling, both real-time and normal processes are managed as EFQ class, or simply fair class. The 0 to 99 priority range for the original Linux real-time class can

actually be cancelled; however, we keep this priority range for creating privileged threads that are used to trace task energy consumptions in the experimental test-bench.

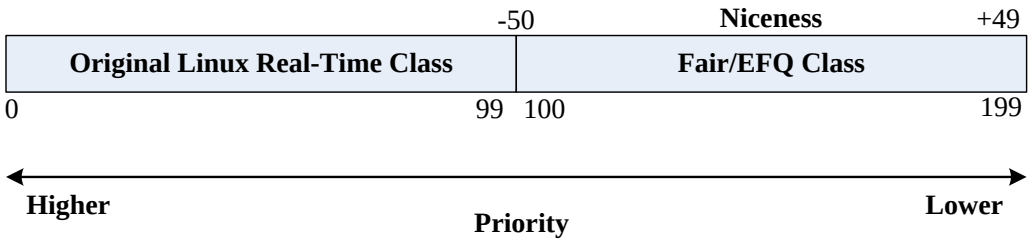


Figure 5. 8: The Extended Priority Scale for Linux-EFQ

include/linux/sched.h

```
#define  MAX_USER_RT_PRIO      100
#define  MAX_RT_PRIO           MAX_USER_RT_PRIO
#define  MAX_PRIO              (MAX_RT_PRIO + 100)
#define  DEFAULT_PRIO          (MAX_RT_PRIO + 50)
```

kernel/sched/sched.h

```
#define NICE_TO_PRIO(nice)     (MAX_RT_PRIO + (nice) + 50)
#define PRIO_TO_NICE(prio)     ((prio) - MAX_RT_PRIO - 50)
#define TASK_NICE(p)           PRIO_TO_NICE((p)->static_prio)
```

Figure 5. 9: Modified Linux Macros for Priority Conversion

5.1.3 EFQ implementation within the core scheduling functions

The Linux scheduler relies on two core scheduling functions: the periodic scheduler function *scheduler_tick* and the main scheduler function *schedule*. The related functions are mainly defined in the files *kernel/sched/core.c*, *kernel/sched/fair.c* and *kernel/sched/rt.c*. Understanding the related functions of the periodical scheduler and the main scheduler is pivotal to the EFQ implementation.

5.1.3.1 The periodic scheduler function *scheduler_tick*

The periodic scheduler function *scheduler_tick* is automatically called when a software interrupt is generated by the scheduling timer. Its main assignment is to update the scheduling-related elements of the current task (being served on the CPU)

and determine whether the current task should be preempted by another one. The code flow diagram in Figure 5.10 provides an overview of the main functions in *scheduler_tick*.

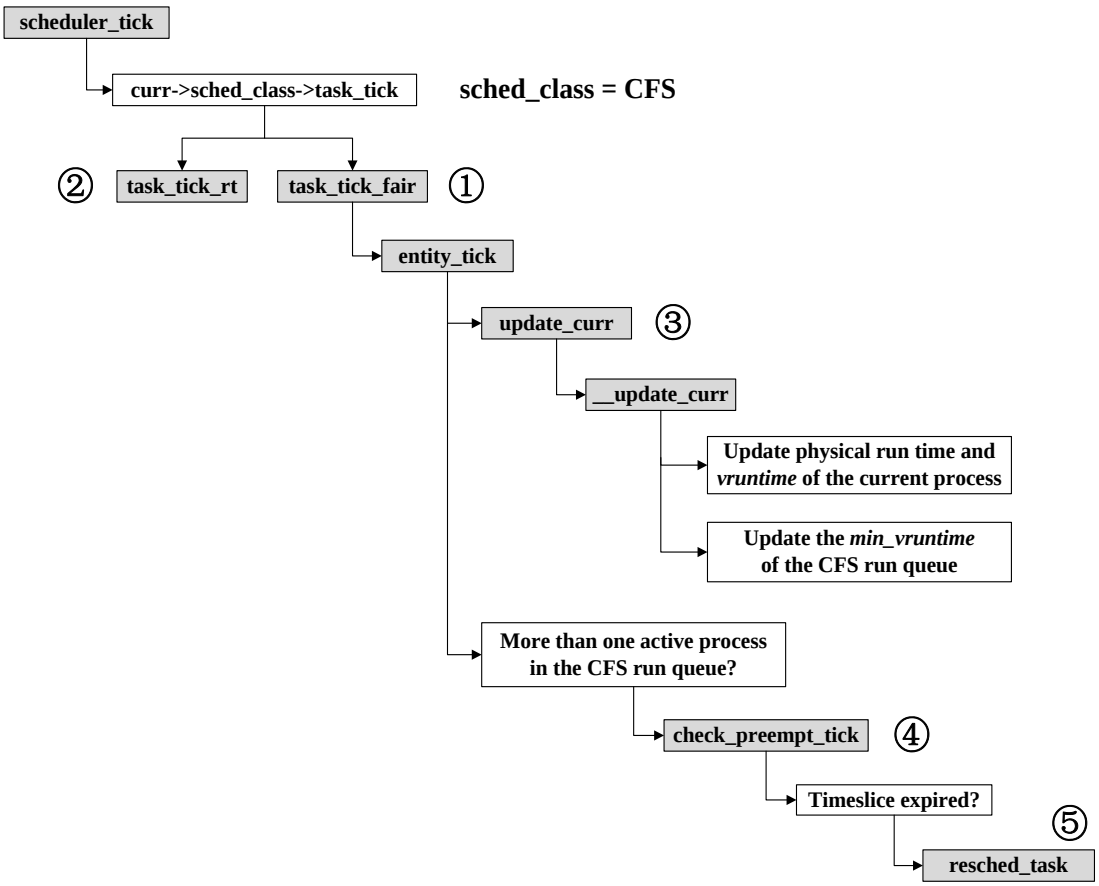


Figure 5. 10: Code Flow Diagram for the Periodic Scheduler Function *scheduler_tick*

Depending on the scheduling class of the current task, different functions are called to implement the periodic scheduler. In the figure, the function *task_tick_fair* (Figure 5.10-①) is called because the current task belongs to the CFS class. However, if the current task belongs to the real-time class, the function *task_tick_rt* (Figure 5.10-②) will be called by *scheduler_tick*. In this thesis, we only focus on function *task_tick_fair* because it is where the core of the CFS scheduling algorithm is implemented. Inside the function *task_tick_fair*, there are two core functions known as *update_curr* (Figure 5.10-③) and *check_preempt_tick* (Figure 5.10-④). The function *update_curr* does two important things, the first is to update the current task

timing information, including the physical run time and the virtual run time *vruntime*, and the second is to update the minimum virtual run time *min_vruntime* of all active tasks in the CFS run queue. After the update, the function *check_preempt_tick* (Figure 5.10-④) is called to determine whether a process rescheduling should be executed. If the current task has used up its timeslice and needs to be rescheduled, the function *resched_task* (Figure 5.10-⑤) will be called to set the TIF_NEED_RESCHED flag. Up to this point, the main assignment of the periodical scheduler function *scheduler_tick* is done. When the kernel returns from the software interrupt of the scheduling timer, the main scheduler function *schedule* will be called to carry out the process rescheduling operations.

To maximize the code sharing between the default Linux-CFS scheduler and the EFQ scheduler, the main work of EFQ implementation is taken place inside the function *update_curr* (Figure 5.10-③) and function *check_preempt_tick* (Figure 5.10-④). The details are given in section 5.1.3.3.

5.1.3.2 The main scheduler function *schedule*

The main scheduler function *schedule* can be called from many points in the kernel to execute the process rescheduling operations. It is either directly called within the current process or called with some delay after the kernel is returning from interrupts and system calls. The first situation happens when the current task is blocked for resource uses or goes to idle, while the second situation happens when the current task finds itself using up the timeslice in a scheduling tick, or a task with higher priority is woken up, or an active task in the run queue changes its scheduling policy and priority. In the second situation, the function *need_resched* is first called to check the TIF_NEED_RESCHED flag; the main scheduler function *schedule* continues to be executed only in condition that the TIF_NEED_RESCHED flag is set.

The main scheduler function *schedule* is in charge of removing the current process from the CPU, selecting the next process to execute, and implementing the context switch between the two processes. The code flow diagram in Figure 5.11 provides a simplified overview of the main functions inside *schedule*.

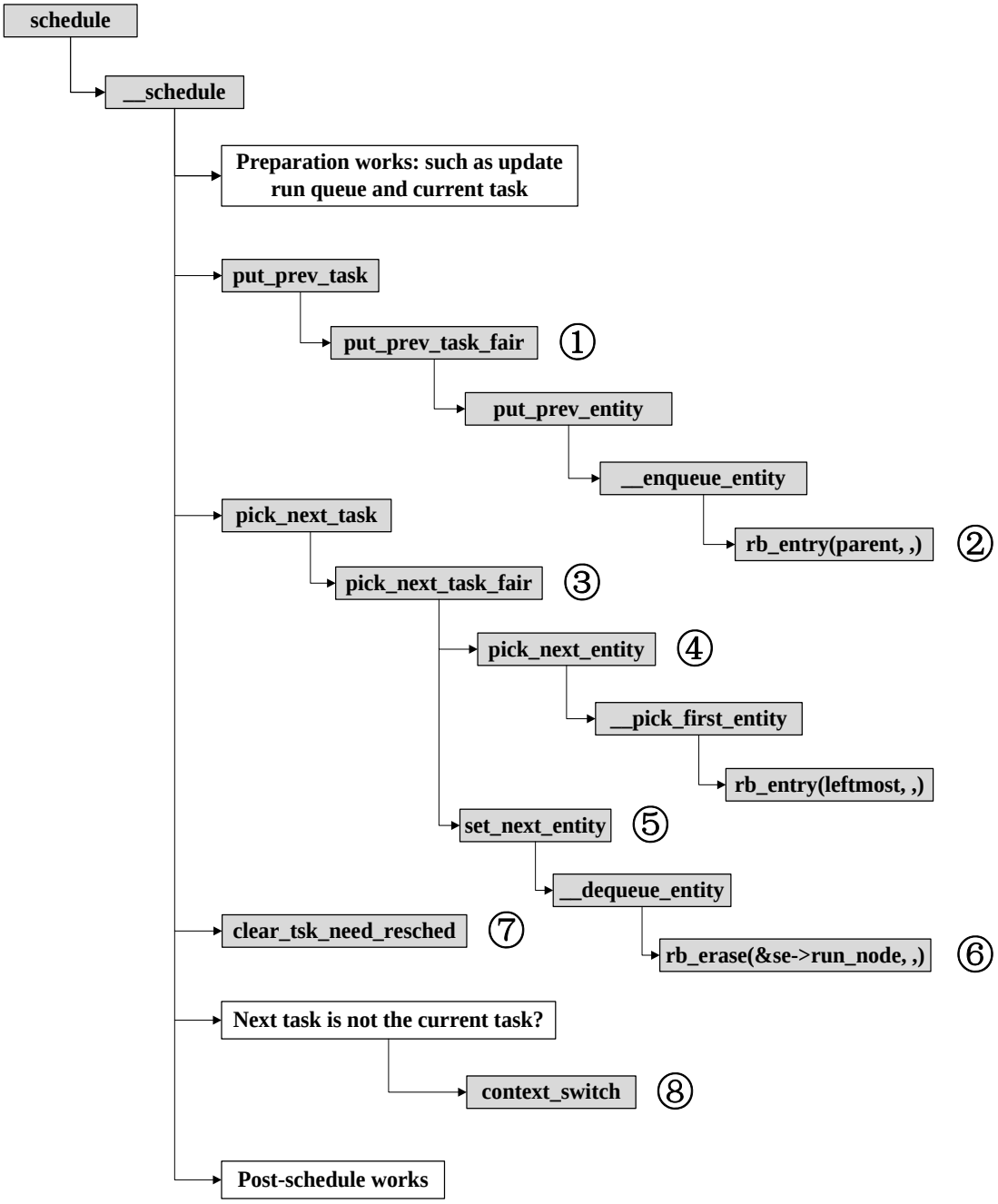


Figure 5. 11: Code Flow Diagram for the Main Scheduler Function *schedule*

Again, different functions may be called to implement the main scheduler depending on the scheduling class; only those functions that are related to the CFS class are shown in Figure 5.11. When the current process needs to be removed from the CPU, the function *put_prev_task_fair* (Figure 5.11-①) is called to move the current process to the CFS run queue that is based on the red-black tree. A task is inserted into the red-black tree via the function *rb_entry* (Figure 5.11-②). Note that,

within the main scheduler function *schedule*, the current task is referred as the previous task *prev_task* due to the process switch that is going to happen. After the removal of the previous process, the function *pick_next_task_fair* (Figure 5.11-③) is called to select the next task to dispatch. This function does two things. First, it calls the function *pick_next_entity* (Figure 5.11-④) to select the leftmost process in the red-black tree, that is, the process with the lowest virtual run time *vruntime*. Second, it calls the function *set_next_entity* (Figure 5.11-⑤) to remove the selected task from the red-black tree and set the pointer *rb_leftmost* of the data structure *cfs_rq* to the new leftmost task in the red-black tree. The leftmost task of the red-black tree is picked also via the function *rb_entry*, while its removal is achieved via the function *rb_erase* (Figure 5.11-⑥). After the current task is removed from the CPU and the next task is selected from the CFS run queue, the TIF_NEED_RESCHED flag of the current task is cleared via the function *clear_tif_need_resched* (Figure 5.11-⑦), and finally, the context switch (Figure 5.11-⑧) between the two processes is executed.

5.1.3.3 EFQ implementation

As mentioned earlier in section 5.1.3.1, to maximize the code sharing between the default Linux-CFS scheduler and the EFQ scheduler, the EFQ implementation work is mainly carried out inside the CFS class functions *update_curr* and *check_preempt_tick*. These two functions are included in the CFS class function *task_tick_fair*, which is defined in the file *kernel/sched/fair.c* and called by the periodic scheduler function *scheduler_tick*. Figure 5.12 shows a simplified flow chart of the EFQ implementation based on the periodic scheduler function *scheduler_tick*. The readers are advised to refer back to Figure 5.10 to have a complete view of the periodic scheduler. The main scheduler function *schedule* and its related CFS class functions keep unchanged and are directly utilized by the EFQ scheduler. Especially, by employing the new element *starting_energy_tag_effective* of the data structure *sched_entity* as the index for task insertion on the red-black tree, the CFS run queue is fully utilized by the EFQ scheduler for process queuing operations.

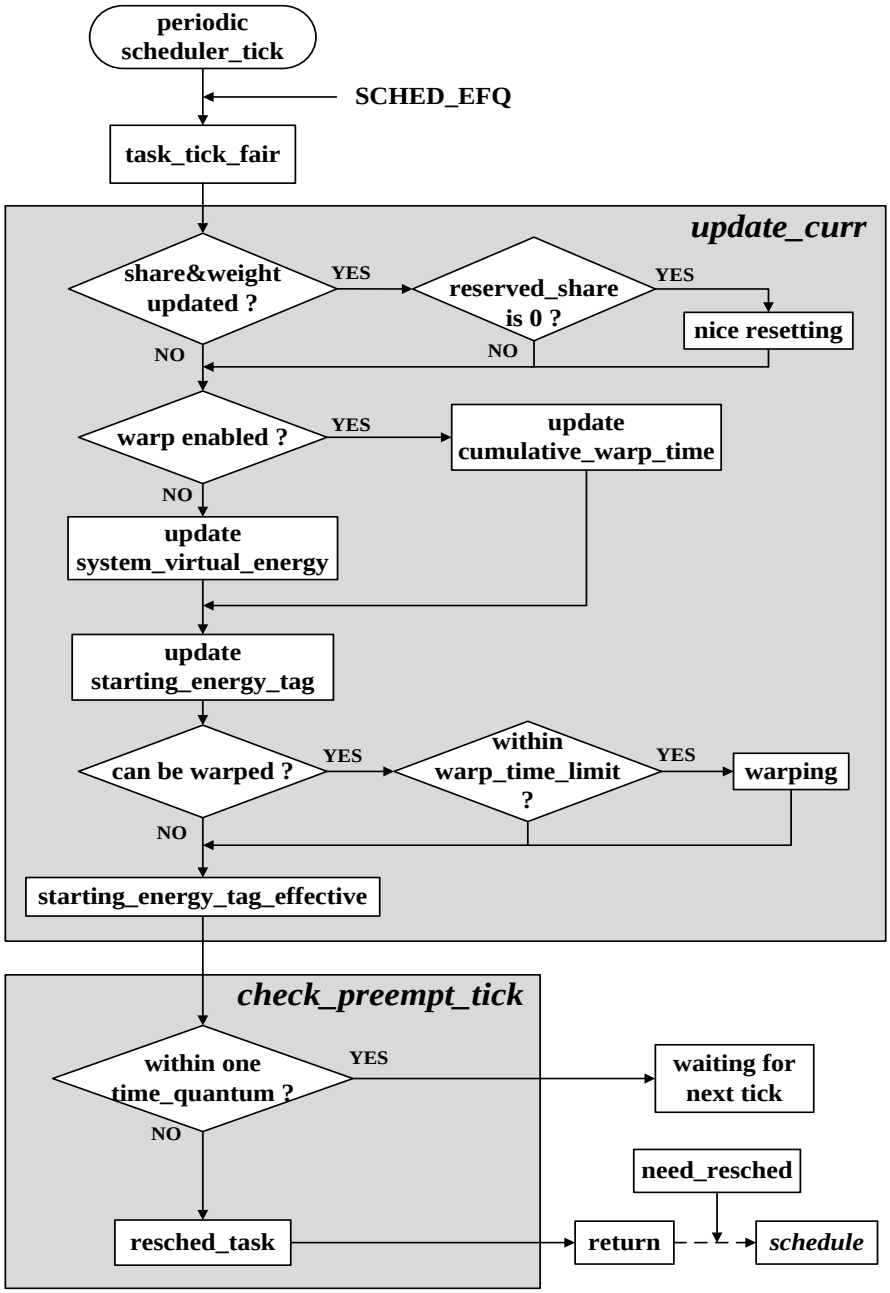


Figure 5. 12: Flow Chart of EFQ Implementation within the Periodic Scheduler *scheduler_tick*

In Figure 5.12, the function *task_tick_fair* is periodically called for EFQ scheduling if the current task belongs to the EFQ class. Then, *task_tick_fair* calls the *update_curr* function to update the execution time and EFQ-related variables of the current task.

Inside the *update_curr* function, the first assignment is to check the scheduling environment. For a task whose reserved share is zero, if there has been a change of the active task set, or the reserved share and initial weight of certain task have been

changed, the nice level as well as the kernel load weight of the task should be recomputed before updating the starting energy tag; however, for a task with a non-zero reserved share, its kernel load weight should be kept unchanged and nice resetting is not required. The method of nice resetting is introduced in section 5.1.4.

Then, the next assignment is to update the system virtual energy and the cumulative warp time of the current task. Recall that the system virtual energy is updated through the new element *min_starting_energy_tag* of the data structure *cfs_rq*. The operation of updating the system virtual energy is executed only in condition that the task is not warped; this ensures that the system virtual energy is monotonically non-decreasing and it always traces the lowest starting energy tag of all active tasks. Instead, if the current task has been running warped, its cumulative warp time is updated by adding the task execution time since its last update. After that, we update the starting energy tag of the current task. The starting energy tag is updated based on the task execution time, energy packet size and kernel load weight.

Finally, we implement the warp mechanism and compute the effective starting energy tag. A predefined warp value is subtracted from the starting energy tag of the current task if it is time-sensitive and the continuous time it has been running with warp in the current period is less than the warp time limit. A time-sensitive task that reaches to its warp time limit will keep the original starting energy tag; however, the task will be allowed to warp the energy tag another time once its next period starts.

Once the above works have been done, the periodic scheduler will check if the current task should be preempted by another task by calling the *check_preempt_tick* function. If the task execution time is within one scheduling time quantum, nothing is to be done until the next scheduling tick occurs; otherwise, the *resched_task* function is called to set the TIF_NEED_RESCHED flag of the current task. After the kernel returns from the scheduling tick, the main scheduling function *schedule* is called to determine the next task to be dispatched by comparing the effective starting energy tag of the current task with the effective starting energy tag of the left-most task in the red-black tree-based run queue *cfs_rq*. If the current task has a smaller effective starting energy tag (especially when the task has a large power share or it is running

with a wrap), it will stay on the CPU for another time quantum; otherwise, the current task is forcibly removed from the CPU and inserted back to the run queue based on its effective starting energy tag, while the scheduler will make a context switch to the left-most task in the red-black tree.

Besides of the virtual energy updating in the periodic scheduler, the EFQ scheduler should also update the virtual energy and the run queue *cfs_rq* when a new task is launched or an old task wakes up and re-joins the energy competition. Figure 5.13 shows the flow chart of EFQ implementation upon task launch or wakeup.

