View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

© 2004 by Wanghong Yuan. All rights reserved

GRACE-OS: AN ENERGY-EFFICIENT MOBILE MULTIMEDIA OPERATING SYSTEM

BY

WANGHONG YUAN

B.S., Beijing University, 1996
M.S., Beijing University, 1999

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

Abstract

Multimedia-enabled mobile devices, such as camera phones, need to support multimedia semantics with high quality of service (QoS) requirements under limited system resources such as CPU time and battery energy. On the other hand, these mobile devices also provide new opportunity for QoS provisioning and energy saving due to the *adaptive* hardware and software components. Researchers have therefore introduced *adaptation* into various system layers, ranging from hardware to applications. Previous adaptation work often adapts only some layers or only at coarse time granularity such as application entry or exit. We believe that to fully reap the benefits of adaptation, it is necessary to take a *cross-layer adaptation* approach, in which all system layers are adaptive and cooperate with each other in response to system changes at different time granularity.

This thesis presents a novel operating system, called *GRACE-OS*, to support such cross-layer adaptation in the operating system by coordinating the adaptation in different layers and enforcing the coordinated decisions via energy-aware real-time scheduling. This thesis makes four major contributions. First, we propose a hierarchical adaptation framework to coordinate adaptation in different layers. This framework consists of *global* and *internal* adaptation. The former coordinates all three layers in response to large system changes, while the latter adapts each individual layer in response to small system changes. This two-level adaptation hierarchy achieves the benefits of the cross-layer adaptation with acceptable overhead. Second, we extend traditional real-time scheduling with another dimension, *speed*, for mobile devices with a variable-speed processor. That is, the scheduler decides how fast to execute applications in addition to when to execute what applications. This extended scheduling algorithm enables applications to operate at the coordinated

quality level with minimum energy. Third, we develop a set of algorithms for internal adaptation in the operating system and CPU hardware. These algorithms adjust the CPU speed to handle small variations in application CPU demand. Their goal is to minimize the total power consumed by the device while preserving the soft deadline guarantees. Finally, we develop a kernel-based profiling technique to monitor the CPU usage of individual applications and predict their CPU demand for both global and internal adaptation.

We have implemented GRACE-OS in the Linux kernel and evaluated it with adaptive Athlon processor and adaptive video codec applications. Our experimental results show that GRACE-OS efficiently trades off QoS for energy with acceptable overhead. In particular, compared to previous systems that adapt only some system layers, GRACE-OS achieves the user-desired battery lifetime and saves energy by up to 59% while providing better or the same multimedia quality. Compared to previous systems that adapt only at coarse time granularity, GRACE-OS saves energy by 2% to 8.9% without affecting multimedia quality.

To my parents and Wei

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Klara Nahrstedt, for her invaluable directions and continuous support throughout my PhD research. Her insights and guidance to my thesis topic enlightened me in various detailed aspects throughout the work. Her advice in the development of my writing and presentation skills has been especially helpful for my future career. Her help and encouragement have smoothed the process of my PhD research. I feel extremely lucky to have Klara as my advisor.

I would like to thank Professor Sarita Adve. As a committee member and the leader of the GRACE project, she has given me a lot of help on the related work and ideas of my thesis. She has also encouraged me to communicate my ideas with others. I would like to thank other members of my thesis committee, Professors Roy Campbell and Lui Sha, for their feedback on the ideas in my research work and their advice on improving this thesis. In addition, I am grateful to Professors Jiawei Han and Kevin Chang for their kind help and advice during my research.

Thanks also go to the members in the GRACE project. Specifically, I would like to thank Professors Robin Kravets and Douglas Jones for their helpful discussion and feedback. I am also grateful to Albert Harris, Christopher Hughes, Daniel Grobe Sachs, and Vibhore Vardhan for their collaboration during the development of the GRACE system.

I am grateful to my colleagues in the MONET group. We have been working together for paper submissions, system demos, and technical presentations. Special thanks go to Baochun Li and Dongyan Xu for their advice during the PhD study, to Jun Wang, Li Xiao, Xiaohui Gu, and Yi Cui for their informative discussion, and to Bin Yu for his technical help. I would also like to thank Yuan Xue, Kai Chen, Samarth Shah, Jingwen Jin, Won J. Jeon, Kihun Kim, Duangdao

Wichadakul, Chui Sian Ong, and Jin Liang. I am grateful to Anda Ohlsson, Sheila Clark, Erna Amerman, and Molly Flesner for their great administrative support during my graduate study.

Last but not least, my family deserves particular recognition for their greatest love and support. Specially, I am grateful to my parents for their encouragement and to my wife, Wei, for her love and belief in my graduate studies, without which all that I have achieved was not possible.