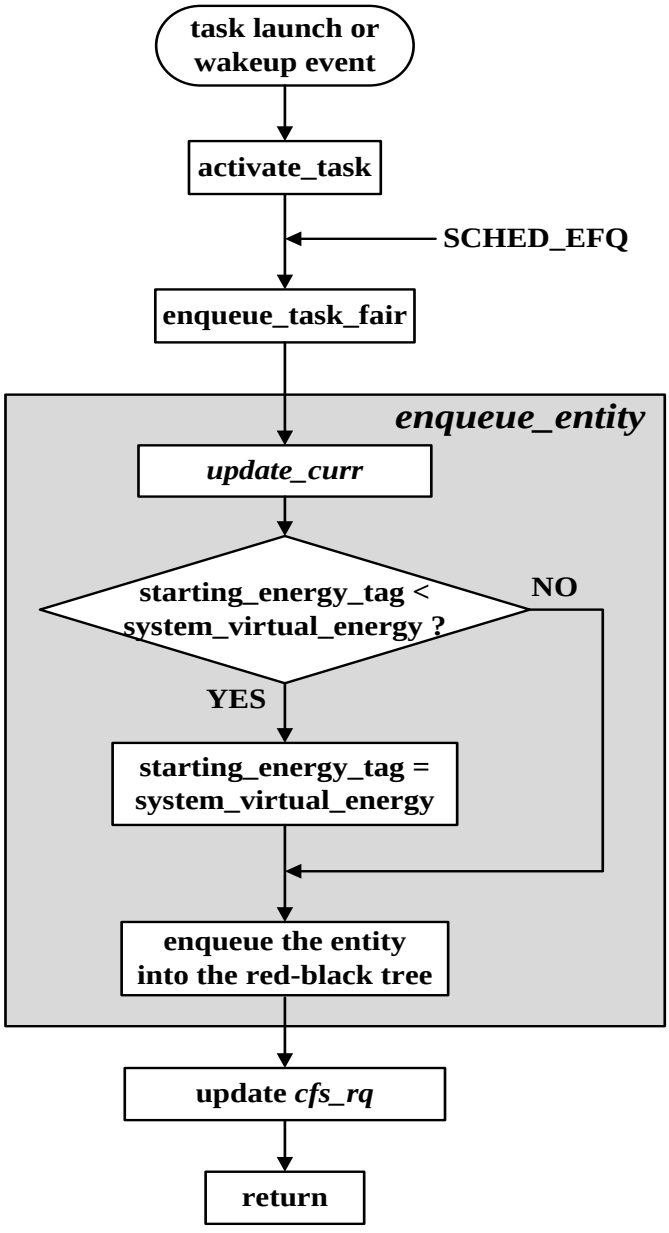


Figure 5. 13: Flow Chart of EFQ Implementation upon Task Launch or Wakeup

As shown in Figure 5.13, an event of task launch or wakeup will interrupt the current kernel work, and then, the newly-joined task is activated via the *activate_task* function, which further calls the *enqueue_task_fair* function to complete the activation work for EFQ class tasks. Inside the *enqueue_task_fair* function, the main work is done via the *enqueue_entity* function. It first calls the *update_curr* function to update the EFQ-related statistics of the current task in service, and then, the starting energy tag of the newly-activated task is updated. To prevent the newly-activated task from continuously occupying the CPU with a considerably smaller energy tag, the starting energy tag of the newly-activated task is updated to the value of the system virtual energy of the run queue *cfs_rq* if it has a smaller value. After the updating work is done, the newly-activated task is inserted into the run queue *cfs_rq* based on the energy tag, and the number of active tasks as well as the total weight load of the *cfs_rq* are updated before the kernel returns to its original work.

5.1.4 Implement the share protection and reallocation

The flowchart of share protection and reallocation is demonstrated in Figure 5.14, the related functions are defined mainly in *kernel/sched/fair.c*. The module of share management may be called upon four events: new task launching (Figure 5.14-①), share or weight adjusting (Figure 5.14-②), old task leaving (Figure 5.14-③), and the request of nice resetting from the *update_curr* function (Figure 5.14-④).

When a new task is launched (Figure 5.14-①), or the reserved share and initial weight of an old task are modified (Figure 5.14-②) in the user space, the values of the reserved share and initial weight are passed into the kernel space to update the corresponding variables in the *sched_entity* structure. This update event will be later detected in the *update_curr* function (Figure 5.14-④) that is periodically performed by the scheduling tick, so that other tasks can have the chance to adjust the effective weight if necessary. To avoid incurring float computation in the kernel space and reduce the scheduling overhead, the reserved share is scaled up by 10,000 in the user space.

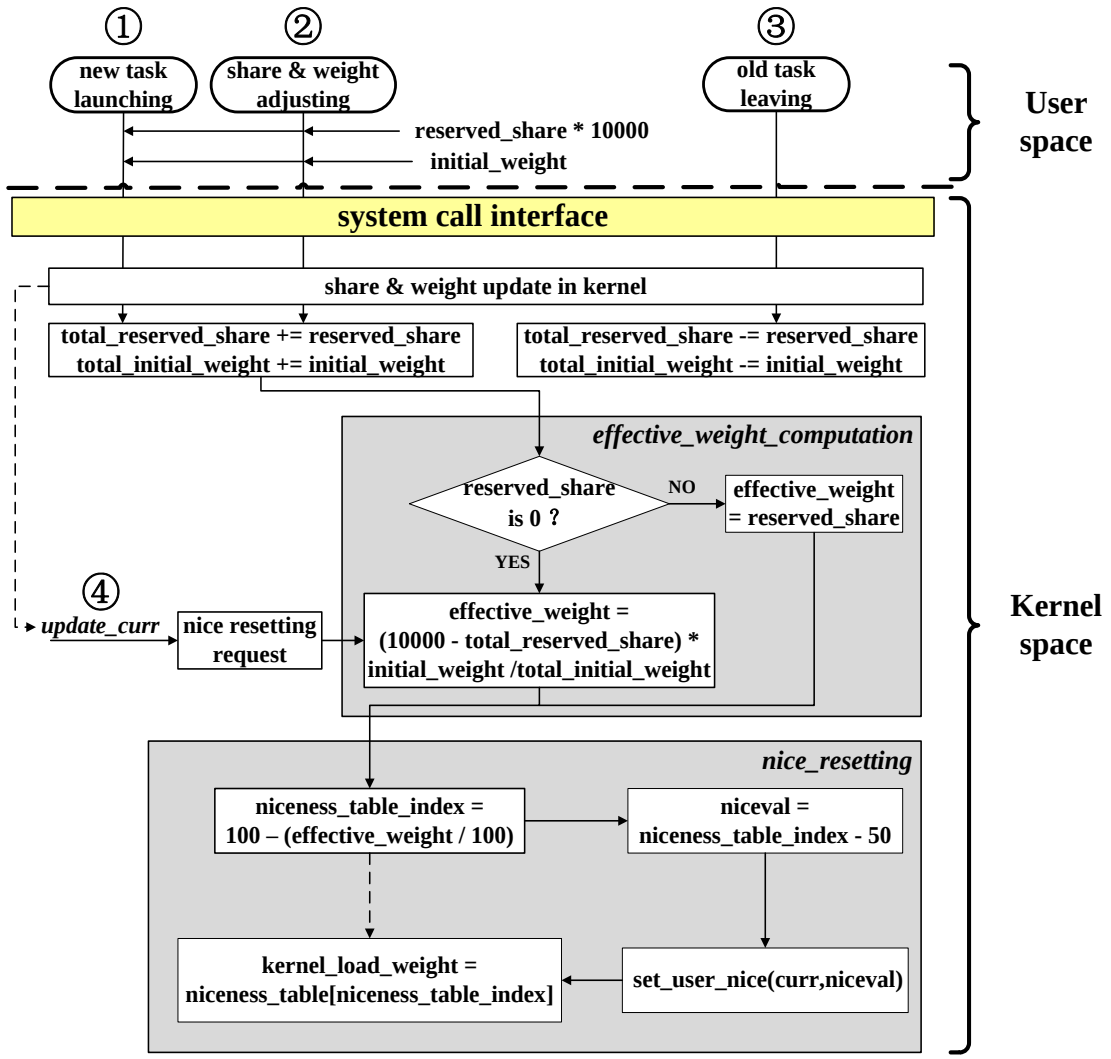


Figure 5. 14: Flowchart of Power Share Management

After the updating of the reserved share and initial weight, the module of *effective_weight_computation* is called to compute the effective weight for the task. If the task is an important or time-sensitive one that is holding a non-zero reserved share, the effective weight is equal to the reserved share; otherwise, as shown in the figure, the effective weight should be computed based on the total reserved share, the total initial weight of all active tasks, and the initial weight of the concerned task.

Finally, the effective weight is passed to the *nice_resetting* module to compute the nice value and niceness table index, and the kernel load weight is set to be the indexed value of the niceness table. With the modified niceness table for the EFQ policy in Figure 5.7, the niceness table index and the nice value can be easily

computed as shown in Figure 5.14. In Linux, the *set_user_nice* function is specifically defined to set the nice value and the kernel load weight of tasks.

In the case when an old task leaves the system after finishing its work (Figure 5.14-③), we only update its kernel-space reserved share and initial weight at that time. This event of share and weight updating will be detected by the *update_curr* function (Figure 5.14-④) in a later moment, if the task that calls the *update_curr* function is a normal task with no share reservation, the *effective_weight_computation* module and *nice_resetting* module will be called to reset its nice value and kernel load weight.

5.1.5 Extend the system call interface

The final step of EFQ implementation concerns the extension of the system call interface. Figure 5.15 shows a list of the system calls that are specifically designed for EFQ scheduling.

```
// create a task and initialize the EFQ-related variables
sys_clone;

// set scheduling policy, priority and EFQ-related variables after task creation
sys_sched_setscheduler;

// update the share and weight when a new thread joins the system
sys_thread_join;

// update the share and weight when an old thread leaves the system
sys_thread_leave;

// adjusts the share and weight of a thread during its execution
sys_weight_adjust;

// reset the warp value and warp time limit
sys_warp_reset;

// adjust the energy packet size of a thread during its execution
sys_powert_adjust;
```

Figure 5. 15: List of System Calls for EFQ Scheduling

First, the existing system call *sys_clone* and its related kernel functions are modified to allow the initialization of EFQ-related variables when a thread is created or a process is forked. Then, another existing system call *sys_sched_setscheduler* and its related function *sched_setscheduler* are modified so that the new scheduling policy SCHED_EFQ can be recognized and set as the policy of the created task structure; this system call is also used to pass EFQ parameters, such as warp, warp time limit and energy packet size, to the task structure of a thread right after it is created.

Besides, several new system calls have been implemented to enable the user space thread to interact with its kernel task structure during the whole life cycle of the thread. Specifically, the system call *sys_thread_join* and *sys_thread_leave* allow updating the share and weight in the kernel space when a new thread joins the system and an old thread finishes the work, respectively; the system call *sys_weight_adjust* allows the thread adjusting its share and weight in the middle of the thread execution; the system call *sys_warp_reset* allows resetting the warp value and warp time limit; and the system call *sys_power_adjust* allows changing the energy packet size of the thread based on the executed codes in the program.

5.2 Simulation-based debugging

In the previous section, we have introduced the EFQ implementation based on the Linux scheduling subsystem; in this section, a simulation test-bench that relies on the EFQ-embedded Linux scheduling subsystem is further presented for user-space debugging of the EFQ implementation in the Linux kernel. This simulation test-bench is based on a Linux scheduler simulator called LinSched [79], which is able to simulate the scheduling behavior of the Linux scheduler in an isolated user-space environment. Such a simulation test-bench is very useful because it can be used to verify the scheduling behavior of the Linux-based EFQ scheduler from the user space. Debugging the EFQ implementation on Linux is now as easy as debugging a user space program by employing traditional Linux debugging tools like the GDB debugger. This greatly eases the EFQ implementation in this thesis work. If the EFQ

scheduling algorithm is found with any problem during the simulation, we can simply debug the EFQ scheduler with breaking points, modify the EFQ codes, re-compile the simulation test-bench program, and re-check the updated EFQ scheduling behaviors without incurring the trouble of recompiling the whole Linux kernel and rebooting it on hardware platforms. The LinSched-based test-bench allows a pre-evaluation of the Linux-based EFQ scheduling algorithm before moving to its complete implementation and assessment on a concrete computing platform.

In the remaining of this section, we will first have an overview of the Linux scheduler simulator Linsched; then we go deep into its simulation engine API and show how the API can be extended to support the EFQ scheduling simulation and debugging from the user space; finally, a LinSched script is given to demonstrate how to create an EFQ scheduling simulation scenario based on the extended Linsched API functions. Note again that, the simulation results under the LinSched-based test-bench are not specifically analyzed in this dissertation work because they are very similar to those of the SystemC-based test-bench when the same task characterizations are used as the simulation input.

5.2.1 The Linux scheduler simulator

The Linux scheduler simulator (LinSched) [79] is an open-source tool that hosts the Linux scheduling subsystem in an isolated user-space environment. It can be used to observe and modify the behaviour of the Linux kernel scheduler on various platforms and workloads independently of other Linux subsystems, and it offers stable and repeatable scheduling results that are not affected by the program operating environment. These features make it a valuable tool in simulating and prototyping new Linux scheduling policies, such as the EFQ algorithm.

The overall architecture of LinSched is illustrated in Figure 5.16. LinSched is a user-space program that consists of four main components: stimulti, simulation engine, environment module, and portions of the Linux kernel (including the Linux scheduler subsystem). At the bottom level is the host operating system that runs the

LinSched program. The Linux kernel of the host operating system is not necessarily the same as the one included in the LinSched program, in fact, they are totally independent of each other.

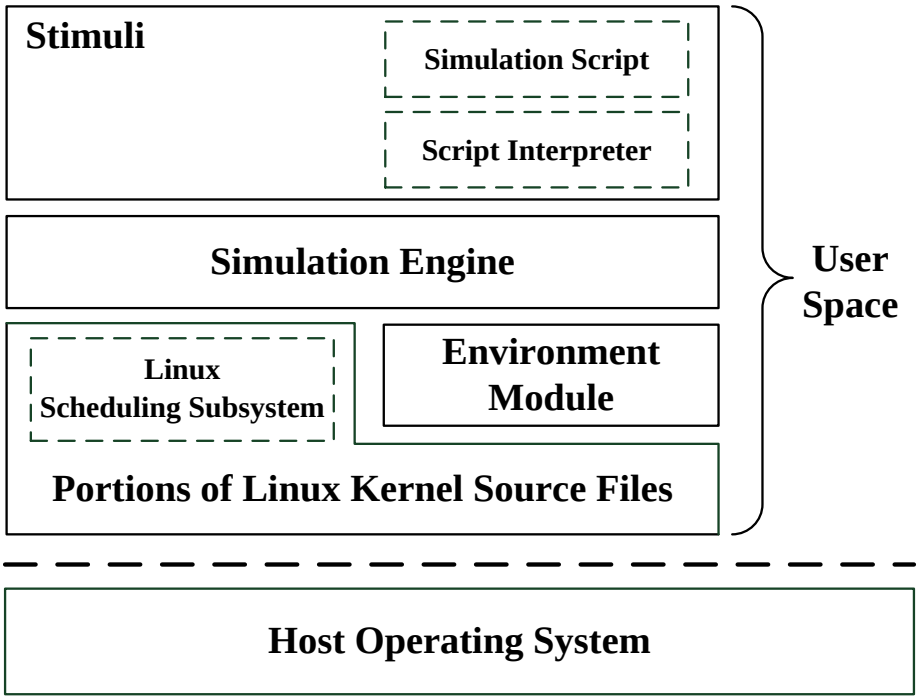


Figure 5. 16: Architecture of the LinSched Scheduler Simulator [79]

The environment module provides a minimum emulation of the necessary components of the Linux kernel so that the Linux scheduler can be executed outside of the kernel. It satisfies the code dependencies of the Linux scheduling subsystem and the simulation engine by abstracting the indispensable functions and macros, most of which are supplied directly by including a subset of the Linux kernel source and header files. On the environment module is the simulation engine, which provides an Application Programing Interface (API) than can be used to run a simulation and interact with the Linux scheduling subsystem. Through the API functions, the simulation engine can control the simulation and call appropriate scheduler functions from the kernel source files to simulate the Linux scheduling. Additionally, the API can call functions to create tasks with specific workloads and initialize the kernel environment emulation with user-specified processor topologies. Finally, above the simulation engine is the stimuli. It is a scripting environment that performs a set of

API functions to set up the desired test-bench scenario. The simulation scripts can be parameterized to run simulations based on a variety of different workloads and platforms.

The LinSched simulator was originally developed at the University of North Carolina based on the Linux version 2.6.23 [80]. It was later revived by Google based on the Linux version 2.6.35 to validate the behaviour of the Completely Fair Share (CFS)-based Linux kernel scheduler [81]. The latest LinSched simulator is based on the Linux kernel version 3.3-rc7 [82], it is employed to design the simulation test-bench and prototype the EFQ scheduling algorithm in this work. The LinSched repository is actually the Linux kernel sources 3.3-rc7 with a new subdirectory called *linsched* under the *tools* directory. The LinSched source code for the environment emulation module and simulation engine is contained in the *tools/linsched* subdirectory. In this work, the LinSched-based test-bench program is compiled using the GCC 4.6.3 compiler under the 64-bit Ubuntu 12.04 LTS OS.

5.2.2 Extend the LinSched API for EFQ simulation and debugging

To develop a simulation test-bench with LinSched, it is important to know the main LinSched simulation engine API functions and understand how they interact with the Linux scheduling subsystem and the environment emulation module.

```
// initialize the Linux scheduling subsystem and environment emulation module
linsched_init;

// create a CFS class task with the scheduling policy marked as SCHED_NORMAL
linsched_create_normal_task ;

// create a task data structure that defines the call back function and sleep/run pattern
linshced_create_sleep_run;

// runs the simulation for a number of timer ticks
linsched_run_simulation;

// print out the simulation results and individual task statistics
linsched_print_task_stats;
```

Figure 5. 17: List of Main LinSched Simulation Engine API Functions

Figure 5.17 provides a list and simple description of the most important LinSched API functions. To know more details on these functions, the readers are advised to refer to the paper [79] and LinSched source code [82].

To design the LinSched simulation test-bench and utilize it for user-space EFQ debugging, the original LinSched simulation engine API should be extended not only to create tasks that are managed by the EFQ scheduler but also to support the specification of the energy load for each task.

To start with, a new LinSched API function is defined for creating EFQ tasks. As shown in Figure 5.18, the function first creates a task with certain task data *td* (Figure 5.18-①), then it sets the scheduling policy of the task as SCHED_EFQ and passes the EFQ-related parameters to the kernel functions via the system call function *sched_setscheduler* (Figure 5.18-②), finally it passes the reserved share and initial weight to the kernel functions and sets the nice and kernel load weight of the task via the system call function *thread_join* (Figure 5.18-③).

```
struct task_struct *linsched_create_EFQ_task
(struct task_data *td, float reserved_share, int initial_weight,
u64 warp_value, int warp_time_limit)
{
        struct sched_param params = { };
        struct task_struct *p;
        int id = num_tasks++;
        assert(id < LINSCHED_MAX_TASKS);

        ① /* Create task with task data td*/
        p = __linsched_tasks[id] = __linsched_create_task(td);
        __linsched_set_task_id(p,id);

        ② /* Initialize scheduling priority and EFQ-related variables */
        params.sched_priority = 0;
        params.warp_value = warp_value;
        params.warp_time_limit = warp_time_limit;

        ② /* Set scheduling policy and pass EFQ-related variables */
        sched_setscheduler(p, SCHED_EFQ, &params);

        ③ /* Pass the share and weight, set the nice and kernel load weight*/
        thread_join(p, reserved_share, initial_weight);

        assert(current->cgroups->subsys[0]);
        assert(p->cgroups->subsys[0]);
        return p;
}
```

Figure 5. 18: The New LinSched API Function for Creating EFQ Tasks

The task data *td* determines the energy requesting pattern of each task. To simulate the energy loads of normal batch tasks, interactive tasks and real-time tasks, we have implemented three new LinSched API functions to create different task data structures. Figure 5.19 shows the API function for interactive tasks. The API functions for real-time and batch tasks are not specifically given as they are similar and simpler.

```
struct task_data *linsched_create_sleep_run_interactive
(int period_average, int period_range, int busy_average, int busy_range,
int energy_packet_size_ave, int energy_packet_size_range)
{
    struct task_data *td = malloc(sizeof(struct task_data));
    struct sleep_run_task *d = malloc(sizeof(struct sleep_run_task));
    d->task_type = 1;  //Mark this task as an interactive task

    ① /* Initialize the energy packet size*/
    d->min_energy_packet_size = energy_packet_size_ave - energy_packet_size_range;
    d->max_energy_packet_size = energy_packet_size_ave + energy_packet_size_range;

    ① /* Initialize the busy time of one period */
    d->min_busy = busy_average - busy_range;
    d->max_busy = busy_average + busy_range;
    d->busy = random(d->min_busy,d->max_busy);

    ① /* Initialize the length of one period */
    d->min_period = period_average - period_range;
    d->max_period = period_average + period_range;
    d->period = random(d->min_period, d->max_period);

    ① /* Initialize the sleep time of one period */
    d->sleep = d->period - d->busy;

    ② /* Initialize the variables used for generating scheduling statistics */
    d->cumulative_period_time = d->period;
    d->n_deadline_missed = 0;
    d->n_deadline_met = 0;
    d->response_cumulative = 0;
    d->response_maximum = 0;
    d->task_create_time = current_time;

    ③ /* Initialize the sleep run functions with the defined task data*/
    sleep_run_init(&d->sr_data);
    td->data = d;
    td->init_task = sleep_run_start;
    td->handle_task = sleep_run_handle;
    return td;
}
```

Figure 5. 19: The LinSched API Function for Generating Energy Loads of Interactive Tasks

As can be seen in Figure 5.19, this API function first initializes the range of period, busy time, sleep time and energy packet size for interactive tasks (Figure 5.19-①). All of the above parameters are fluctuating ones and their values are generated over the time with the *random* function. Besides, to generate scheduling statistics of the simulation, some extra parameters are defined to record the task response time and count the deadline misses (Figure 5.19-②). Finally, the sleep run functions are initialized and the task data is set to be handled by the *sleep_run_handle* function (Figure 5.19-③), which is the core function to control the energy load simulation based on the task data *td->data*. To avoid incurring too much complexity, we will not go into the details on how the *sleep_run_handle* function is implemented.

Besides of the above extensions on the Linsched API, the API function *linsched_run_sim* and *linsched_print_task_stats* have also been modified to allow the recoding of EFQ scheduling results. Inside the *linsched_run_sim*, a piece of code has been inserted so that the simulation test-bench can periodically sample the cumulative energy consumption of each task into a text file; also, an energy budget has been set so that the test-bench can terminate the simulation once the energy budget is used up. In the *linsched_print_task_stats*, the response time and deadline misses of time-sensitive tasks have been computed and sent to the scheduling statistics files.

5.2.3 An EFQ simulation scenario

With the extended LinSched API interface, we are now ready to create the EFQ simulation test-bench based on the EFQ implementation in the Linux scheduling subsystem.

Figure 5.20 shows a LinSched script for an EFQ scheduling scenario that is characterized in Table 5.1. The scheduling scenario is the same as the one in Table 4.1 of the SystemC-based simulation.

```
void linsched_test_main(int argc, char **argv)
{
  linsched_init( ); // Initialize the simulator

   // Create a real-time task, period fixed at 10 Tus, requests 3±1 time quanta per period, energy packet size
   is 10±3 Eus, power share reserved as 0.375, warp value set as 80000000, warp time limit is 4 Tus
  linsched_create_EFQ_task(linsched_create_sleep_run_rt(10, 3, 1, 10, 3), 0.375, 0, 80000000, 4);

   // Create an interactive task, variable period of 50±10 Tus, requests 10±4 time quanta per period, energy
   packet size is 5±2 Eus, power share is 0.125, warp value set as 20000000, warp time limit is 14 Tus
   linsched_create_EFQ_task(linsched_create_sleep_run_interactive(50, 10, 10, 4, 5, 2), 0.125, 2000000,
   14);

  // Create a batch task, it is busy 100% of the time, energy packet size is 8±1 Eus, initial weight is 3
  linsched_create_EFQ_task(linsched_create_sleep_run(0, 100, 8, 1), 3);

  // Run the simulation for 2000 time units (Tus) or timer ticks
  linsched_run_sim(2000);

  // Create a batch task with a delay of 2000 Tus, it is busy 100% of the time, energy packet size is 8±3,
  initial weight is 2
  linsched_create_EFQ_task(linsched_create_sleep_run(0, 100, 8, 3), 3);

  // Run the simulation for another 6000 Tus
  linsched_run_sim(6000);

  // Print out the task statistics
  linsched_print_task_stats( );

  return;
}
```

Figure 5. 20: A LinSched Script for EFQ Simulation based on Linux

Table 5. 1: Characterization of Tasks for A Linux-based Simulation of EFQ

	Real-time	**Interactive**	**Batch 1**	**Batch 2**
Period (Tus)	10	50±10	N/A	N/A
Num. of service quanta / period	3±1	10±4	N/A	N/A
Energy packet size (Eus)	10±3	5±2	8±1	8±3
Reserved share	0.375	0.125	0	0
Initial weight	0	0	3	2
Launch time / Delay (Tu)	0	0	0	2000

In the script, we first create a real-time task with fixed-length period, an interactive task with variable period, and a batch task that is continuously busy before its termination; after running the simulation for 2,000 Tus, we launch another batch task to join the simulation. All the tasks have a variable size of the energy packet.

Specifically, the real-time task and interactive task both have a variable number of service quanta per period; they are reserved a power share of 0.375 and 0.125, respectively. The two batch task has no share reservation; they compete for energy with the initial weight, being 3 and 2, respectively. There is a total energy budget for the whole simulation, and every task can be set with an energy budget. If a task finds itself using up its energy budget, it will goes to idle and leaves the energy competition. The test-bench will terminate the whole simulation and output the simulation results if the total energy budget is used up.

5.3 Summary

In this chapter, the EFQ algorithm is implemented within the Linux scheduling subsystem. Since the Linux-CFS scheduler is an implementation of the fair queuing algorithm in processor scheduling, the organizational structure of the CFS scheduler has been maximally utilized to implement the EFQ scheduler. To add fundamental supports to the EFQ algorithm, the scheduling-related data structures, the priority scale and the load weight scale in Linux are firstly extended. Then, the main components of the EFQ algorithm, including the starting tag computation, the system virtual energy updating and the warp mechanism, are implemented within the core scheduling functions of the Linux scheduler, and the parameter based on which the tasks are inserted into the run queue is set as the effective starting energy tag. In this way, the tasks can be managed with the default Linux scheduler but the scheduling decisions are made according to the EFQ policy. After that, the power share protection mechanism is implemented within and on top of the core Linux scheduling functions, this mechanism can detect the change of the scheduling environment and re-compute the effective weight for each task. Finally, the system call interface of Linux is extended to allow kernel-space EFQ class tasks interacting with the user-space threads or processes.

To debug the EFQ implementation, the simulation engine API of the Linux scheduler simulator LinSched is extended to add the support of EFQ scheduling

simulation, and based on that, a simulation test-bench is developed to isolate the Linux scheduling subsystem and simulate the EFQ scheduling behavior in the user space. This allows a convenient debugging of the Linux-based EFQ implementation from the user space. With the LinSched-based simulation test-bench, the EFQ implementation can be verified through a comparison of the simulating results with those of the SystemC-based high-level simulation. At the end of the chapter, a LinSched script is provided as an example of creating one simulation scenario for the Linux-EFQ scheduler. In the next chapter, the experimental test-bench based on a concrete computing platform will be introduced to verify the Linux-based EFQ implementation.

Chapter 6

Experimental Test-bench

The implementation of the Linux-based EFQ scheduler is further verified though experiments that are based on a concrete computing platform. This chapter presents the test-bench design of the experiments. It starts with an overview of the test-bench architecture and the experimental method, and then, continues with an introduction on the computing platform and the power supply and measurement system; after that, follows a presentation of the benchmarks employed in the experiments and the method to characterize their energy loads; finally, it ends with a presentation of the multithreading test-bench program that is employed to generate experimental results.

6.1 Test-bench and methodology overview

6.1.1 Test-bench architecture

The architecture of the test-bench for EFQ assessment is shown in Figure 6.1.

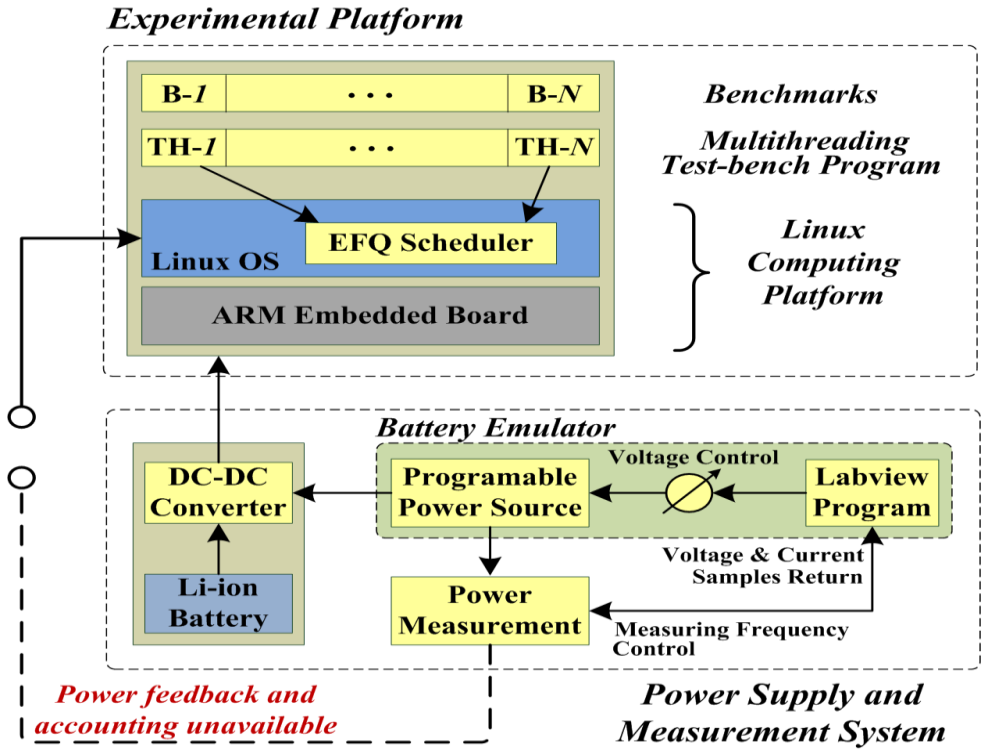


Figure 6. 1: Architecture Overview of the Experimental Test-bench

As can be observed, the test-bench consists of two parts, the experimental platform and the power supply and measurement system.

The experimental platform is an ARM-based Linux computing platform which hosts a multithreading test-bench program and an open-source benchmark suite. The multithreading test-bench program is in charge of creating different scheduling scenarios and generating statistic results, threads created by the program are managed by the EFQ scheduler that is implemented in the Linux OS of the computing platform. The benchmarks are employed to program the thread functions of the multithreading test-bench program. In the following sections, we will provide an expanded description of the computing platform, the benchmarks, and the multithreading test-bench program.

The power supply and measurement system is composed of three modules: a DC-DC converter, a battery emulator, and a power measurement unit. The DC-DC converter is contained in a portable Lithium-ion battery pack; it is used to boost the battery voltage output to a proper level of the computing platform. The battery emulator is composed of a programmable power source and a Labview-based battery simulation program; it can emulate the battery voltage output and is employed to replace a real Lithium-ion battery. The power measurement unit is internally contained in the programmable power supply device; it is used to profile the power consumption of the whole experimental platform, the measured voltage and current values are returned to the Labview program, through which the measuring frequency can be configured. A detailed description of the power supply and measurement system is provided in section 6.3.

6.1.2 Experimental methodology

The functioning of the EFQ scheduler relies on an online real-time feeding of the power information of each task or thread. Therefore, the ideal way to assess the Linux-based EFQ scheduler is to build a complete energy-centric system with the energy allocation and energy accounting module properly implemented.

However, build the whole energy-centric system is a huge and complex project, not to mention that there are several unsolved yet challenging problems in energy modeling and activity tracing.

In our test-bench that is shown in Figure 6.1, unfortunately, it is neither able to feed the power consumption of each task to the EFQ scheduler in real-time nor able to account the energy consumption to the correct thread. However, the test-bench still allows carrying out a simplified experimental work that solely focuses on the scheduling assessment. To achieve that, a two-phase experimental methodology as demonstrated in Figure 6.2 is employed in this work.

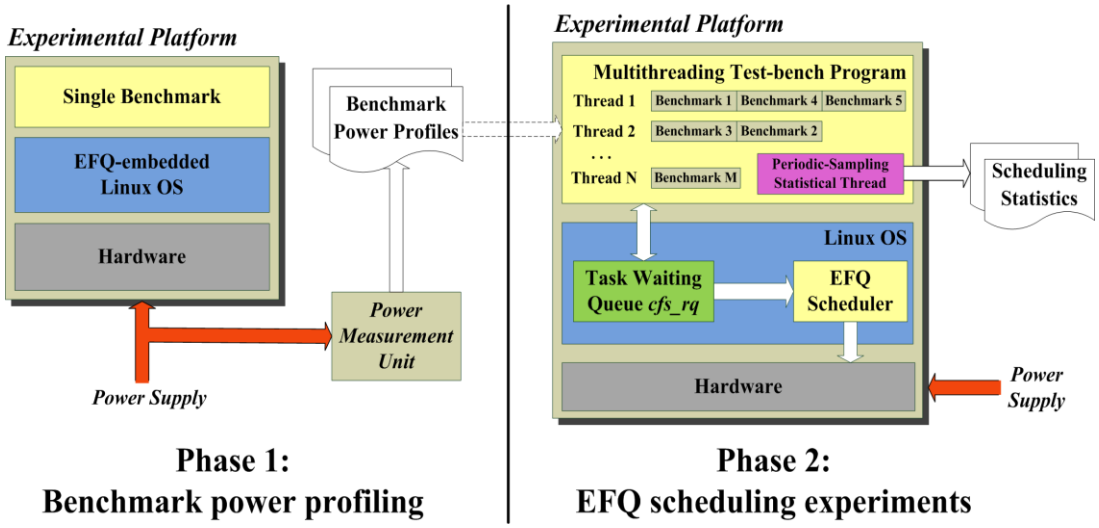


Figure 6. 2: Overview of the Experimental Methodology

While the first phase is focused on the power profiling of each benchmark, the second phase is where different EFQ scheduling experiments are carried out and statistic results are obtained. In the first phase, each benchmark is individually launched and run on the experimental platform; and in the meantime, the power consumption is profiled via the power measurement unit that periodically samples the voltage and current of the experimental platform. After repeating this process for each benchmark, we can obtain the power profiles of all benchmarks at the end of the first phase. In the second phase, we use the benchmarks and their power profiles to program the threads of the multithreading test-bench program, and run the test-bench

on the experimental platform to generate the scheduling results. In this way, the power of one thread is determined by the power profiles of the related benchmarks that are obtained in the first phase, and the power values are already fixed when the test-bench program is compiled. Therefore, online real-time power modelling and accounting is self-contained in the multithreading test-bench program, and there is no need to implement extra energy modelling and activity tracing mechanisms in the experimental platform. With this two-phase methodology, we can focus on the assessment of the Linux-based EFQ scheduler while avoid any interference from the modules of energy allocation and energy accounting.

6.2 Computing platform

The computing platform of the experiments is a commercial board named BeagleBoard [83] that runs the Angstrom Linux operating system [84] with the EFQ scheduler implemented in the kernel. In the remaining of this section, the experimental platform will be introduced from two aspects: the hardware environment and the software environment.

6.2.1 The hardware environment

The BeagleBoard is a low-power single-board computer that combines all the functionality of a basic personal computer. Many popular operating systems such as Android, Angstrom Linux, Ubuntu, Windows CE and RISC OS have been ported to the BeagleBoard. A general description of the BeagleBoard architecture is given in Figure 6.3.

As shown in Figure 6.3, the BeagleBoard shelters an OMAP 3530 system-on-a-chip (SOC), which includes a 720 MHz ARM Cortex-A8 CPU for general purpose computation and a TMS320C64x+ DSP for accelerated multimedia applications. Build-in storage and memory is provided for the OMAP 3530 SOC through a Package-On-Package (POP) chip that includes 256MB of NAND flash and 256MB of SDRAM. Additional memory can be added to the BeagleBoard by

installing a SD or MMC card in the SD/MMC slot, or driving a USB thumb drive or hard drive through the USB OTG port and the EHCI USB port. The TPS65950 is a power management chip (PMIC) that provides different power domains and clock frequencies to the BeagleBoard, its 5V power source can come from the USB OTG port connected to a PC powered USB HUB, or a 5V DC supply. Besides, TPS65950 also provides stereo audio in and out. The video output of the BeagleBoard is provided through a separate S-Video connector and a DVI-D connector that can partially support High-Definition Multimedia Interface (HDMI). In addition, BeagleBoard provides a RS-232 serial connector, a Join Test Action Group (JTAG) connector, and an expansion connector.

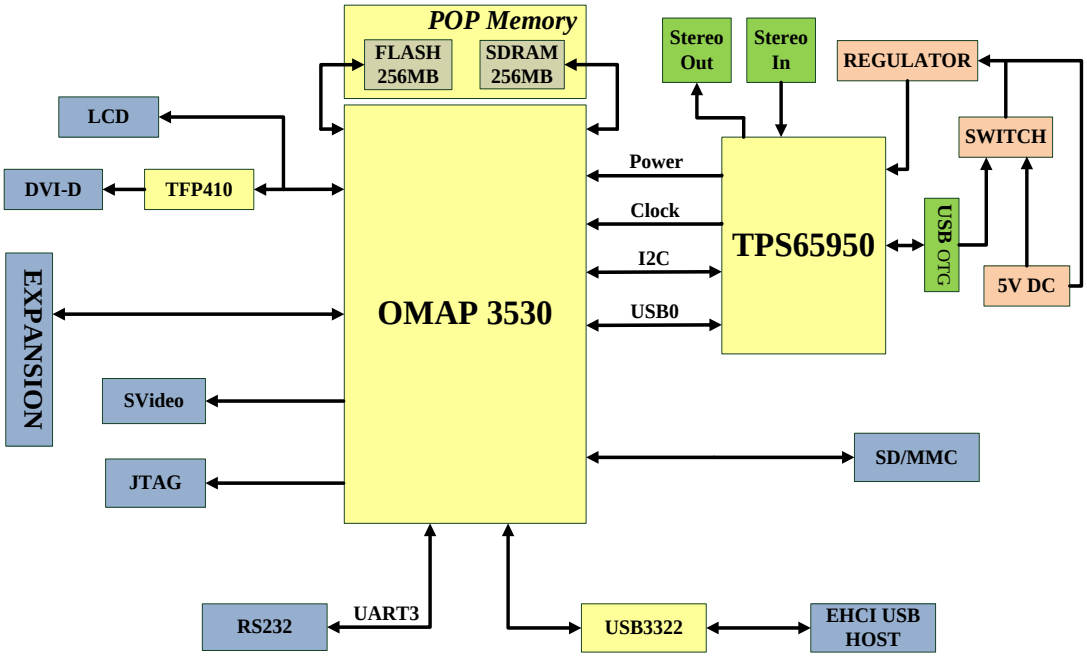


Figure 6. 3: Block Diagram of BeagleBoard

The functions of the BeagleBoard can be divided into four categories: computation, storage, I/O, and communication. Note that the communication unit and the I/O unit are combined together due to the lack of specific physical interface for network communication. But there are several USB to network adapters on the market that can add Ethernet, Wi-Fi, or Bluetooth connectivity to the BeagleBoard by using the EHCI USB port or the USB OTG port in the host mode.

In this thesis work, not all functions and devices of the BeagleBoard are employed for the experiments. To simplify the work and focus on the energy consumptions caused by the ARM Cortex-A8 CPU, the memory subsystem, and the related I/O buses, the BeagleBoard has been configured as a minimal system that disables the unnecessary components such as the display and network subsystems.

6.2.2 The software environment

The Angstrom Linux [84] is a special Linux distribution that is tailored for embedded systems and shipped with the BeagleBoard. A full package of the Angstrom distribution images includes an X-loader (MLO), a U-boot (u-boot.bin), a Linux kernel image (uImage), and a Linux root filesystem. To boot the Angstrom Linux on the BeagleBoard from SD card, the SD card has been formatted into two partitions, with the X-loader, U-boot and uImage held in the first partition and the Linux root filesystem held in the second partition. The procedure of Linux booting is as follows: when the BeagleBorad is powered on, the ROM program loads and executes the X-loader, which further loads the U-boot and executes it; the U-boot reads its commands and loads the Linux OS kernel image with the U-boot commands as arguments; once the kernel image is fully loaded to the memory, it is uncompressed and begins the initialization procedure; at certain point of the kernel initialization, the kernel mounts the root filesystem partition based on the U-boot commands; after the Linux OS is fully booted, a login interface appears and the system is ready for use.

In this work, the Linux kernel V3.3 has been employed to implement the EFQ scheduling algorithm. To obtain the EFQ-embedded Linux kernel image, the Linux kernel source codes with the EFQ implementation are configured for the BeagleBoard OMAP 3530 architecture and compiled with a GCC ARM cross compiler.

To set the minimum schedulable CPU time quantum for the EFQ scheduler, the Linux kernel V3.3 has been configured to use the 32 KHz timer of the BeagleBoard to generate a kernel internal timer frequency of 1000 Hz. A 1000 Hz kernel internal timer can produce a one millisecond-granularity scheduling tick and provide a maximum preemption delay of 1 millisecond.

The Linux kernel V3.3 can support the frequency and voltage scaling of the ARM Cortex-A8 CPU from the user space. In this work, the default frequency and voltage of the ARM CPU are set as 720 MHz and 1.35 V, respectively. The CPU frequency governor is set as *cpufreq_performance*, which will force the CPU to always use the highest possible clock frequency of 720 MHz.

6.3 Power supply and measurement system

As has been mentioned in the architecture overview of the test-bench, the power supply and measurement system consists of three functional modules: the battery emulator, the power measurement unit, and the DC-DC boost converter.

The block diagram of the power supply and measurement system is shown in Figure 6.4, and the device connections of the system are demonstrated in Figure 6.5.

The battery emulator and the power measurement system are built up together with a PC that executes a LabView-based battery simulation program [85] and an Agilent 66321D programmable power supply [86] that internally includes a digital voltmeter and a digital ammeter. The DC-DC boost converter is contained in a portable Lithium-ion battery pack called BeagleJuice [87].

The platform of battery emulation and power measurement can measure the power consumption of the device-under-test (DUT) while emulating the voltage output of a lithium-ion battery. Besides, it allows setting up the battery full-charge state instantaneously; therefore, in this thesis work, the battery emulator is employed to replace the lithium-ion battery of the BeagleJuice, which requires a time-consuming procedure of battery recharging. The power source to be connected to the DC-DC converter can be selected through the switcher of the BeagleJuice. Finally, the DC-DC converter can boost the battery voltage output to a level of 5V to power on the BeagleBoard.

To build the battery emulator and the power measurement system, the PC and the Agilent 66321D are connected through a USB to General-Purpose Interface Bus (USB-GPIB). The battery simulation program on the PC is designed in Labview 2009.