The work presented in this thesis was supported by the DARPA grant under contract number F30602-97-2-0121, the National Science Foundation under contract number CCR 96-23867, CISE CDA 96-24396, CISE EIA 99-72884, and CCR 02-05638, and the NASA grant under NAG 2-1250. However, views and conclusions of this thesis are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, NSF, NASA, or the U.S. government.

Table of Contents

Chapter 1 Introduction 1
1.1 Motivation 1
1.1.1 What Is Available 2
1.1.2 What Is Missing 3
1.2 GRACE-OS: An Energy-Efficient Multimedia OS 4
1.2.1 Research Problems 5
1.2.2 Solution Overview 5
1.2.3 Major Contributions 7
1.3 Thesis Organization 9

Chapter 2 System Models 10
2.1 CPU Frequency and Voltage Adaptation 10
2.2 Application Quality Adaptation 12
2.3 Operating System Allocation Adaptation 13
2.4 Cross-Layer Configuration 15
2.5 Adaptation Triggers 15

Chapter 3 GRACE-OS: An Overview 17
3.1 Background— The GRACE System 17
3.1.1 GRACE Architecture 19
3.1.2 Adaptation Hierarchy 21
3.2 Overview of GRACE-OS 25
3.2.1 Operating System Roles in GRACE 25
3.2.2 The GRACE-OS Architecture 27
3.3 Summary 28

Chapter 4 Global Adaptation 29
4.1 Global Adaptation Problem 30
4.2 Solution 33
4.2.1 NP Hardness 33
4.2.2 Heuristic Algorithm 35
4.3 Global Adaptation Protocol 36
4.4 Long-Term Demand Prediction 38
4.4.1 Kernel-Based Profiling of Cycle Usage 39

4.4.2 Estimation of Demand Distribution . . . 41
4.4.3 Determining Long-Term Demand . . . 43
4.5 Summary . . . 44

Chapter 5 Internal Adaptation . . . 46
5.1 Overview . . . 46
5.2 Soft Real-Time Scheduling . . . 49
5.2.1 The Scheduling Algorithm . . . 50
5.2.2 An Scheduling Example . . . 51
5.3 Reactive Internal Adaptation . . . 53
5.3.1 Per-job Adaptation . . . 53
5.3.2 Multi-job Adaptation . . . 56
5.4 Proactive Internal Adaptation . . . 58
5.4.1 Adaptation with Speed Schedule . . . 59
5.4.2 Speed Schedule for Ideal Processors . . . 62
5.4.3 Speed Schedule for Non-Ideal Processors . . . 68
5.4.4 Stability of Demand Distribution . . . 73
5.5 Summary . . . 76

Chapter 6 Implementation . . . 78
6.1 Hardware Platform . . . 78
6.2 Implementation of GRACE-OS . . . 80
6.2.1 Adding New System Calls . . . 80
6.2.2 Modifying the Processor Control Block . . . 83
6.2.3 Implementing the CPU Adaptor . . . 84
6.2.4 Implementing the Soft Real-Time Scheduler . . . 84
6.2.5 Modifying to Standard Linux Scheduler . . . 86
6.3 Implementation of Adaptive Multimedia Tasks . . . 87
6.3.1 Adaptive MPEG Decoder . . . 87
6.3.2 Adaptive H263 Encoder . . . 88
6.4 Summary . . . 89

Chapter 7 Experimental Evaluation . . . 91
7.1 Experimental Setup . . . 91
7.1.1 Experimental Applications . . . 91
7.1.2 Metrics . . . 93
7.2 Overhead . . . 95
7.2.1 Cost for Global Adaptation . . . 95
7.2.2 Cost for Internal Adaptation . . . 96
7.2.3 Cost for Real-Time Scheduling . . . 98
7.2.4 Cost for New System Calls . . . 99
7.3 Benefits of Global Adaptation . . . 101
7.3.1 Maximizing Utility . . . 103
7.3.2 Achieving Desired Lifetime . . . 105
7.3.3 Summary of Global Adaptation Results . . . 108

7.4 Benefits of Internal Adaptation 108
7.4.1 Energy Saving 110
7.4.2 QoS Support 112
7.4.3 Summary of Internal Adaptation Results 112
7.5 Summary 113