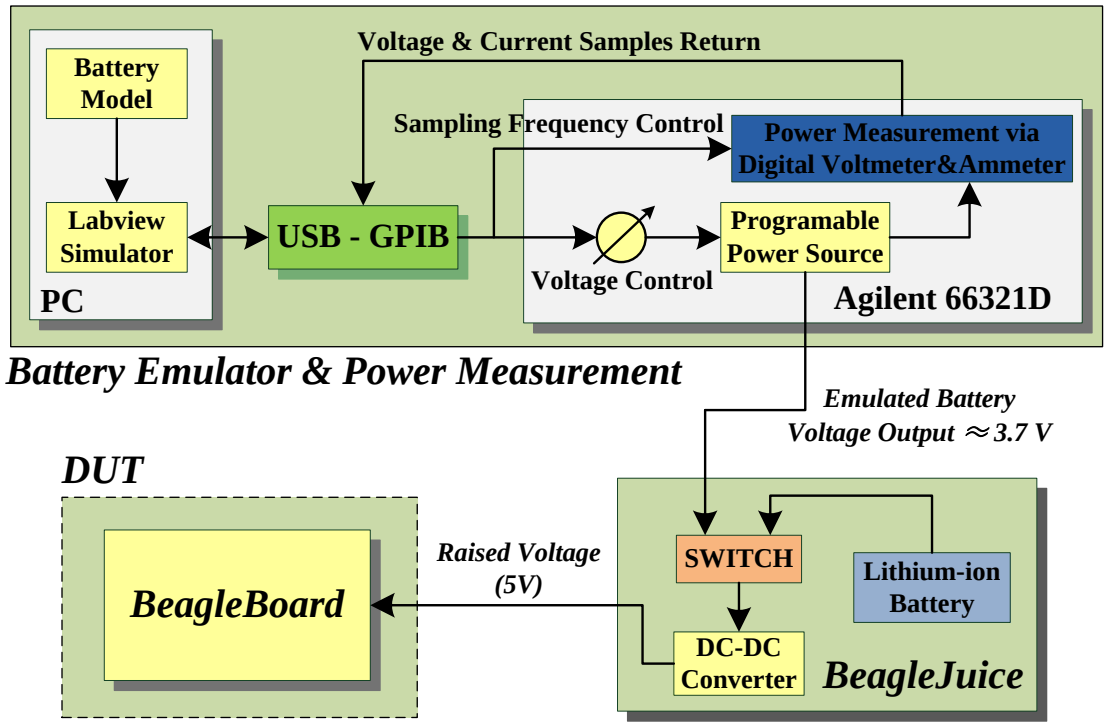


Figure 6. 4: Block Diagram of the Power Supply and Measurement System

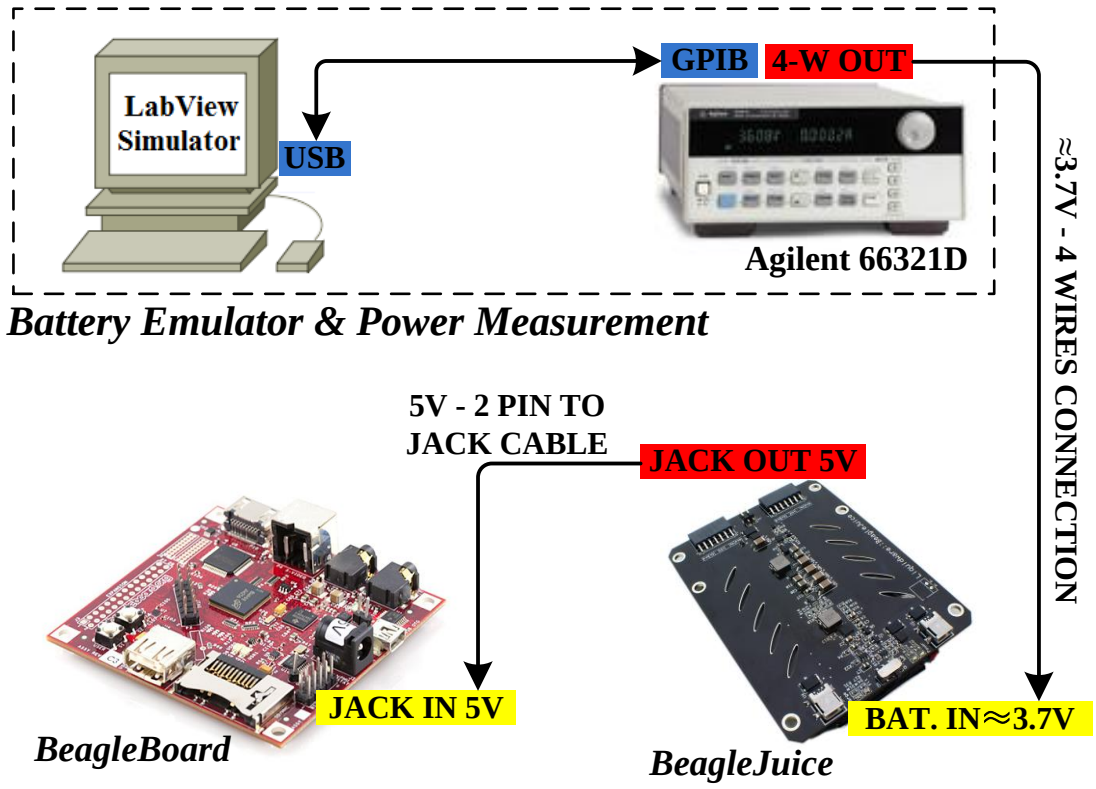


Figure 6. 5: Device Connections of the Power Supply and Measurement System

The LabView program takes the battery discharging model and simulates the battery voltage drop according to the battery discharge state along the time. The voltage values generated from the battery simulator are passed via the USB-GPIB to the voltage control module of the Agilent 66321D. The Agilent power supply outputs the emulated battery voltage to the DC-DC converter of the BeagleJuice and, in the meantime, meters the voltage and current values of the DUT with the internal-contained digital voltmeter and ammeter. Again, through the USB-GPIO interface, the LabView program on the PC can configure the measuring frequency of the digital meters and obtain the voltage and current samples from the Agilent 66321D.

The battery model of the emulator is built with the polynomial regression technique based on the pre-measured voltage samples of the discharging curve of a reference battery [85]. This model assumes a constant discharge current and achieves a mean error less than 2%. As drawbacks the model does not take into account the battery working temperatures, neither the working age nor the Peukert's law. On the whole, this power model is far enough for this dissertation work because its accuracy will not affect the power profiling of the benchmarks.

The Graphical User Interface (GUI) of the LabView-based battery emulator and simulator is shown in Figure 6.6. To start the battery emulation, a sample file of the reference discharging voltage curve (in blue) should be firstly loaded, and then the battery model (voltage curve in red) should be calculated based on the selected polynomial regression order; after that, the internal resistance and the maximum current of the battery should be introduced, and finally, the battery emulation is started by pressing the start button below the icon of battery charge indicator. At the middle bottom of the GUI, the DLOG option is available for observing the measured voltage and current in real-time and saving the historical data. At the right bottom of the GUI, the measuring period of the digital voltmeter and ammeter can be set via the USB-GPIB interface. For the power profiling of the benchmarks in this thesis work, the sampling period is set as 100 milliseconds, that is, 10 samples of voltage and current per second.

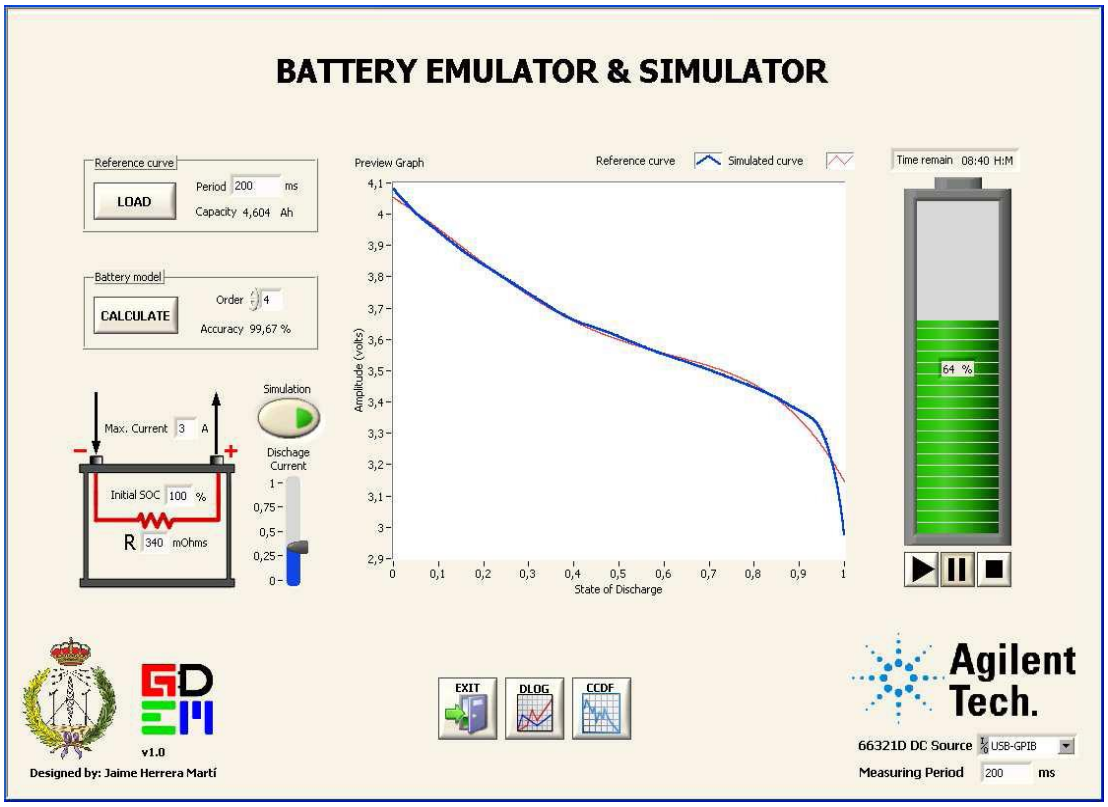


Figure 6. 6: GUI of the Battery Emulator and Simulator [85]

6.4 Benchmark characterization

An open-source and widely-referred embedded benchmark suite named MiBench [88] is employed to program the benchmarks for the experiments. The MiBench contains a set of 35 representative embedded programs of six categories, including Consumer devices, Telecommunications, Office automation, Automatic and Industrial control, Networking, and Security. The MiBench programs have been slightly modified to fit the multithreading test-bench program and the experimental purpose. For the experiments of this dissertation work, the following benchmarks are employed:

rt_fft: adapted from the *FFT* benchmark of MiBench, performs Fast Fourier Transform (FFT) every 200ms, modeling a CPU-intensive periodic real-time application. The amount of FFT work in each period is adjustable so that the benchmark can produce a fluctuating workload in each period.

int_stringsearch: adapted from the *stringsearc*h benchmark of MiBench, searches for given words in sentences every one second, modeling a CPU-intensive interactive application through which a user can send periodic requests on string searching. The amount of search work in each period is adjustable so that the benchmark can produce a fluctuating workload in each period.

batch_fft: continuously performs the Fast Fourier Transform (FFT) until the application terminates, modeling a CPU-intensive batch application.

batch_fft_io: first continuously performs the FFT in a CPU-intensive style as the *batch_fft*, and then repeatedly writes the FFT results into the SD card, modeling a batch application that consumes a large amount of energy on I/O operations.

batch_cubic: adapted from the *basicmath* benchmark of MiBench, continuously performs the cubic function solving, modeling a CPU-intensive batch application.

batch_isqrt: adapted from the *basicmath* benchmark, continously computes integer square roots, modeling a CPU-intensive batch application.

batch_rad2deg: adapted from the *basicmath* benchmark, continuously performs angle conversions from degrees to radians, modeling a CPU-intensive batch application.

batch_mix: performs the *batch_fft*, *batch_rad2deg* and *stringsearch* benchmark one by one, modeling a batch application that has different power consumptions along the time.

The above benchmarks are programed based on a number of basic computational components; they are *fft, fft_io, stringsearch, cubic*, *isqrt,* and *rad2deg*. To profile the power consumption of the experimental benchmarks, each basic computational component has been executed individually on the experimental platform, whose power is supplied and in the meanwhile measured by the battery emulator and power measurement system.

Table 6.1 lists both the total power and the active power of each basic computational component; the baseline power when the platform is running no benchmark is 1.188W. As can be seen, different CPU-intensive components can have various power consumptions depending on the specific codes that are executed. Notably, the *fft_io* component has the highest power (active power almost twice of the *cubic*) due to the extra power consumption caused by I/O operations to the SD card. Initially, it has an active power of 0.759 W that is almost the same as *fft*. The active power jumps to 1.402 W when an additional power of 0.643 W is caused by the I/O operations. Note that asynchronous energy consumptions caused by the I/O operations are accounted onto the CPU occupation time.

Table 6. 1: Power Profiles of the Basic Components of Benchmarks

	Total power (W)	**Active power* (W)**
fft	1.946	0.758
fft_io	1.947+0.643(I/O) = 2.59	0.759+0.643(I/O) = 1.402
stringsearch	2.259	1.071
cubic	1.929	0.741
isqrt	1.943	0.755
rad2deg	2.074	0.886

* System baseline power with no active benchmark is 1.188 W

Next, the benchmarks are characterized for the experiments based on the power profiles in Table 6.1. Since the total powers include the power consumptions of the DC-DC converter and the whole computing platform, the active powers are employed to determine the standard energy packet size of each benchmark. In this way, we exclude the baseline power of the experimental platform and focus on the power consumptions that are additionally caused by the benchmark activities on the CPU, memory and I/O buses. Note that this will neither destruct the energy model nor affect the scheduling assessment, because the system base power can be simply combined into the model by adding a constant value. Besides, for the sake of clarity and easy-analysis, the energy unit (Eu) and power unit (Pu) are employed to measure the energy consumption in the remaining of this dissertation. Specifically, one energy unit

(Eu) is defined as one micro joule (µJ); and one time unit (Tu) is defined as one millisecond (ms). Therefore, one power unit (Pu or Eu/Tu) equals one µJ/ms or one milliwatt (mW).

The characterizations of the experimental benchmarks are provided in Table 6.2 and Table 6.3; they are referred to build diverse scheduling experiments in the next chapter.

Table 6. 2: Characterization of Benchmarks with Constant Workload

	Period# (Tus)	**Workload# (Tus) / period**	**Power* (Pus)**	**Average power* (Pus)/ Period**
rt_fft	200	72 (36%)	758	758*36% = 273
int_stringsearch	1,000	193 (19.3%)	1071	1071*19.3% = 207
batch_fft	N/A	100%	758	758
batch_fft_io	N/A	100%	759+643(I/O) = 1402	759+643(I/O) = 1402
batch_cubic	N/A	100%	741	741
batch_isqrt	N/A	100%	755	755
batch_rad2deg	N/A	100%	886	886
batch_mix	N/A	100%	varies among 758, 886, 1071	varies among 758, 886, 1071

1 time unit (Tu) equals 1 millisecond (ms), 1 energy unit (Eu) equals 1 micro joule (µJ)

* 1 power unit (Pu) is defined as 1 Eu/Tu, equals 1 µJ/ms or 1 milliwatt (mW)

Table 6. 3: Characterization of Periodic Benchmarks with Variable Workload

	rt_fft	***int_stringsearch***
Power* (Pus)	758	1071
Period# (Tus)	200	1000
Average workload (Tus) / period	71	190
Average CPU utilization	35.5%	19.0%
Average power (Pus) / period	269	203
Worst-case workload (Tus) / period	109	312
Worst-case CPU utilization	54.5%	31.2%
Worst-case power (Pus) / period	413	334

1 time unit (Tu) equals 1 millisecond (ms), 1 energy unit (Eu) equals 1 micro joule (µJ)

* 1 power unit (Pu) is defined as 1 Eu/Tu, equals 1 µJ/ms or 1 milliwatt (mW)

In Table 6.2, benchmark *rt_fft* and *int_stringsearch* are characterized as periodic tasks with a constant workload of 36% and 19.3%, respectively; both of them cannot fully utilize the CPU because they will go idle once the work of one period is completed; while the other benchmarks are batch tasks that can utilize the CPU up to nearly 100%.

In Table 6.3, benchmark *rt_fft* and *int_stringsearch* are characterized as periodic tasks with a variable workload in each period. The variable workloads are produced by repeating the basic computational component (*fft* or *stringserach*) for a random number of times in each period. The random numbers are uniformly distributed. Based on the average value and maximum value of the random numbers, the average and worst-case workloads, CPU utilizations, and powers can be determined for the periodic benchmarks. In Table 6.3, the average workload of benchmark *rt_fft* and *int_stringsearch* are 35.5% and 19%, respectively. These values are slightly different from those in Table 6.2 due to unavoidable experimental errors.

6.5 Multithreading test-bench program

A multithreading test-bench program based on the POSIX-thread (Pthread) API has been developed to assess the Linux implementation of the EFQ scheduler. Figure 6.7 demonstrates the structure diagram of the Pthread-based test-bench and its interaction with the EFQ-embedded Linux kernel.

Besides of the system calls required by EFQ, two extra system calls, namely *sys_get_total_energy* and *sys_get_thread_energy*, have been implemented to support the test-bench. The system call *sys_get_total_energy* reads the total energy consumption of benchmarks from the kernel space and returns the value to the test-bench so that it knows when to terminate the program under an energy budget. The system call *sys_get_thread_energy* is used to periodically return the energy consumption of each benchmark and write the values into a statistics file.

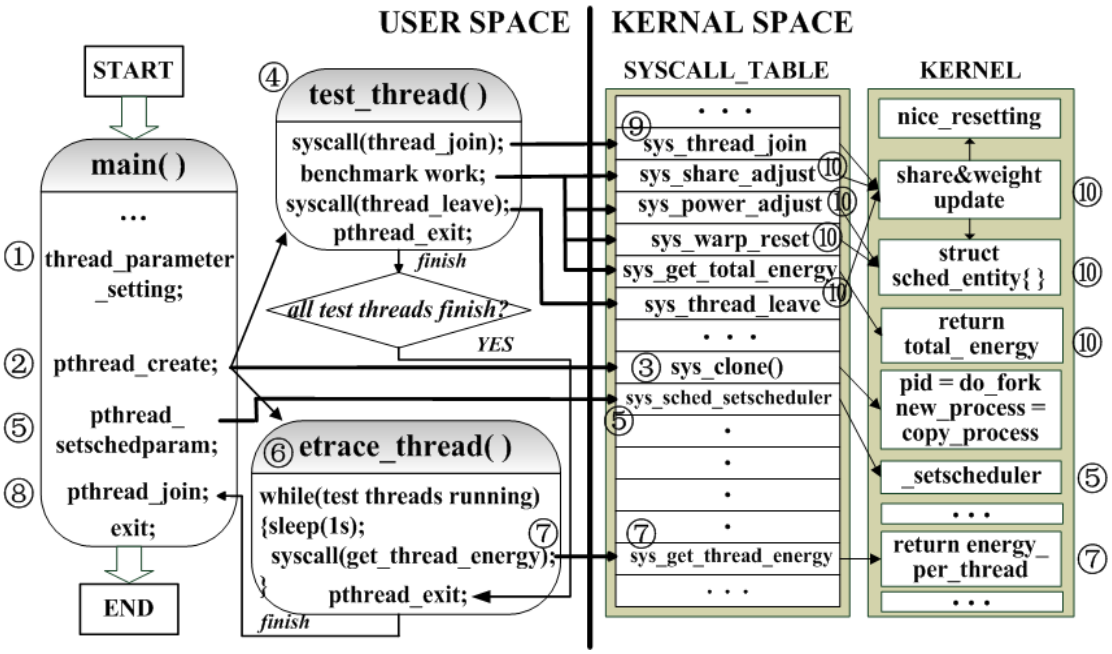


Figure 6. 7: Structure Diagram of the Multithreading Test-bench Program

The Pthread-based test-bench starts with a main thread, and the benchmarks are executed as child threads. After the characterizations of the child threads have been defined in the beginning of the main thread (Figure 6.7-①), the library function *pthread_create* is called to create child threads that execute their pre-defined works (Figure 6.7-②). The *pthread_create* can create a schedulable entity in the kernel space and return the PID to the main thread through the system call *sys_clone* (Figure 6.7-③). At this point two types of child threads, namely test thread and etrace thread, are created. Test threads (Figure 6.7-④) are those that execute the benchmark works under the EFQ scheduler. Once a test thread is created, the library function *pthread_setschedparam* is called by the main thread to set the scheduling policy (SCHED_EFQ), the energy packet size, and the warp parameters in the corresponding entities of the kernel (Figure 6.7-⑤). The *etrace* thread (Figure 6.7-⑥) is an energy statistic thread that is only created once. It is a high-priority and lightly-loaded thread that wakes up every second to read the energy consumption of each test thread through the system call *sys_get_thread_energy* (Figure 6.7-⑦); its effect to the EDF scheduling of test threads can be ignored. After all threads have been created, the

main thread will suspend itself and waiting for the termination of the child threads by calling the library function *pthread_join* (Figure 6.7-⑧). When a test thread begins executing, the system call *sys_thread_join* (Figure 6.7-⑨) is firstly called to update the share and weight in the kernel space and reset the nice value. During the benchmark work, diverse system calls can be used to adjust the EFQ parameters or frequently check the total energy consumption (Figure 6.7-⑩). If the energy budget of the test-bench is exhausted, all test threads are forced to exit even before finishing their benchmark works, this event will signal the exit of the *etrace* thread and finally wakes up the main thread to terminate the test-bench.

6.6 Summary

This chapter presents the experimental test-bench that is employed to verify the Linux-based implementation of the EFQ algorithm. The experimental test-bench is composed of two parts: the experimental platform and the power supply and measurement system. The experimental platform is a Linux computing platform that hosts a multithreading test-bench program. The Linux computing platform consists of an ARM-based embedded board and a specially-tailored Linux distribution with the EFQ algorithm implemented in the kernel. Both the hardware environment and the software environment of the Linux computing platform are described. The multithreading test-bench program is designed to create different scheduling scenarios and generate statistic experimental results; it is developed with the POSIX-thread (Pthread) API and an open-source benchmark suite is referred to program the threads under testing or scheduling. The multithreading test-bench program design and the related benchmarks are also introduced in this chapter. The power supply and measurement system is used to power the experimental platform and measure its energy consumption; it is composed of three functional modules: the battery emulator, the power measurement unit and the DC-DC converter, a description of these modules are also provided.

To avoid incurring the complexity of implementing the whole energy-centric system and focus on the assessment of the energy-centric scheduling, the energy accounting and energy allocation is not implemented in the experimental test-bench. Because of this limitation, a two-step approach is employed to build the EFQ scheduling experiments of this work. In the first step, the power consumption of each reference benchmark is profiled, and in the second step, the power profiles of the reference benchmarks are programed into the threads of the multithreading test-bench program so that the EFQ scheduler can work without a real-time and on-line accounting of the hardware energy consumption. This chapter also provides a description on the benchmark power profiling and the two-step experimental methodology. The next chapter will introduce the experiment design and discuss the experimental results.

Chapter 7

Experimental Results

To meet the requirements of energy-centric processor scheduling, the energy-based fair queueing (EFQ) algorithm should be able to achieve proportional power sharing, time-constraint compliance, and when necessary, a tradeoff between the task power share and time-constraint compliance. Based on the Linux-based EFQ implementation and the experimental test-bench, this chapter provides a more realistic and comprehensive evaluation of the EFQ algorithm and explores the potential of employing EFQ to optimize the mobile system user experience via specifically-designed experiments. The analysis and discussion of the experimental results are organized in three sections. Section 7.1 and 7.2 validates the properties of EFQ in providing proportional power sharing and time-constraint compliance, and section 7.3 explores the potential of employing EFQ to optimize the user experience of energy-limited mobile systems.

7.1 Maintaining proportional power sharing

This section evaluates the ability of EFQ in maintaining proportional power sharing among applications through two experiments. The first experiment checks whether EFQ can achieve a proportional sharing of the system-wide power, the Linux-CFS scheduler is employed as a reference. The second experiment verifies whether EFQ can protect the power share of specific applications in a dynamic environment.

7.1.1 Proportional sharing of the system-wide power

In Figure 7.1, benchmark *batch_fft_io* is scheduled against benchmark *batch_mix* with a 1:1 weight ratio; the power share of benchmark *batch_fft_io* under the EFQ scheduler is compared with the one under the Linux-CFS scheduler. In reference to Table 6.2 of section 6.4, the power of *batch_fft_io* changes from 759 Pus (*cpu only*) to

1402 Pus (*cpu and io)* at time 6 KTus; the power of *batch_mix* changes from 758 Pus (*fft*) to 886 Pus (*rad2deg, at 28 KTus*) and 1071 Pus (*stringsearch, at 52 KTus*).

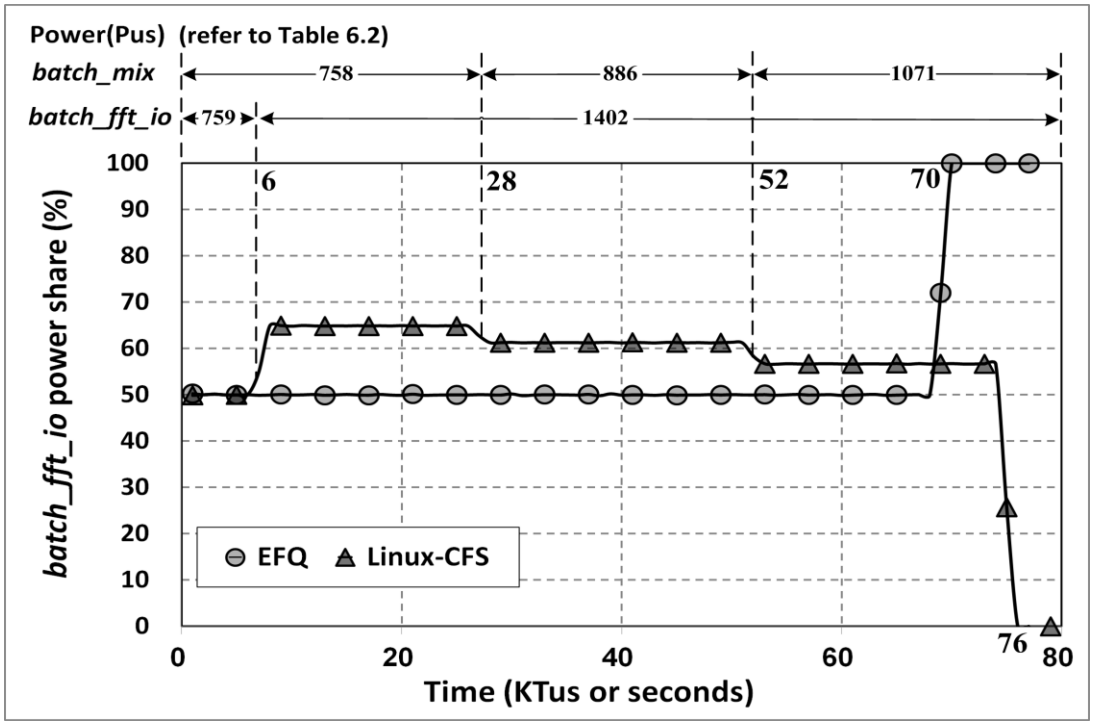


Figure 7. 1: Comparison of the System-Wide Power Share under EFQ and Linux-CFS

As can be observed, under the Linux-CFS scheduler, the power share of *batch_fft_io* varies when the benchmark power changes along the time. Initially, *batch_fft_io* is allocated a 50% power share due to the equal power of the two benchmarks; when *batch_fft_io* begins performing I/O operations at time 6 KTus, its power share increases to 65%; later, the share drops to 61% at time 28 KTus and 57% at time 52 KTus as the power of *batch_mix* increases. The Linux-CFS scheduler achieves proportional share of the CPU, but totally ignores the energy consumption on I/O operations. Therefore, *batch_fft_io* is allowed to take a power share that is always larger than 50%; at time 76 KTus, it finishes the work before *batch_mix*.

Under the EFQ scheduler, however, no matter how the power changes, the power share of *batch_fft_io* is constantly maintained at 50% before *batch_mix* leaves the energy competition at time 70 KTus. This is because EFQ computes the starting energy tag by accounting the energy consumption as a global resource, regardless of in which device the energy is spent. One task, such as *batch_fft_io*, will be delayed in

the CPU dispatching if it consumes a large amount of energy on I/O operations; this helps to maintain a user-desired proportional share of the system power.

7.1.2 Power share protection

An EFQ scheduler should protect the power share of some specific tasks upon the change of the scheduling environment. Figure 7.2 shows the EFQ scheduling results when the task set under scheduling is dynamically changed and the total available energy is limited at 80,000 KEus. The benchmark characterizations for this experiment are given in Table 7.1, the readers are advised to refer Table 6.2 of section 6.4 for more details on the selected benchmarks.

Table 7. 1: Benchmark Characterizations for Power Share Protection Experiment

#	*rt_fft*	*int_stringsearch*	*batch_cubic*	*batch_rad2deg*
Power (Pus)	758	1071	741	886
Total energy request (KEus)	23496	11776	46045	21621
Reserved share	0.3	0.15	0	0
Initial weight	0	0	1	10
User preference	high	high	low	middle

Refer to Table 6.2 of section 6.4 for more details of the selected benchmarks

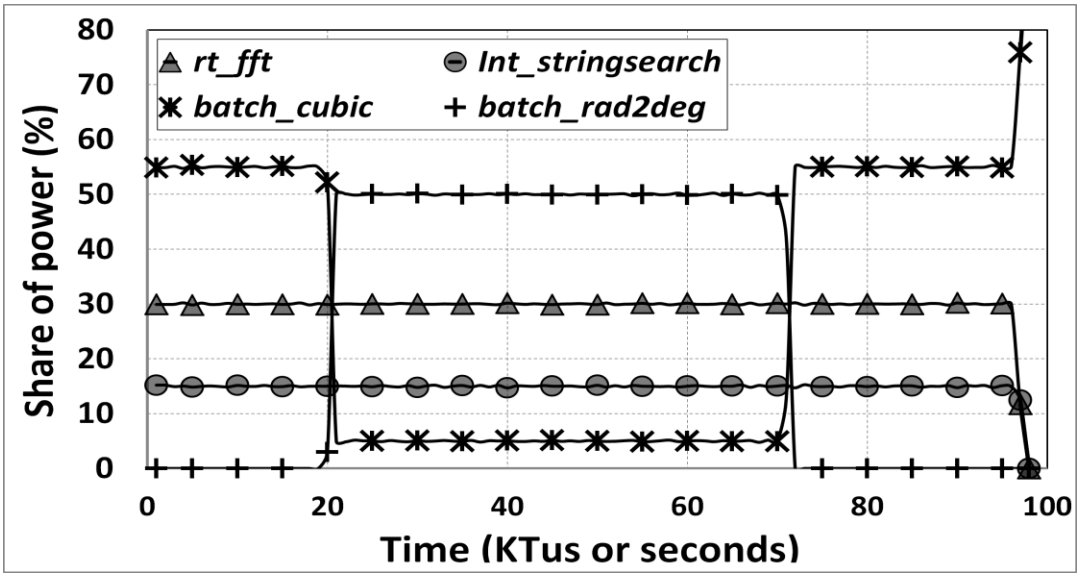


Figure 7. 2: Power Share Protection under EFQ

In Table 7.1, the total energy request is obtained by running each benchmark individually on the test-bench. The benchmark *rt_fft* and *int_stringsearch* are assumed as the most user-preferred applications, they are expected to finish the work before power off; therefore, those two benchmarks are reserved a power share of 30% and 15%, respectively, to ensure the meeting of their total energy requests. The benchmark *batch_cubic* is a background task, it is the least user-preferred task, and therefore, is only assigned an initial weight of 1. Finally, the benchmark *batch_rad2deg* is a short and urgent task that is launched in the middle of the experiment (20 KTus later). To accelerate the execution, *batch_rad2deg* is assigned a weight of 10 to give priority on energy competition with *batch_cubic*; however, *batch_rad2deg* should by no means reduce the power shares of *rt_fft* and *int_stringsearch*.

Note that the power share reservation in this experiment is only to show the power protection under EFQ. A higher power share is required by *rt_fft* and *int_stringsearch* if the time constraints are considered; this issue will be discussed in the next section.

The scheduling results in Figure 7.2 verify the power protection under EFQ; they match the ones that are observed under the SystemC-based and LinSched-based simulation test-benches. As can be observed, the power shares of benchmark *rt_fft* and *int_stringsearch* are constantly near 30% and 15%, respectively, without being affected by the launch and leave events of *batch_rad2deg*. Therefore, the two most user-preferred benchmarks are guaranteed to finish the work before the energy budget is exhausted at 98 KTus. As for the benchmark *batch_rad2deg*, upon its joining to the energy competition at 20 KTus, a power share of 50% is immediately allocated to enable a fast execution and early complication (at 72 KTus). The benchmark *batch_cubic* is executed as the least preferred task; although it can take the whole remaining power share of 55% when *batch_rad2deg* is idle, its power share is reduced to a level as low as 5% between 20 KTus and 72 KTus; thus, it is not guaranteed to finish the work before power off.

As the above experimental results demonstrate, energy management in EFQ is straightforward; this enables the achievement of advanced energy goals (such as the one in Figure 7.2) that are impossible in Linux-CFS and other schedulers.

7.2 Time-constraint compliance

This section evaluates the time-constraint compliance under the SEFQ and BSEFQ scheduling algorithms. The benchmark *rt_fft* and benchmark *int_stringsearch* are run against batch applications, and their real-time performances are assessed based on the number of deadline misses as well as the response time.

The remaining of this section is separated into two parts. The first part evaluates the basic ability of EFQ in achieving time-constraint compliance; the time-sensitive benchmarks, *rt_fft* and *int_stringsearch,* are characterized with variable workloads. The second part evaluates the robustness of achieving time-constraint compliance with EFQ, upon the change of energy estimation error and task number; for simplicity, *rt_fft* and *int_stringsearch* are characterized with constant workloads.

7.2.1 Time-constraint compliance under variable workloads

7.2.1.1 Experimental task characterizations

To evaluate the performance of time-sensitive tasks with fluctuating workloads, the task characterizations of Table 7.2 are employed in this experiment. Both *rt_fft* and *int_stringsearch* are time-sensitive tasks with a variable workload in each period. Specifically, the benchmark *rt_fft* has an average CPU utilization of 35.5% and a worst-case CPU utilization of 54.5%; the benchmark *int_stringsearch* has an average CPU utilization of 19% and a worst-case CPU utilization of 31.2%. The readers are advised to refer to Table 6.3 for more detailed benchmark characterizations.

Unlike batch tasks whose power share can be infinitely close to one, the maximum long-term power share and worst-case power share (definitions refer to section 3.3) of periodic tasks are limited to certain thresholds. These share thresholds can be computed based on the power and CPU utilization of the benchmarks.

Table 7. 2: Benchmark Characterizations with Variable Workloads

#	*rt_fft*	*int_ stringsearch*	*batch_ cubic*	*batch_ isqrt*
Power (Pus)	758	1,071	741	755
Period (Tus)	200	1,000	N/A	N/A
Average CPU utilization	35.5%	19.0%	100%	100%
Long-term average power (Pus)	269	203	741	755
Maximum long-term power share	0.333	0.251	$\lim(x \to 1)$	$\lim(x \to 1)$
Worst-case CPU utilization	54.5%	31.2%	100%	100%
Worst-case power (Pus)	413	334	741	755
Worst-case power share	0.509	0.393	$\lim(x \to 1)$	$\lim(x \to 1)$

Refer to Table 6.2 and Table 6.3 of section 6.4 for more details of the selected benchmarks

According to Table 7.2, the long-term average power of *rt_fft* and *int_stringsearch* is 269 Pus and 203 Pus, respectively; and based on the average CPU utilization of the two periodic time-sensitive tasks, the (remaining) average CPU utilization of all batch benchmarks is 0.455. Because the weight ratio of benchmark *batch_cubic* and *batch_isqrt* can vary over the time, the average power of all batch tasks is in the range of:

$$[741 \times 0.455, 755 \times 0.455] = [337,344]\ Pus$$

For simplicity, the lowest power value of 337 Pus is conservatively (and practically) referred to compute the maximum long-term power share and the worst-case power share for periodic benchmarks. Specifically, the maximum long-term power shares of benchmark *rt_fft* and *int_stringsearch* are:

$$\frac{269}{269 + 203 + 337} = \frac{269}{809} = 0.333, \qquad and$$

$$\frac{203}{809} = 0.251, \qquad respectively.$$

The worst-case power share of a periodic benchmark is practically computed based on its worst-case power and the average power of other benchmarks (refer to section 3.3).

When benchmark *rt_fft* has a worst-case power of 413 Pus, the average power of benchmark *int_stringsearch* is 203 Pus, and the average power of the two batch tasks is at least:

$$(1 - 0.545 - 0.19) \times 741 = 196 \text{ Pus.}$$

Thus, the worst-case power share for benchmark *rt_fft* is around:

$$\frac{413}{413 + 203 + 196} = \frac{413}{812} = 0.509.$$

When benchmark *int_stringsearch* has the worst-case power of 334 Pus, the average power of benchmark *rt_fft* is 269 Pus, and the average power of both batch tasks is at least:

$$(1 - 0.355 - 0.312) \times 741 = 247 \text{ Pus.}$$

Thus, the worst-case power share of benchmark *int_stringsearch* is around:

$$\frac{334}{334 + 269 + 247} = \frac{334}{850} = 0.393$$

Note again that the above values of the maximum long-term power share and the worst-case power share are all conservative ones that are computed in reference to the power of benchmark *batch_cubic*, the exact values are slightly smaller due to the existence of benchmark *batch_isqrt*.

7.2.1.2 Experimental results analysis

Table 7.3 shows the scheduling results under SEFQ and BSEFQ. The results are based on the averaging of 10 sets of experimental data. Benchmark *rt_fft* has a total number of 500 periods, and benchmark *int_stringsearch* has a total number of 100 periods. Note that, because the resolution of power share assignment is 0.01 according to the *niceness* table (refer to Figure 5.7 of section 5.1.2), the reserved power shares in Table 7.3 cannot be exactly the same as the values of the maximum long-term power share and the worst-case power share of benchmark *rt_fft* and *int_stringsearch*.

Table 7. 3: Time-constraint Compliance under Variable Workloads

	rt_fft **reserved share**	*int_stringsearch* **reserved share**	**Warp value**	*rt_fft* **Num. deadline misses**	*int_stringsearch* **Mean response time (Tus)**	**Max. response time (Tus)**
SEFQ$_1$	0.34	0.26	N/A	179	1,439	4,227
SEFQ$_2$	0.51	0.40	N/A	0	401	916
SEFQ$_3$	0.50	0.39	N/A	1	426	970
BSEFQ$_1$	0.34	0.26	*rt_fft* > *int_stringsearch*	0	260	587
BSEFQ$_2$	0.34	0.26	*rt_fft* < *int_stringsearch*	55	170	335

In the first scheduling test under SEFQ (SEFQ$_1$), both benchmark *rt_fft* and *int_stringsearch* are reserved a power share that is close to the maximum long-term power share. Although all of their energy demands are met in the long-term, the performances of time-sensitive tasks are very poor. Benchmark *rt_fft* misses 179 deadlines out of 500 periods. In the case of benchmark *int_stringsearch*, the mean response time is about 450 Tus larger than its period of 1,000 Tus, and the maximum response time is around four times larger than the period.

The real-time performances are largely improved in the second scheduling test under SEFQ (SEFQ$_2$), in which benchmark *rt_fft* and *int_stringsearch* are both reserved a power share that is close to the worst-case power share. In this case, benchmark *rt_fft* meets all deadlines and benchmark *int_stringsearch* achieves a maximum response time that is less than 1,000 Tus.

However, the scheduling delay under SEFQ is easily affected by the reserved power shares of the time-sensitive tasks. As shown in the third scheduling test under SEFQ (SEFQ$_3$), when the reserved power share is slightly smaller than the worst-case power share of *rt_fft* and *int_stringsearch*, benchmark *rt_fft* experiences the risk of missing a few deadlines (deadline misses range from 0 to 3 in 10 sets of scheduling results) and benchmark *int_stringsearch* experiences an increase of the response time (maximum response time is approaching its period of 1,000 Tus). Therefore, the reserved power shares for time-sensitive tasks should be conservatively and carefully determined for ensuring strict time-constraint compliance under SEFQ.

Under BSEFQ, benchmark *rt_fft* and *int_stringsearch* are both reserved a power share that is close to the maximum long-term power share, and the real-time performances are dependent on the priorities (indicated by the warp value) of time-sensitive tasks.

In the first scheduling test under BSEFQ ($BSEFQ_1$), benchmark *rt_fft* is assigned a higher warp value to allow its energy requests being served immediately after beginning a new period. Therefore, its time-constraint compliance is strictly guaranteed. In the case of benchmark *int_stringsearch*, its response time is significantly better than the ones under SEFQ. This is because when being warped with a lower warp value, the energy requests of benchmark *int_stringsearch* are continuously scheduled right after the ones of benchmark *rt_fft* and no energy request of batch tasks is scheduled ahead of it.

In the second scheduling test under BSEFQ ($BSEFQ_2$) where the interactive task is assigned the highest warp value, benchmark *int_stringsearch* achieves the optimal response time while benchmark *rt_fft* misses around 11% of its deadlines. This is because benchmark *int_stringsearch* has a longer period and its energy requests are always scheduled ahead of those of benchmark *rt_fft*; a deadline may be missed each time benchmark *int_stringsearch* begins a new period.

Anyhow, the time and the number of deadline misses are predictable in the above case. A soft real-time application, such as video decoder, can abandon the decoding work of one frame that is going to miss its deadline and replace the current frame with the previous one that has already been decoded, this helps to minimize user experience degradation and reduce unnecessary energy expenditures.

Based on the above experimental results, we can reach the same conclusion as in the simulation: BSEFQ is more flexible and effective in supporting different types of time-sensitive tasks in comparison with SEFQ. Under BSEFQ, if benchmark *int_stringsearch* is the most user-preferred application and a maximum response time that is less than 500 Tus is required by the user, it should be assigned the highest priority (as in $BSEFQ_2$) to achieve the optimal response time at the cost of missing a few predictable deadlines in benchmark *rt_fft*.

In a more general case, if a response time that is less than the period of benchmark *int_stringsearch* is acceptable by the user, the highest priority should be assigned to the benchmark *rt_fft* which has a smaller period (in a rate-monotonic manner, refer to section 3.4.4), so that all benchmarks can finish their work before the beginning of the next period (as in $BSEFQ_1$). Note that benchmark *int_stringsearch* can also be seen as a task that has periodic deadlines.

To maintain the long-term proportional fairness under BSEFQ, each time-sensitive task is reserved a share that approaches its maximum long-term power share; therefore, the total share reserved for time-sensitive tasks is around 0.6 in Table 7.3. The reserved power share does not need to be accurately the value of the maximum long-term power share because the scheduling latency under BSEFQ is dominated by the warp mechanism.

Under SEFQ, however, the deadline misses are not as predictable as the ones under BSEFQ, and the scheduling latency is sensitive to the reserved power share. As a result, the power shares of time-sensitive tasks have to be overly and carefully reserved to ensure the compliance of time constraints. According to Table 7.3, although the total power share of benchmark *rt_fft* and *int_stringsearch* is around 0.6 in average, a total power share of at least 0.91 has to be reserved under SEFQ, leaving a very small power share that is less than 0.09 for later-joined time-sensitive tasks.

7.2.2 Robustness of time-constraint compliance

7.2.2.1 Experimental task characterizations

To evaluate the robustness of time-constraint compliance under EFQ scheduling algorithms, the task characterizations in Table 7.4 are employed in this experiment. Without loss of generality, the periodic benchmarks are characterized with constant workloads over the time. The readers are advised to refer to Table 6.3 for more detailed benchmark characterizations.

Table 7. 4: Benchmark Characterizations for Robustness Validation of Time-constraint Compliance

#	rt_fft	int_stringsearch	batch_cubic	batch_isqrt
Energy packet size (Eus)	758	1071	741	755
Period (Tus)	200	1,000	N/A	N/A
CPU utilization	36%	19.3%	100%	100%
Long-term average power (Pus)	273	207	741	755
Max. long-term & worst-case share	0.336	0.255	$\lim(x \to 1)$	$\lim(x \to 1)$

Refer to Table 6.2 of section 6.4 for more details of the selected benchmarks

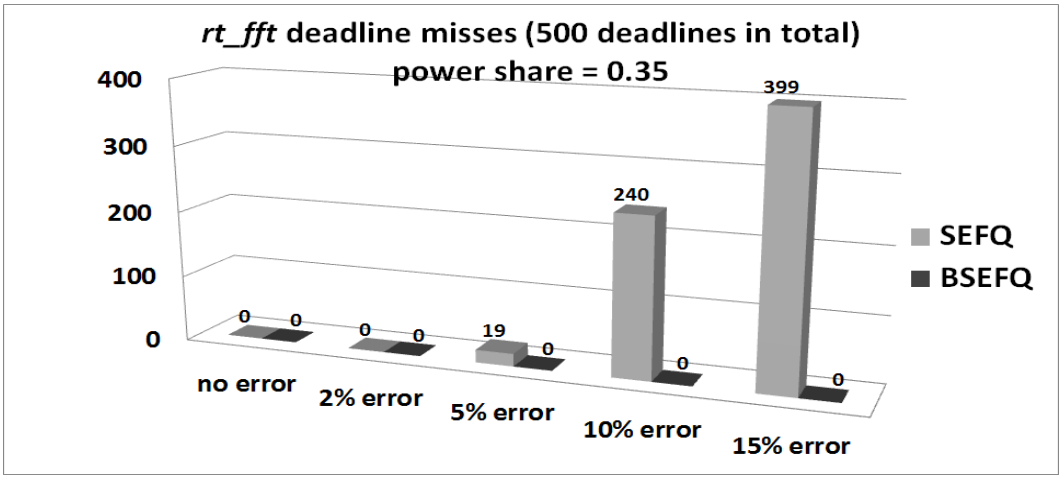
As usual, the maximum long-term power share of periodic tasks can be computed based on the power and CPU utilization of the benchmarks. For benchmark *rt_fft* and *int_stringsearch*, the maximum long-term power share is 0.336 and 0.255, respectively. To ensure strict time-constraint compliance under SEFQ, a time-sensitive task should be reserved a power share that is larger than its worst-case power share, which equals the maximum long-term power share in this case due to the constant workloads that are employed. In this experiment, benchmark *rt_fft* and *int_stringsearch* are reserved a power share of 0.35 and 0.26, respectively.

7.2.2.2 Results analysis: robustness upon energy estimation error

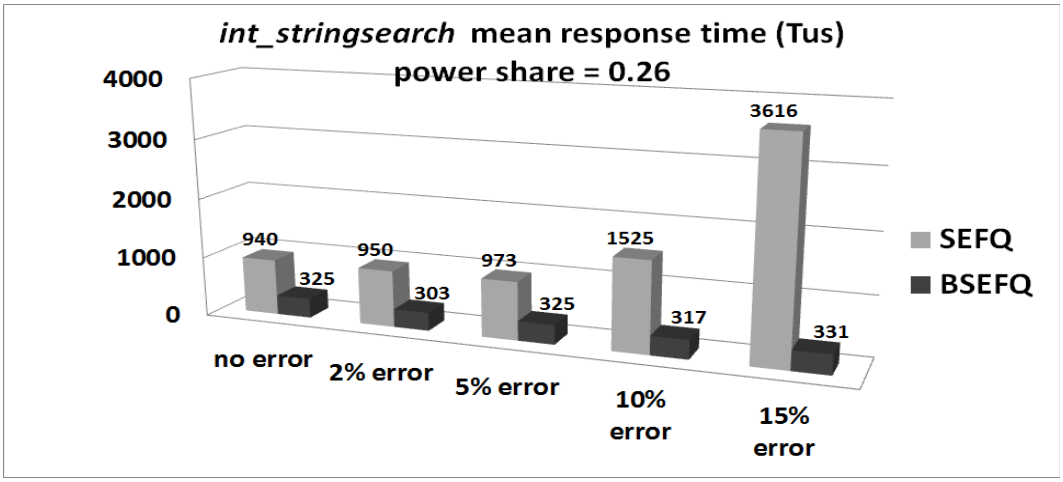
EFQ computes the starting energy tags based on the energy packet size (energy consumption during an allocated time quantum, refer to section 3.2), which should be estimated in the real-system scheduling. The error between the actual energy packet size and the estimated one is called energy estimation error.

Figure 7.3 compares the performance of *rt_fft* and *int_stringsearch* under SEFQ and BSEFQ when there are different levels of energy estimation error. As can be observed, both SEFQ and BSEFQ can achieve a good real-time performance when the energy estimation error is low (less than 2%). Especially, BSEFQ provides strict time-constraint compliance and a robust response time that are unaffected by the energy estimation error. This is benefited from the warp mechanism, with which the benchmark *rt_fft* and *int_stringsearch* can be immediately dispatched after a new

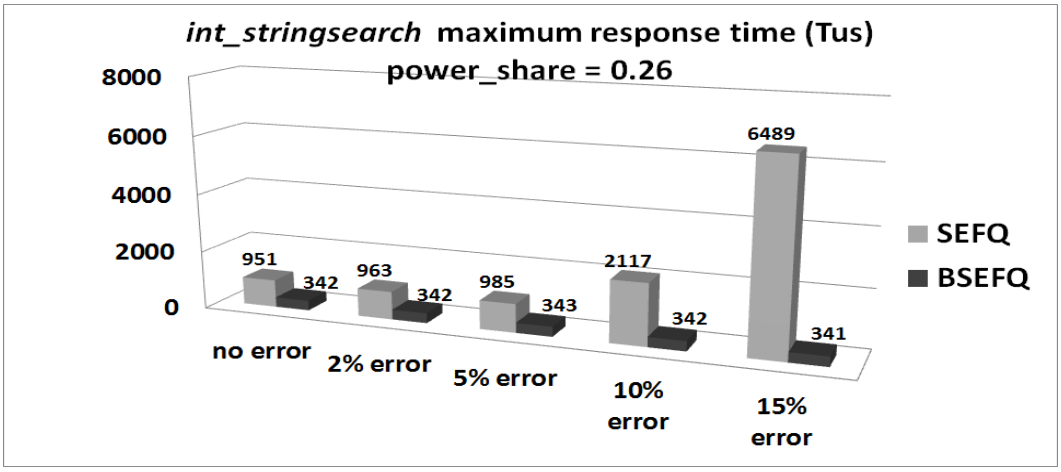
period begins. However, the real-time performance under SEFQ is non-robust to the energy estimation error. When the error increases over 10%, a great number of



a) *rt_fft* deadline misses



b) *int_stringsearch* mean response time



c) *int_stringsearch* maximum response time

Figure 7. 3 : Real-time Performances upon Different Levels of Energy Estimation Error

deadline misses in *rt_fft* and a significant increase of the response time in *int_stringsearch* are observed.

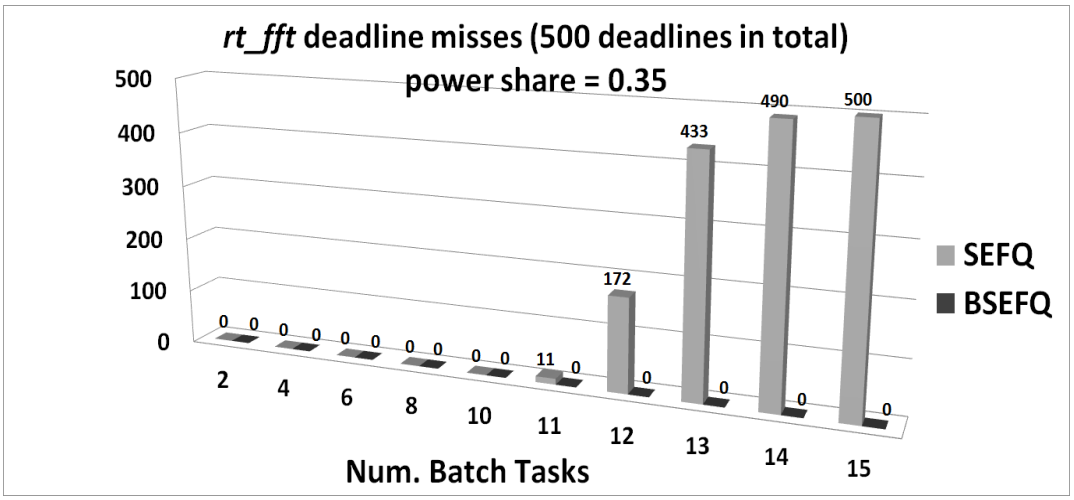
7.2.2.3 Results analysis: robustness upon task number change

Under the EFQ scheduler, the performance of *rt_fft* and *int_stringsearch* may also be affected by the number of active tasks that are competing for the system resources. In the next experiment, the number of background batch tasks is gradually increased to assess the robustness of time-constraint compliance under SEFQ and BSEFQ. The scheduling results are compared in Figure 7.4.

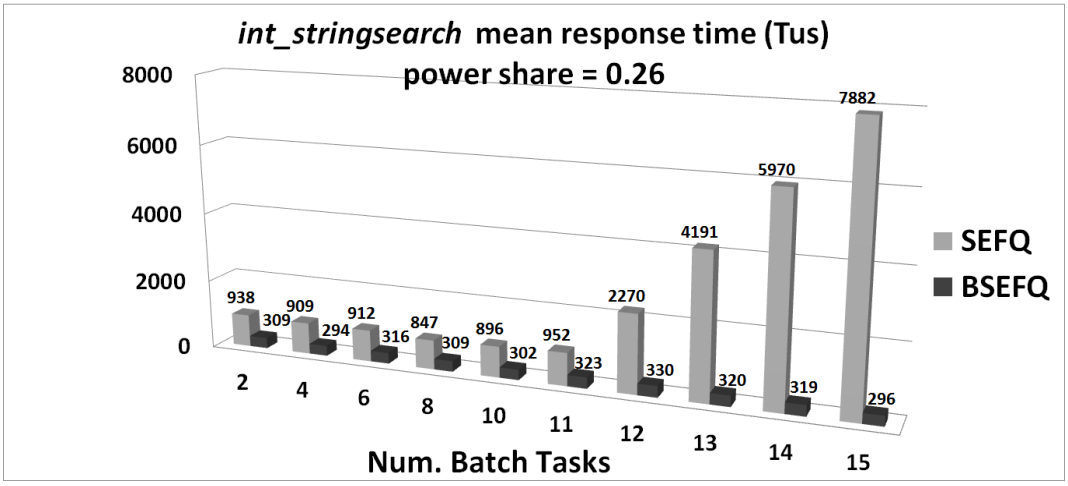
As can be observed, a good and stable real-time performance is achieved both under SEFQ and BSEFQ when the number of background batch tasks is smaller than 10. Especially, due to the scheduling priority given to the time-sensitive tasks with the warp mechanism, stringent time-constraint compliance and a robust response time are achieved under BSEFQ regardless of the batch task number.

Under SEFQ, however, the real-time performance is significantly deteriorated when the number of batch tasks is over 11. This is because the dispatch delay bound and the scheduling window (the length of time within which each task is scheduled once) under SEFQ can increase with the number of active tasks. When the scheduling window is small, time constraints can be met with a conservative power share reservation; but when the scheduling window increases to a considerable length, the real-time performance of benchmark *rt_fft* and *int_stringsearch* will be greatly deteriorated.

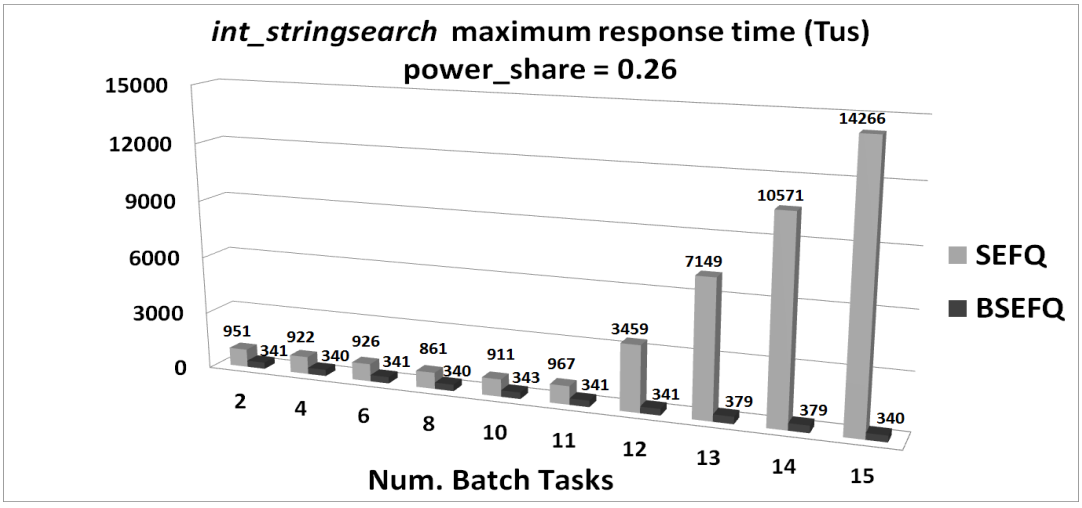
One simple yet effective solution to enhance the robustness of SEFQ is to increase the reserved power share for time-sensitive tasks; however, this method can causes over-reservation problems and thus is only applicable when the CPU workload is moderate. Instead, BSEFQ can support more stringent and robust time-constraint compliance upon energy estimation errors and task number variations without incurring the over-reservation problem.



a) *rt_fft* deadline misses



b) *int_stringsearch* mean response time



c) *int_stringsearch* maximum response time

Figure 7. 4: Real-time Performances upon Different Number of Background Batch Tasks

7.3 User experience optimization under energy limit

This section aims to experimentally explore the potential of employing EFQ algorithms in optimizing the user experience of energy-limited systems. To achieve the goal, it is firstly required to set up some assumptions on the experimental system. In section 7.3.1, we characterize the energy loads of the experimental benchmarks and build assumptions on the task user preference and energy allocation. Then in section 7.3.2, we explore the user experience optimization with EFQ under the experimental assumptions of section 7.3.1.

7.3.1 Experimental assumptions and task characterizations

For the experiments of this section, the task characterizations in Table 7.5 are employed. Time-sensitive benchmark *rt_fft* and *int_stringsearch* are run against batch benchmark *batch_cubic* and *batch_fft_io*. Without loss of generality, the periodic time-sensitive benchmarks are characterized with constant workloads (refer to Table 6.2 of section 6.4). The power of benchmark *batch_fft_io* is initially 759 Pus, it can increase to 1,402 Pus when there are extra I/O activities. The maximum long-term power share of periodic time-sensitive benchmarks varies with the average power of the two benchmarks, which is at least 741 Pus (when the weight ratio of *batch_cubic* to *batch_fft_io* is infinitely large). Therefore, according to Table 7.4, the maximum long-term power shares of benchmark *rt_fft* and *int_stringsearch* are respectively less than 33.6% and 25.5%.

Table 7. 5: Benchmark Characterizations for Demonstration on User Experience Optimization

#	*rt_fft*	*int_ stringsearch*	*batch_ cubic*	*batch_ fft_io*
Power (Pus)	758	1,071	741	[759, 1,402]
Period (Tus)	200	1,000	N/A	N/A
CPU utilization	36%	19.3%	100%	100%
Long-term average power (Pus)	273	207	741	[759, 1,402]
Max. long-term power share	< 33.6%	< 25.5%	100%	100%

Refer to Table 6.2 of section 6.4 for more details of the selected benchmarks

The experiments of this section are based on the following assumptions. First, it is assumed that the residual energy in the battery is 1,200,000 KEus (1,200 joules) and the user-desired target lifetime of the system is 1,600 KTus (1,600 seconds ≈ 27 minutes). Then, it is assumed that benchmark *int_stringsearch* is the most user-preferred task, its user-acceptable response time is 500 Tus, within which the user can hardly perceive the delay; benchmark *rt_fft* is the second user-preferred task, its acceptable deadline miss ratio is 10%; the two batch benchmarks are the least user-preferred tasks, their performance is solely measured by the amount of finished work.

To achieve a battery lifetime of 1,600 KTus for the system, the target lifetime is divided into 20 epochs, with each epoch lasting 80 KTus (80 seconds) and containing 60,000 KEus energy. It means an amount of 60,000 KEus energy is infused to drive the execution of tasks in each 80 KTus epoch. The 60,000 KEus energy should be properly allocated to the different benchmarks based on their energy requirements as well as the user preferences.

Table 7.6 summarizes the user preference and energy allocation over the benchmarks. As the most user-preferred task, benchmark *int_stringsearch* is expected to finish 80 periods of work and is estimated to consume 16,560 KEus energy, because its period is 1,000 Tus and its average power is 207 Pus. Considering that the average power of a benchmark can have slight deviation in the real system, a total of 16,860 KEus energy (with a margin of 300 KEus) is reserved to benchmark *int_stringsearch* to ensure its normal execution during each epoch. As the second preference, benchmark *rt_fft* is expected to finish 400 periods of work and is estimated to consume 21,840 KEus energy, considering its period of 200 Tus and its average power of 273 Pus. Therefore, a total of 22,140 KEus energy is reserved to benchmark *rt_fft*. Finally, the remaining 21,000 KEus energy is allocated to benchmark *batch_cubic* and *batch_fft_io* in a ratio of 1:2 based on the user preference.

Note that the average powers of benchmark *rt_fft* and *int_stringsearch* in the experiments are assumed as constant in different epochs. In a system where the

average powers of periodic time-sensitive tasks are variable over epochs, the energy allocation among benchmarks may have to be adjusted in different epochs.

Based on the above assumptions, the demonstration of the user experience optimization of an energy-limited system can be made on the basis of one epoch. In the remaining of this section, all results are analyzed on an epoch basis.

Table 7. 6: Characterization of User Preference and Energy Allocation

	rt_fft	***int_stringsearch***	***batch_cubic***	***batch_fft_io***
User preference	high	highest	low	middle
User requirement	miss ratio < 5%	response time < 500 Tus	As much as possible	As much as possible
Estimated energy request (KEus)	21,840	16,560	N/A	N/A
Energy allocation# (KEus)	22,140	16,860	7,000	14,000

Energy budget per epoch is 60,000 KEus, a margin of 300 KEus is considered in the energy allocation

7.3.2 Experimental results analysis and discussion

This section aims to explore the use experience optimization with EFQ via experimental results analysis and discussion.

7.3.2.1 Results analysis: optimized use experience with EFQ scheduling

Based on the above energy allocation in each epoch, the user experience of the system is dependent on how each benchmark is scheduled to consume their energy quotas.

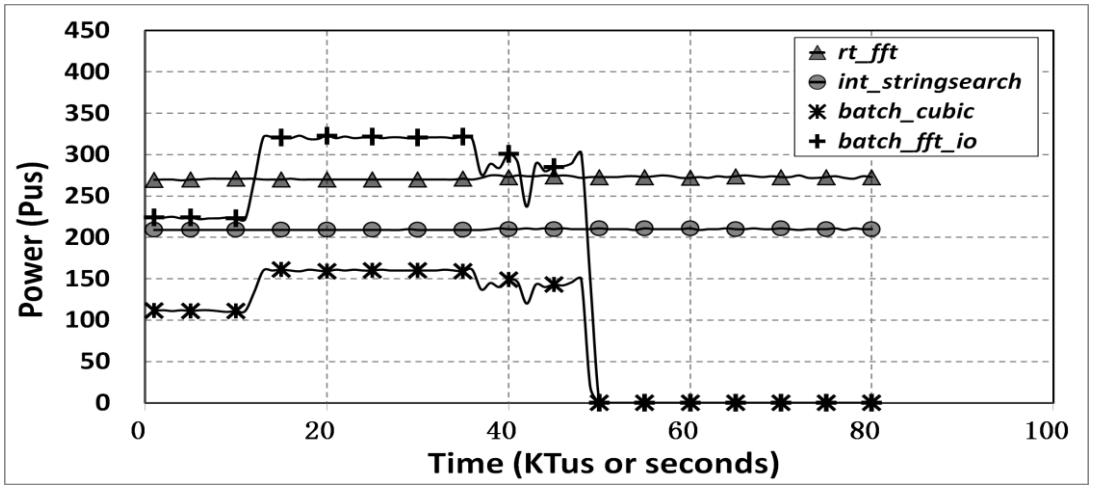
Table 7.7 describes the optimal user experience that can be achieved under EFQ, and Figure 7.5-a) shows the power change of the benchmarks within one epoch under EFQ. Because the user can hardly appreciate the difference of delay that is less than 500 Tus, a higher priority (warp value) is assigned to benchmark *rt_fft* for ensuring stringent deadline compliance. Benchmark *rt_fft* and *int_stringsearch* are assigned an initial share of 0.34 and 0.26, respectively, and the two batch benchmarks are assigned initial weights in a ratio of 1:2.

Table 7. 7: Epoch-based User Experience Optimization under EFQ

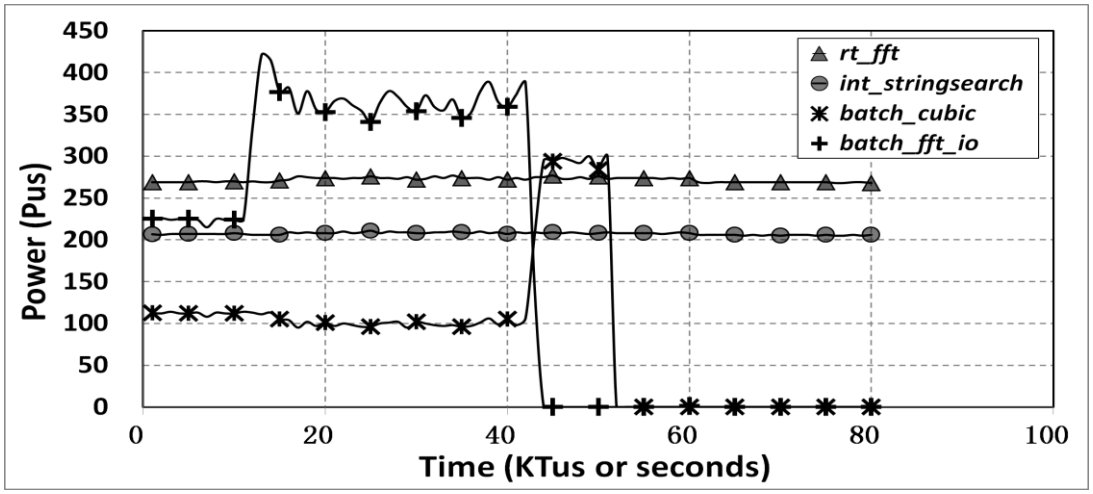
	rt_fft	*int_stringsearch*	*batch_cubic*	*batch_fft_io*
Initial share / weight	0.34	0.26	1	2
Completed periods of work	400	80	N/A	N/A
Number of deadline misses	0	N/A	N/A	N/A
Mean response time (Tus) *	N/A	< 230	N/A	N/A
Maximum response time (Tus) *	N/A	< 340	N/A	N/A
Actual energy consumption (KEus) #	21,805	16,820	7,001#	14,006#
Residual energy per task (KEus)	335	40	0	0
Total residual energy (KEus)	60,000 – 59,632 = 368			

* Because the response time may vary slightly in repeated experiments, only the upper limit value is roughly given

The given values of energy consumption are only based on one experiment, they may vary slightly in several repeated experiments; batch tasks consume slightly more than their energy quotas because the time interval for checking the residual energy cannot be infinitely small.



a) EFQ



b) Linux scheduler

Figure 7. 5: Power Management and Optimization within One Epoch

As can be seen from the experimental results under EFQ, both benchmark *rt_fft* and *int_stringsearch* are executed constantly and normally during the whole epoch. Specifically, benchmark *rt_fft* completes 400 periods of work and strictly meets all the deadlines; benchmark *int_stringsearch* completes 80 periods of work and achieves a response time that is acceptable to the user. Besides, benchmark *batch_cubic* and *batch_fft_io* are dispatched to consume their energy quotas in a power ratio of 1:2. As a result, the two batch benchmarks use up their energy quotas at almost the same time (around 50 KTus). At the end of the epoch, a total of 368 KEus energy is remained in the resource containers of all benchmarks. This is the cost of ensuring a safe energy allocation for user-preferred tasks; however, it is a small amount (0.6%) in comparison with the total energy of one epoch.

In the case that user-preferred tasks are overly allocated energy quotas and a big amount of residual energy is observed at the end of one epoch, the energy allocation in the next epoch should be adjusted to reduce the residual energy (refer to section 2.1.4.1). In any case, once the cumulative energy of a resource container reaches to its capacity limit, the overflowed energy will be redistributed to the resource containers of bench tasks.

7.3.2.2 Extended discussion in comparison with Linux scheduler

Figure 7.5-b) shows the power management within one epoch under the default Linux scheduler. From the graph, we can see the powers of *rt_fft* and *int_stringsearch* are also constantly guaranteed in the whole epoch. This is because, under the Linux scheduler, *rt_fft* belongs to the real-time class so that is treated as an absolutely higher-priority task than others, and *int_stringsearch* is assigned a niceness value that is far greater than those of the two batch benchmarks. For the above reasons, the performance of *rt_fft* and *int_stringsearch* are also guaranteed to meet the user requirements.

However, because of the absolute higher-priority given to real-time class tasks, the default Linux scheduler may suffer several limitations. First, if the real-time task becomes abnormally behaved by over-requesting system resources, the power and

performance of *int_stringsearch* may not be guaranteed even it is assigned a large niceness value. The same situation may also happen if multiple real-time tasks are simultaneously executed in the system. This is not a problem under the EFQ scheduler because of the maximum time that one task can run with priority is restricted. This point will be verified in the next section (section 7.3.2.3) through an experiment assuming abnormal energy requesting behaviors from the real-time benchmark *rt_fft*. Second, in case the performance requirement on *int_stringsearch* has changed and a shorter response time (e.g. 200 Tus) is preferred, then, the Linux scheduler will fail to optimize the user experience, while EFQ can still achieve the optimization goal by exchanging the warp values of benchmark *rt_fft* and *int_stringsearch*. This point is quite obvious so it will not be expanded with experiments.

Another limitation of the Linux scheduler in power optimization is that it has no control of the energy consumptions on non-CPU devices. As shown in Figure 7.5-b), when *batch_fft_io* starts I/O operations on the SD card, the power ratio of the two batch tasks deviates from the initial ratio of 1:2, causing the earlier idle of *batch_fft_io*. This is because the Linux niceness values can only determine the sharing of CPU time quanta among competing tasks. While the activities in other devices are not considered when making the CPU scheduling decisions, the sharing of system power is totally out of the Linux scheduler's control. In comparison, EFQ can achieve proportional sharing of the system power (as shown in Figure 7.5-a)) by accounting system-wide energy consumption in the CPU dispatching. This is beneficial for the system to optimize the energy utilization and provide guaranteed services to different applications. In the experiment of the next section (section 7.3.2.3), we will show how guaranteed power consumption can affect the energy utilization.

7.3.2.3 Results analysis: preserving user experience upon abnormal behaviors

Because the most user-preferred task may not be assigned the highest priority, its power and performance should be protected by the scheduler when any task with a

higher priority is behaving abnormally with excessive energy demands. For simplicity and without loss of generality, the following experiment only considers three benchmarks: *rt_fft, int_stringsearch,* and *batch_cubic*; the remaining energy quota of 21,000 KTus (refer to Table 7.6) is all assigned to benchmark *batch_cubic*.

Figure 7.6 shows the desired power consumptions when all tasks are behaving normally. Figure 7.7 demonstrates the power consumptions under the Linux scheduler and the BSEFQ when benchmark *rt_fft* is ill-behaved and over-requesting a workload of 173 Tus per period (86.5% CPU utilization) during time 20 to 40 KTus. Table 7.8 compares the system performance under the two schedulers.

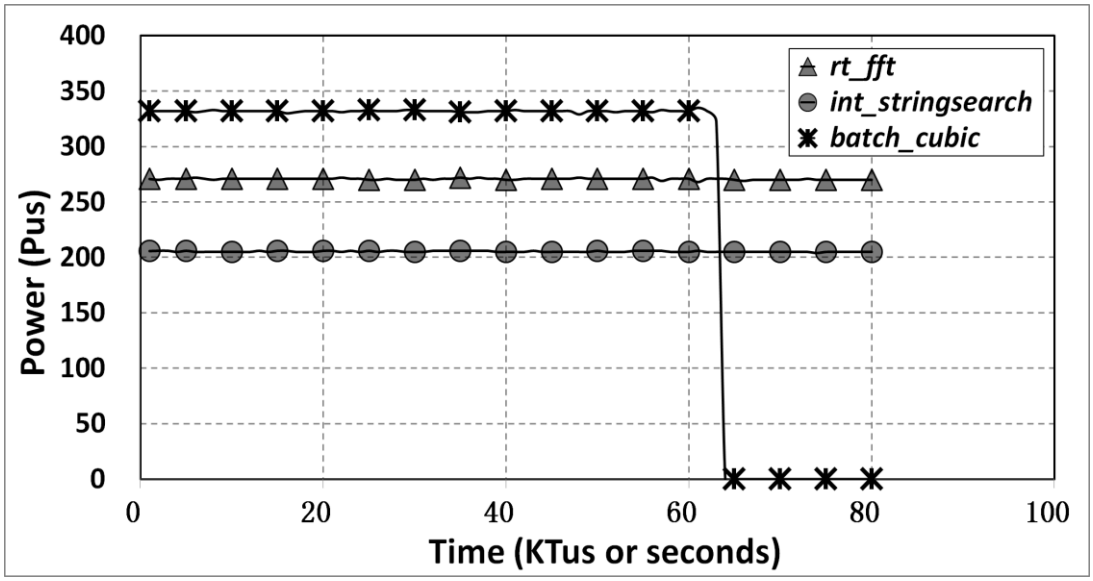
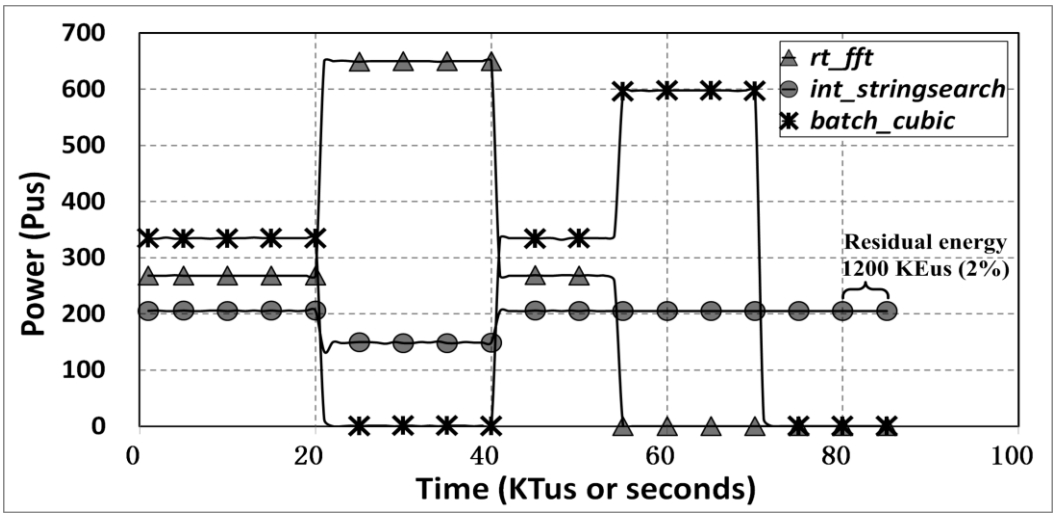


Figure 7. 6: Desired Power Consumptions when all Tasks are Behaving Normally

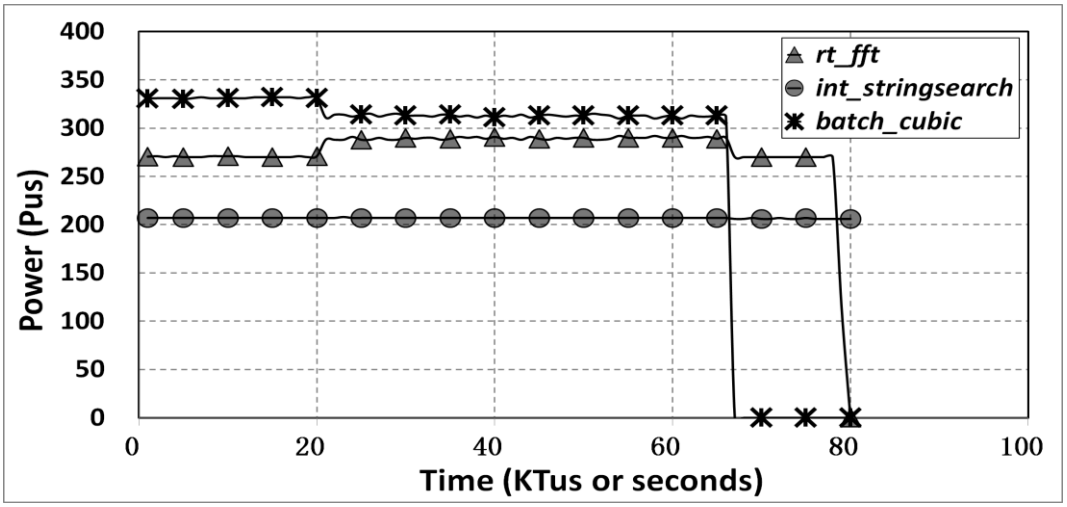
Table 7. 8: Comparison of System Performance under BSEFQ and Linux Scheduler when the Benchmark rt_fft is Abnormally-behaved

	Warp time limit (Tus)	***int_stringsearch* Mean Response time (Tus) ***	***int_stringsearch* Maximum Response time (Tus) ***	***rt_fft* Deadline miss ratio**
BSEFQ	75 to 170	< 260	< 360	>= 25%
	>= 175	> 4400	> 7000	0%
Linux Scheduler	N/A	> 3700	> 6300	0%

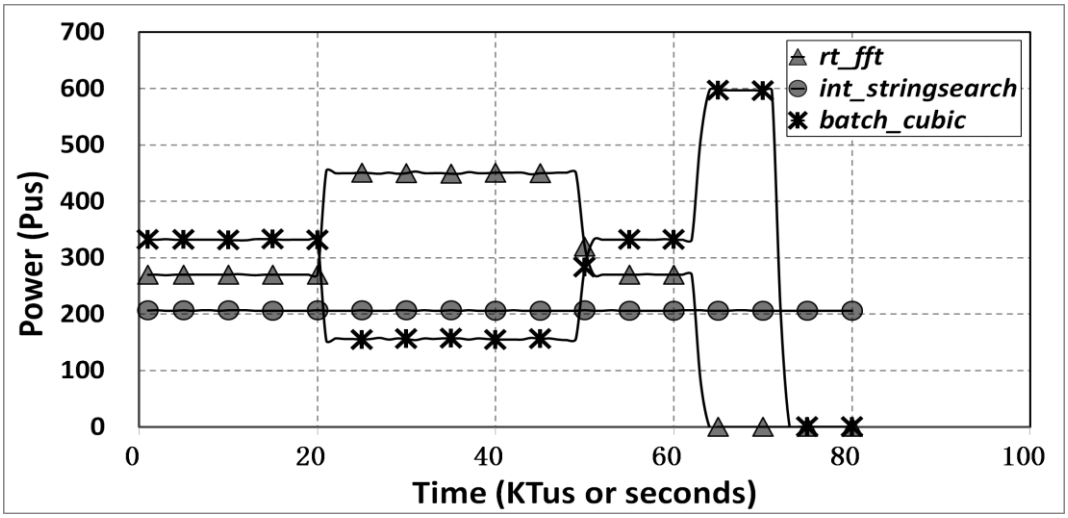
* Because the response time may vary slightly in several repeated experiments, only the upper or lower limit value is roughly given.



a) Linux scheduler



b) BSEFQ, warp time limit = 75 Tus



c) BSEFQ, warp time limit = 130 Tus

Figure 7. 7: Protecting Task Power upon Abnormal Behaviors from Benchmark *rt_fft*

In Figure 7.7-a), the Linux scheduler treats benchmark *rt_fft* as an absolute higher-priority task over others; therefore, it allows benchmark *rt_fft* to excessively consume energy with a power as high as 650 Pus during the ill-behaved period. The high power pulse in *rt_fft* causes the depletion of its energy quota at around 55 KTus. More importantly, it reduces the remaining power share that is available for *int_stringsearch* and *batch_cubic*. Because the niceness value of *int_stringsearch* is set as far greater than that of *batch_cubic*, the power share left by *rt_fft* is totally allocated to *int_stringsearch*, causing the power of *int_stringsearch* and *batch_cubic* to be around 150 Pus and 0 Pus, respectively, during time 20 to 40 KTus.

The reduced power in benchmark *int_stringsearch* causes the degradation of the system performance from two aspects. First, as shown in Table 7.8, although the deadline miss ratio of *rt_fft* is 0% under the Linux scheduler, both the mean response time and the maximum response time of *int_stringsearch* are significantly greater than the user-acceptable level of 500 Tus. It makes no sense to achieve strict time-constraint compliance for a less-preferred yet ill-behaved task at the cost of causing an unacceptable response time to the more-preferred interactive task. Second, because of the reduced power of 150 Pus during the 20 KTus ill-behaved period, a residual energy of more than 1200 KEus (2%) are left by the benchmark *int_stringsearch* at the end of one epoch time 80 KTus. This residual energy appears as a 207 Pus power that lasts until time 85 KTus in Figure 7.7-a). At the same time, benchmark *batch_cubic* is forced to idle before the 80 KTus epoch: the total energy available in one epoch is not fully utilized to maximize the system performance as well as the user experience.

In comparison with the Linux scheduler, BSEFQ performs better in preserving the system performance and user experience upon abnormal behaviors.

In Figure 7.7-b), because a warp time limit of 75 Tus is applied to benchmark *rt_ff*, it is scheduled by BSEFQ as the highest-priority task only within the allowed period of time. Once 75 Tus expires, benchmark *rt_fft* has to compete for the energy with other tasks based on its weight. Therefore, the power of *rt_fft* is controlled under 290 Pus, and the power of *int_stringsearch* is guaranteed to be 207 Pus during the

whole epoch. The available energy within one epoch is fully utilized, which provides an opportunity to optimize the user experience upon abnormal behaviors. Because the power of *rt_fft* is restricted during its ill-behaved period, the unfinished work is postponed to its later periods, causing the power to be continuously maintained at the abnormal level (290 Pus) for a time longer than 20 KTus. The length of time that the abnormal power level continues is dependent on the selected warp time limit.

In Figure 7.7-c), because a larger warp time limit of 130 Tus is applied, the maximum power of *rt_fft* is restricted at 450 Pus, and the length of time that *rt_fft* stays at the abnormal power level is apparently shorter than the one in Figure 7.7-b). The task performance under BSEFQ is also dependent on the warp time limit. According to the results in Table 7.8, when the warp time limit is larger than 175 Tus, benchmark *rt_fft* is allowed to take an overly high power pulse to meet all of its deadlines while benchmark *int_stringsearch* is suffering a terribly long response time. However, as long as the warp time limit is properly set (between 75 and 170 Tus), the response time of benchmark *int_stringsearch* is guaranteed to be no greater than the user-acceptable delay (500 Tus), and the deadline miss ratio of benchmark *rt_fft* is at least 25% because the deadlines during the 20 KTus abnormal period are all missed.

Note that the deadline miss ratio under BSEFQ is dependent on not only the warp time limit but also the specific real-time application. If a real-time application can abandon its unfinished service quanta and skip to the next period, the deadline miss ratio is constantly 25% no matter how the warp time limit changes (within the 75 to 170 Tus range). Otherwise, as the warp time limit increases, more time quanta of benchmark *rt_fft* is served ahead of those of benchmark *int_stringsearch*, therefore, the deadline miss ratio of *rt_fft* is reduced while the response time of *int_stringsearch* is increased. In a real system where time-sensitive tasks can have a variable workload per period, it is more practical to assign a compromised value to the warp time limit, so that, on one hand, the time-constraint compliance under normal energy load can be guaranteed, and, on the other hand, high power pulses caused by abnormal energy request can be restricted under a proper level.

7.4 Summary

In this chapter, the EFQ algorithm is evaluated through specifically-designed experiments that are performed in the Linux experimental platform with the Pthread-based multithreading test-bench program. The default Linux scheduler is employed as a reference to highlight the EFQ scheduling properties. The EFQ algorithm is evaluated from the ability of managing the power share and guaranteeing the time-constraint compliance to the potential of optimizing the mobile system user experience under the energy limit. Next, the experimental results will be summarized from these three aspects.

First, EFQ provides a straightforward way to manage the power share of each task by scheduling tasks based on their system-wide energy consumption. Experimental results show that the system power can be proportionally shared among the tasks in a user-desired ratio under the EFQ scheduling, regardless of how the task power changes and in which device the energy is spent. This is impossible to achieve under the default Linux scheduler. In addition, it is also shown through experiments that EFQ can protect the power share of specific tasks upon the change of the scheduling environment. With this property, the power share of user-preferred or time-sensitive tasks can be guaranteed so that they can finish a target amount of work with a stable performance when the total energy budget is limited.

Second, BSEFQ can support effective and flexible time-constraint compliance in GPOSs. Through a series of comparative experiments on SEFQ and BSEFQ, it is found that, the time-constraint compliance under SEFQ is susceptible by the task number and the energy estimation error. Therefore, in order to ensure the time-constrain compliance, the power share of time-sensitive tasks should be overly and conservatively allocated, which is applicable only if the CPU workload is moderate. In contrast, with a power share that is no less than the maximum long-term power share, BSEFQ can provide robust and effective time-constraint compliance upon the increase of task number and energy allocation error. And since BSEFQ can adjust the warp values of different time-sensitive tasks, it allows a flexible tradeoff of

their performance.

Finally, in comparison with the default Linux scheduler, EFQ is more flexible and effective in optimizing and preserving the mobile system user experience under the energy limit. Experimental results demonstrate that, by sharing the system power proportionally to the ratio of the energy allocation, EFQ can guarantee that each task has the opportunity to use up its energy quota in a stable rate. This is pivotal in ensuring a stable performance for each task and maximizing the battery energy utilization. Besides, it is also demonstrated that, upon the appearance of high energy load or abnormal behaviour with excessive energy demands from a high-priority task, EFQ can trade off the power share and performance of the task by adjusting the maximum time that the task can run with high priority. Thus, the power share of other user-preferred tasks can be protected and the system user experience can be optimally preserved. In the case of the default Linux scheduler, since the power share of each task is not absolutely protected from competition, certain task may suffer power share reduction and fail to use up its energy quota. This leads to the loss of opportunity to maximize the energy utilization and optimize the user experience.

Chapter 8

Conclusions

This chapter is divided into three sections. The first section summarizes the work of this dissertation and discusses its contributions; the second section analyzes the limitations of the work and suggests the directions for future research; and the final section highlights again the general contribution of this work in reference to the research community of power management.

8.1 Summary and discussion

The user experience of modern GPOS-based mobile devices is increasingly limited by the battery capacity, how to optimize the mobile system user experience under the battery energy limit is a challenging problem that needs to be tackled by system designers and researchers. Considering that, on the one hand, a user-desired battery lifetime is one of the most important, though unstable, contributing factors of the mobile system user experience, and on the other hand, energy-centric power management (PM) schemes can provide strong guarantees regarding the battery lifetime, this dissertation work has investigated on the optimization of the mobile system user experience from the perspective of energy-centric processor scheduling in an energy-centric context.

Energy-centric processor scheduling is the core module of an energy-centric PM scheme in managing the energy flows to the applications and guaranteeing the application performances. So the first question faced in this dissertation work is: what are the requirements of energy-centric processor scheduling with regard to the mobile system user experience optimization under the energy limit. To answer this question, we have focused on the general contributing factors of the mobile system user experience that are independent of specific user preferences, mapped them to the processor scheduling, and determined three essential requirements on the

energy-centric processor scheduling, which are proportional power sharing, time-constraint compliance, and when necessary, a tradeoff between the power share and the time-constraint compliance.

With the general requirements determined, the next step is to design the energy-centric processor scheduling algorithm. We have firstly investigated the traditional GPOS scheduling algorithms and comparatively discussed the possibilities of applying these algorithms on energy-centric processor scheduling, and then, proposed the development of energy-based fair queuing (EFQ) for energy-centric processor scheduling by extending traditional fair queuing algorithms and the Generalized Processor Sharing (GPS) model into the energy management domain. As the basis of energy-based fair queuing, a practical energy-centric scheduling model has been built up by considering the energy as a proportionally shared resource in the GPS model, the management of proportional power shares has been analyzed and effective mechanisms of power share protection and reallocation have been proposed. Based on the energy model, the challenges of EFQ algorithm development have been analyzed; it has been found out that performing EFQ scheduling is restricted by the unpredictability of the energy packet size, the volatility of the system power and the practical issues such as the implementation complexity and the scheduling overhead. With these challenges in mind, the starting-energy fair queueing (SEFQ) has been proposed. It is simple to implement and can achieve near-optimal fairness bound for proportional power sharing under variable system power with low computing-time complexity.

After the proposal of SEFQ, the time-constraint compliance under energy-based fair queuing scheduling has also been analyzed. It has been found out that achieving time-constraint compliance under EFQ scheduling requires an overly-reserved power share for each time-sensitive task. To further improve the support of real-time and multimedia scheduling in EFQ, the borrowed starting-energy fair queuing (BSEFQ) has been proposed by combining a real-time friendly mechanism named warping into the SEFQ. The warping mechanism in BSEFQ can improve the scheduling latency of time-sensitive tasks by giving time-limited priority to them; this breaks the short-term

fairness in energy sharing but maintains the proportional power sharing in the long-term. Besides, by adjusting the maximum time each time-sensitive task can run in warping, the power share and time-constraint compliance of time-sensitive tasks can be flexibly traded off. Based on the theoretical proposal of SEFQ and BSEFQ, a high-level modelling and simulation of EFQ scheduling has been implemented in SystemC; the simulation results have shown that the proposed EFQ algorithm can meet the general requirements of energy-centric processor scheduling in the achievement of proportional power sharing, time-constraint compliance, and a trade-off between the power share and time-constraint compliance upon the appearance of highly variable or abnormally behaving energy requests.

The proposed EFQ algorithm has been implemented in the Linux kernel, a Linux scheduling simulation test-bench has been developed to learn the Linux scheduling subsystem and debug the Linux-based EFQ implementation. It has been found out that the completely fair scheduler (CFS) of Linux is a variant of the fair queuing in processor scheduling and it shares the same basic principles with the EFQ algorithm. Therefore, the organizational structure of the Linux-CFS scheduler has been maximally utilized to implement the EFQ algorithm, and it has been turned out that the EFQ algorithm is simple and low-overhead to implement in Linux. The Linux-based EFQ scheduler has been evaluated in an experimental test-bench, in which a multithreading test-bench program is utilized to create EFQ threads with the power information self-contained. The experimental results have demonstrated the effectiveness of the EFQ algorithm in managing system-wide power shares and guaranteeing time-constraint compliance. Based on the verified properties of the EFQ algorithm, the potential benefits of employing EFQ scheduling in optimizing and preserving the mobile system user experience have been explored through specifically-designed experiments. The experimental results have demonstrated that energy-based fair queuing is more effective than traditional processor scheduling algorithms, such as those of the default Linux scheduler, in user experience optimization for energy-limited mobile systems.

8.2 Limitations and future work

This work explores the optimization of mobile system user experience with energy-centric processor scheduling; it experiences some limitations due to the assumptions and conditions that are made on the applications and the processor scheduling surroundings. To further improve and extend the work, a variety of directions can be followed for future research.

The main limitation of this thesis work is the lack of the support of a complete energy-centric system. Because the energy allocation module has not been implemented, the energy management and energy utilization optimization over a user-specified battery time cannot be demonstrated; and because the energy accounting module has not been implemented, a real-time and on-line feedback of the energy consumption to each application is impossible and the energy consumption on various hardware devices cannot be fully explored to guide the EFQ scheduling. Despite of the above limitation, the main properties of EFQ scheduling have been successfully evaluated by first profiling the power consumption on CPU and SD card I/O and then self-containing the power information of each thread in our experimental test-bench. To further explore the dynamic computing and adjusting of the task weight and power share and better demonstrate the potential benefits of employing EFQ in optimizing the user experience of energy-limited mobile systems, the energy allocation and energy accounting modules should be implemented in the future to form a complete energy-centric system.

Another limitation of the work is the simplicity of the mobile applications. In this thesis work, each application is modelled as a single-threaded process that is independent to other applications, and the applications are programed based on a set of simple benchmarks with specifically-designed workloads. This is practical and effective when designing a very basic processor scheduling algorithm and assessing its properties. To further evaluate the scheduling behaviors of the EFQ algorithm in a realistic mobile system, multithreading applications with real workloads and inter-process communication should be employed in the experimental test-bench.

Correspondingly, the threads of different applications should be organized in groups and the EFQ algorithm should be enhanced to support proportional share group scheduling; also, more challenges on energy accounting and application weight adjustment are expected to arise.

Besides of the future work on energy-centric system and test-bench applications, this work can be further extended from the following two aspects. First, as many modern mobile systems are commonly featured with multi-core processors, the energy-based fair queuing algorithm can be extended to support load balancing and multi-core proportional power share scheduling. Second, classical energy-efficient policies, such as DVFS, DPM, and application-self-adaptation, can be combined into the energy-centric power management framework to further save energy and optimize the user experience of energy-limited mobile systems.

8.3 Final words

This work considers the guarantee of a user-desired battery lifetime as the fundamental requirement of a mobile system user under the energy limit, and explores the user experience optimization of energy-limited mobile systems from the perspective of energy-centric processor scheduling in an energy-centric context, where a strong guarantee of the battery lifetime is possible. The energy-based fair queuing (EFQ) algorithm has been proposed and implemented in Linux for performing the energy-centric processor scheduling, and an experiment-based assessment has demonstrated the effectiveness of EFQ in power management, performance guarantee and, user experience optimization for energy-limited mobile systems. In that context, this is the first presentation of a formally-structured and concretely-implemented energy-centric processor scheduling algorithm. In addition, this is the first work to explore the employment of energy-centric processor scheduling in the optimization of mobile system user experience under the energy limit.

Bibliography

[1] U. Reiter, “Perceived quality in consumer electronics – from quality of service to quality of experience,” 13th IEEE International Symposium on Consumer Electronics (ISCE 2009), Kyoto, Japan, May. 2009.

[2] S. Kim, H. Kim, J. Hwang, J. Lee, and E. Seo, “An event-driven power management scheme for mobile consumer electronics,” IEEE Trans. Consumer Electron., vol. 59, no.1, pp. 259-266, Feb. 2013.

[3] S. Lim, S. W. Lee, B. Lee, and S. Lee, “Power-aware optimal checkpoint intervals for mobile consumer devices,” IEEE Trans. Consumer Electron., vol. 57, no. 4, pp. 1637-1645, Nov. 2011.

[4] H. Zeng, C. S. Ellis, and A. R. Lebeck, “Experiences in managing energy with ECOSystem,” IEEE Pervasive Computing, vol. 4, no. 1, pp. 62–68, 2005.

[5] R. Neugebauer and D. McAuley, “Energy is just another resource: energy accounting and energy pricing in the Nemesis OS,” In Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII), May. 2001.

[6] B. Li, and S. Park, “Energy efficient burst scheduling in mobile TV services,” IEEE Trans. Consumer Electron., vol. 59, no.1, pp. 24-30, Feb. 2013.

[7] J. Lorch and A. Smith. “Software Strategies for Portable Computer Energy Management,” IEEE Personal Commun., vol. 5, pp. 60–73, June 1998.

[8] L. Benini and G. De Micheli. “Dynamic Power Management: Design Techniques and CAD Tools,” Norwell, MA: Kluwer, 1998.

[9] L. Benini, A. Bogliolo, S. Cavallucci, and B. Riccó. “Monitoring System Activity for OS-directed Dynamic Power Management,” In Proceedings of the International Symposium on Low Power Electronics and Design, pp. 185–190, Aug. 1998.

[10] T. Pering, T. D. Burd, and R. W. Brodersen. “The Simulation and Evaluation of Dynamic Scaling Algorithms,” In Proceedings of the International Symposium on Low Power Electronics and Design, August 1998.

[11] W. Yuan and K. Nahrstedt. “Energy-efficient Soft Real-time CPU Scheduling for Mobile Multimedia Systems,” In Proc. 19th ACM Symp. Operating Systems Principles (SOSP 03), ACM Press, pp. 149–163, 2003.

[12] J. Lorch and A. Smith. “Improving Dynamic Voltage Scaling Algorithms with PACE,” In Proc. of ACM SIGMETRICS 2001 Conference, June 2001.

[13] P. Pillai and K. G. Shin. “Real-time Dynamic Voltage Scaling for Low-power Embedded Operating Systems,” In Proc. of 18th Symposium on Operating Systems Principles, Oct. 2001.

[14] J. Flinn and M. Satyanarayanan. “Energy-aware Adaptation for Mobile Applications,” In Symposium on Operating Systems Principles (SOSP), pages 48–63, December 1999.

[15] J. Flinn, M. Satyanarayanan, “Managing battery lifetime with energy-aware adaptation,” ACM Transactions on Computer Systems (TOCS), v.22 n.2, p.137-179, May 2004.

[16] C. S. Ellis. “The Case for Higher-Level Power Management,” In Proceedings of the 7th Workshop on Hot Topics in Operating Systems, Rio Rico, AZ, March 1999.

[17] A. Vahdat, C. Ellis, and A. Lebeck. “Every Joule is Precious: The Case for Revisiting Operating System Design for Energy Efficiency,” In Proceedings of the 9th ACM SIGOPS European Workshop, September 2000.

[18] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. “Ecosystem: Managing Energy as a First Class Operating System Resource,” Proc. 10th Int’l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOSX), ACM Press, pp. 123–132, 2002.

[19] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich, “Energy management in mobile devices with the cinder operating system,” Proceedings of the sixth conference on Computer systems, April 10-13, 2011, Salzburg, Austria.

[20] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. “Currentcy: A Unifying Abstraction for Expressing Energy,” Proc. USENIX Ann. Technical Conf., USENIX, pp. 43–56, 2003.

[21] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile application-aware adaptation for mobility. In Proceedings of the 16th ACM Symposium on Operating Systems and Principles, pages 276–287, Saint-Malo, France, October 1997.

[22] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. “The Design and Implementation of an Operating System to Support Distributed Multimedia Applications,” IEEE Journal on Selected Areas In Communications, 14(7): 1280-1297, Sept. 1996.

[23] G. Banga, P. Druschel, and J. C. Mogul. “Resource Containers: A New Facility for Resource Management in Server Systems,” In Third Symposium on Operating Systems Design and Implementation, February 1999.

[24] J. Flinn and M. Satyanarayanan. “PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications,” Second Workshop on Mobile Computing Systems & Applications (WMCSA’99), New Orleans, February, 1999.

[25] F. Bellosa, “The benefits of event-driven energy accounting in power-sensitive systems”, In proceedings of the 9th ACM SIGOPS European Workshop, pp.37-42, Sept. 2000.

[26] B. Goel, S. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati, “Portable, scalable, per-core power estimation for intelligent resource management”, In Proceedings of the 2010 International conference on Green Computing, pp135 –146, Ago. 2010.

[27] Benjamin C. Lee , David M. Brooks, “Accurate and efficient regression modeling for microarchitectural performance and power prediction,” ACM SIGOPS Operating Systems Review, v.40 n.5, December 2006

[28] K. Singh , M. Bhadauria , S. A. McKee, “Real time power estimation and thread scheduling via performance counters,” ACM SIGARCH Computer Architecture News, v.37 n.2, May 2009

[29] C. Lively, X. F. Wu, V. Taylor, S. Moore, H. Chang, and C. Su et al, “Power-Aware Predictive Models of Hybird (MPI/OpenMP) Scientific Applications on Multicore Systems”, International Conference on Energy-Aware High Performance Computing, Sept. 2011.

[30] David C. Snowdon, Stefan M. Petters, and Gernot Heiser. “Accurate on-line prediction of processor and memory energy usage under voltage scaling,” In 7th Int. Conf. Emb. Softw., Salzburg, Austria, Oct 2007.

[31] X. Yu, R. Bhaumik, Z.Y. Yang, M. Siekkinen, P. Savolainen, and A. Ylä-Jääski, “A System-level Model for Runtime Power Estimation on Model Devices”, IEEE/ACM Int’l Conference on & Int’l Conference on Cyber, Physical and Social Computing, pp.27-34, Dec.2010.

[32] T. Cignetti, K. Komarov, and C. Ellis, "Energy estimation tools for the Palm," in Proc. of the ACM Modeling, Analysis and Simulation of Wireless and Mobile Systems, 2000, pp. 96-103.

[33] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler, "Energy metering for free: augmenting switching regulators for realtime monitoring," in Proc. Int. Conf. Information Processing in Sensor Networks (IPSN) St. Louis , MO , USA, April 2008.

[34] Mian Dong and Lin Zhong, "Self-constructive, high-rate energy modeling for battery-powered mobile systems," in Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys), June 2011.

[35] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in Proc. Int. Conf. Hardware-Software Codesign and System Synthesis (CODES+ISSS), Scottsdale, AZ, USA, October 2010.

[36] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y. M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in Proc. European Conf. Computer Systems (EuroSys), Salzburg, Austria, April 2011.

[37] R. Fonseca, P. Dutta, P. Levis, I. Stoica, C. A. Berkeley, and C. A. Stanford, "Quanto: tracking energy in networked embedded systems," in Proc. USENIX Symp. Operating System Design and Implementation (OSDI) San Diego, CA, USA, December 2008, pp. 323–338.

[38] A. Silberschatz, P. B. Galvin, G. Gagne, Operating System Concepts, 7th edition, John Wiley & Sons, 2014. ISBN 0-471-69466-5

[39] W. Mauerer, Professional Linux Kernel Architecture, Wrox, 2008. ISBN 0-470-34343-5

[40] J. Nieh and M. S. Lam. “A SMART Scheduler for Multimedia Applications,” ACM Transactions on Computer Systems, 21(2), pp. 117-163, May 2003.

[41] Mach Scheduling and Thread Interfaces. https://developer.apple.com/library/mac/documentation/darwin/conceptual/kernelprogramming/scheduler/scheduler.html

[42] Dario Faggioli, Fabio Checconi, Michael Trimarchi, Claudio Scordino, An EDF scheduling class for the Linux kernel, 11th Real-Time Linux Workshop (RTLWS), Dresden, Germany, September 2009.

[43] C.W. Mercer, S. Savage, and H.Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In Proceedings of the IEEE International Conference on Multimedia Computing and Systems, Boston, MA, PP. 90-99, 1994.

[44] Giorgio C. Buttazzo, Rate monotonic vs. EDF: judgment day, Real-Time Systems, v.29 n.1, p.5-26, January 2005

[45] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM (JACM), Volume 20 Issue 1, Pages 46 - 61, Jan. 1973.

[46] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. IEEE Journal of Selected Areas in Communications (JSAC) 14, 7 (Sept.), pp. 1280-1297, 1996.

[47] M. Dertouzos. Control Robotics: The Procedural Control of Physical Processors. In Proceedings of the IFIP Congress. Stockholm, Sweden, pp. 807-813, 1974.

[48] M. B. Jones, D. Rosu, and M. C. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles. St. Malo, France, pp. 198-211, 1997.

[49] Giorgio Buttazzo , Giuseppe Lipari , Luca Abeni , Marco Caccamo, Soft Real-Time Systems: Predictability vs. Efficiency (Series in Computer Science), Plenum Publishing Co., 2005

[50] J. Nieh and M.S. Lam. The Design, Implementation and Evaluation of SMART: A scheduler for Multimedia Applications. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles. ACM, St. Malo, France, pp. 184-197, 1997.

[51] L. Abeni , G. Buttazzo, Integrating Multimedia Applications in Hard Real-Time Systems, Proceedings of the IEEE Real-Time Systems Symposium, p.4, December 02-04, 1998

[52] L. Abeni, G. Lipari, and G. Buttazzo. Constant bandwidth vs. proportional share resource allocation. In Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems (ICMCS '99), June 1999.

[53] Lui Sha , Tarek Abdelzaher , Karl-Erik Årzén , Anton Cervin , Theodore Baker , Alan Burns , Giorgio Buttazzo , Marco Caccamo , John Lehoczky , Aloysius K. Mok, Real Time Scheduling Theory: A Historical Perspective, Real-Time Systems, v.28 n.2-3, p.101-155, November-December 2004

[54] I. Stoica, H. Abdel-wahab, and K. Jeay. On the Duality between Resource Reservation and Proportional Share Resource Allocation, In Proc. of Multimedia Computing and Networking, pp. 207-214, 1997.

[55] I. Stoica, H. Abdel-Wahab, K. Jeffay, and S. K. Baruah. A Proportional Share Resource Allocation Algorithm for Real-time, Time-shared Systems. In Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), pages 288-299, Dec. 1996.

[56] J. Regehr. Some Guidelines for Proportional Share CPU Scheduling in General-purpose Operating Systems, In The 22nd IEEE Real-Time Systems Symposium (RTSS 2001), London, UK, December 3-6 2001.

[57] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," IEEE/ACM Transactions on Networking, Vol. 1, Issue. 3, pp. 344-357, Jun. 1993.

[58] I. Stoica, H. Abdel-Wahab. Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation. Technical Report TR-95-22, CS Dpt., Old Dominion Univ., Nov. 1995.

[59] S.J. Golestani. A Self-Clocked Fair Queueing Scheme for High Speed Applications. In Proceedings of INFOCOM'94, 1994.

[60] D. Stiliadis and A. Varma, Latency-rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms, In Proc. IEEE INFOCOM '96, San Francisco, CA, pp. 111–119, Apr. 1996.

[61] Stiliadis D., and A. Varma. Rate-Proportional Servers: A Design Methodology for Fair Queueing Algorithms. IEEE/ACM Transactions on Networking, Vol. 6, No 2, pp. 164-174. April 1998.

[62] J. S. Goddard and J. Tang. EEVDF Proportional Share Resource Allocation Revisited, in Work-in-Progress Sessions of the 21st IEEE Real-Time Systems Symposium (RTSSWIP00), Nov. 2000.

[63] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. Proceedings of the First Symposium on Operating Systems Design and Implementation, November 1994.

[64] P. Goyal, X. Guo, and H. M. Vin. 1996. A hierarchical CPU scheduler for multimedia operating systems. Proc. Second USENIX Symposium on Operating System Design and Implementation (OSDI), pp 107-122, 1996.

[65] R. E. Al-Ouran, Linux Implementation of a New Model for Handling Task Dynamics in Proportional Share Based Scheduling Systems, Ohio University, 2010. http://rave.ohiolink.edu/etdc/view?acc_num=ohiou1275681466

[66] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, "Weighted Round Robin Cell Multiplexing in a General Purpose ATM Switch", IEEE JSAC, pp. 1265-1279, October 1991.

[67] M. Shreedhar, G. Varghese, "Efficient Fair Queuing Using Deficit Round Robin," IEEE/ACM Trans. Networking, 1996, Vol.4 No. 3, pp. 375-85.

[68] P. Goyal, H. M. Vin, and H. Cheng. Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. IEEE/ACM Transaction on Networking, Vol. 5, No. 5, October 1997.

[69] C. A. Waldspurger and W. E. Weihl. Stride Scheduling: Deterministic Proportional Share Resource Management. Technical Memorandum, MIT/LCS/TM-528, Laboratory for CS, MIT, July 1995.

[70] L. Zhang. VirtualClock: A New Traffic Control Algorithm for Packet Switching Networks, Proceeding of the ACM SIGCOMM'90. pp. 19-29. September 1990.

[71] J.C.R. Bennett and H. Zhang. WF2Q: Worst-case fair weighted fair queueing. In Proceedings of IEEE INFOCOM'96, pages 120-128, San Francisco, CA, March, 1996

[72] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. Proc. Sigcomm '89, 19(4):1-12, September 1989.

[73] K. Lee. Performance Bounds in Communication Net-works With Variable-Rate Links. In Proceedings of ACM SIGCOMM'95, pages 126-136, 1995.

[74] K. J. Duda and D. R. Cheriton. 1999. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general purpose scheduler. In Proceedings of the 17th ACM Symposium on Operating System Principles, Dec. 1999.

[75] A. Bavier, L. Peterson, and D. Mosberger. BERT: A Scheduler for Best-Effort and Realtime Paths. Technical Report TR-602-99, Department of Computer Science, Princeton University, 1999.

[76] IEEE 1666-2011 standard for standard SystemC language reference manual, IEEE, New York, USA, January, 2012.

[77] D. C. Black, J. Donovan, SystemC: From the Ground Up, 2nd ed., Springer 2009. ISBN 0-387-69957-0

[78] J. Wei, E. Juarez, F. Pescador and M. J. Garrido, "Starting-energy fair queuing (SEFQ): a novel class of energy-aware scheduling algorithms for mobile systems," 16th IEEE International Symposium on Consumer Electronics (ISCE 2012). Harrisburg, USA, 4-6 June. 2012.

[79] J. M. Calandrino, D. P. Baumberger, T. Li, J. C. Young, and S. Hahn, "LinSched: the Linux scheduler simulator," In Proceedings of the 21st international conference on parallel and distributed computing and communications systems, 2008. p. 171–6.

[80] LinSched: The Linux Scheduler Simulator. http://www.cs.unc.edu/~jmc/linsched/

[81] LinSched for 2.6.35 released. http://lwn.net/Articles/409680/

[82] LinSched for v3.3-rc7. http://lwn.net/Articles/486635/

[83] BeagleBoard System Reference Manual Rev. C4, December 2009.

[84] The Ångström Distribution. http://www.angstrom-distribution.org/

[85] J. Herrera, Desarrollo de un Emulador de Baterías para el Estudio del Consumo de la Tarjeta BeagleBoard, PFC EUITT-UPM, Jul 2011.

[86] USER'S GUIDE, Agilent Technologies, Model 66319B/D, 66321B/D, Mobile Communications DC Source. http://cp.literature.agilent.com/litweb/pdf/5964-8184.pdf

[87] BeagleJuice WiKi. http://antipastohw.pbworks.com/w/page/31143822/BeagleJuice

[88] M. R. Guthaus , J. S. Ringenberg , D. Ernst , T. M. Austin , T. Mudge , R. B. Brown, "MiBench: a free, commercially representative embedded benchmark suite," Proceedings of 4th Annual IEEE Workshop on Workload Characterization, pp. 3-14, December 02-02, 2001.