Chapter 8 Related Work 114
8.1 Soft Real-Time Scheduling 114
8.1.1 Statistical Scheduling 115
8.1.2 Overrun and Underrun Handling 116
8.2 QoS-Aware Application Adaptation 117
8.3 Energy-Aware CPU Adaptation 118
8.3.1 CPU Adaptation Mechanisms 119
8.3.2 Operating System Directed DVS 120
8.3.3 Compiler Assisted DVS 121
8.3.4 DVS with Discrete Speeds 122
8.4 Energy-Aware Application Adaptation 123
8.5 Coordination of Adaptation 123

Chapter 9 Conclusions and Future Work 126
9.1 Conclusions 126
9.2 Lessons Learned and Future Work 128

References 131

Vita 145

List of Tables

2.1 Speed-power relationship for an HP N5470 laptop. 12

3.1 Summary of global, per-application, and internal adaptations in GRACE. 23

5.1 Comparison between reactive and proactive internal adaptation. 77

6.1 Supported speed and voltage of the Athlon CPU. 78
6.2 New system calls for GRACE-OS 81
6.3 Sample code of an adaptive multimedia task 82
6.4 Modified process control block 83
6.5 Value of register `FidVidCtl` for different speeds. 84
6.6 High resolution timer to trigger soft real-time scheduling 85
6.7 Modification to the standard Linux scheduler 87

7.1 Experimental multimedia tasks. 92
7.2 QoS levels for the three multimedia codecs. 93
7.3 Desired lifetime for the single and concurrent runs. 106

List of Figures

1.1 Adaptation in various system layers: previous work adapts one or two layers at a time (a), while we consider coordinated cross-layer adaptation (b). 3

2.1 Cross-layer adaptation: Integrating the adaptation of CPU speed, operating system allocation, and multimedia quality. 15

3.1 The GRACE approach to trade off QoS for energy: Moving from fixed isolated adaptive layers a cross-layer adaptive system. 19
3.2 Architecture of the GRACE system: Each individual layer has a specific adaptor and monitor; the coordinator mediates the adaptation of all layers based their monitored information. 20
3.3 Hierarchical adaptation: GRACE uses three levels of adaptation with different scope and temporal granularity. 22
3.4 Architecture of GRACE-OS. 27

4.1 Dynamic programming algorithm to solve the global adaptation problems for the *maximum-utility* and *desired-time* policies. 35
4.2 The global adaptation protocol: The coordinator in the operating system makes the global decisions based on the system states collected from multiple layers. 37
4.3 Outline of prediction of long-term CPU demand for each QoS level of a task. . . . 39
4.4 Kernel-based cycle profiling: monitoring the number of cycles elapsed between each task's switch-in and switch-out during context switches. 40
4.5 Histogram-based estimation: the histogram approximates the cumulative distribution function of a task's cycle demand for a QoS level. 42
4.6 Determine the statistical, long-term cycle demand for each QoS level of a task based on its demand distribution. 44

5.1 Variations of instantaneous cycle demand of an MPEG video decoder at fine time granularity: Frame decoding may need more or less cycles than the long-term prediction (95th percentile of all frames). 47
5.2 The scheduling algorithm. 51
5.3 An example of the speed-aware, EDF-based scheduling algorithm. 52
5.4 Per-job adaptation to handle underrun and overrun. 55
5.5 Variations of instantaneous and statistical demand of an MPEG video decoder. . . . 56
5.6 Applying reactive internal adaptation and global adaptation at different time scales to handle CPU usage variations. 58

5.7 Example of speed schedule and corresponding speed scaling for job execution: the scheduler dynamically changes the speed during a job execution. 60
5.8 Scheduling of two tasks: The CPU speed changes during the task execution and in a context switch. 61
5.9 Adaptation based on the speed schedule: Each job starts slowly and accelerate as it progresses. 67
5.10 Comparsion of expected energy of (a) executing each frame at the coordinated speed and (b) adapting the speed based on the demand distribution. 68
5.11 Measured and ideal power on an HP N5470 laptop with an Athlon CPU: The measured power is obtained with an oscilloscope, while the ideal power is calculated by assuming that the power is proportional to the cube of the speed. 69
5.12 Dynamic programming algorithm to calculate the speed schedule for non-ideal processors with a discrete set of speed options. 72
5.13 Cycle usage and estimated demand distribution of `MPGDec`: its instantaneous cycle demands change greatly, while its demand distribution is much more stable. . . 74
5.14 Stability of demand distribution of other codecs: *toast* and *madplay*'s are stable, and *tmn* and *tmndec*'s change slowly and smoothly. 75

6.1 Power measurement with a digital oscilloscope. 79
6.2 Total power consumed by the laptop at different speeds: Each power value is the average of 2000 measurements. 79
6.3 Software architecture of GRACE-OS implementation. 80

7.1 Cost of global adaptation: the solid line shows the mean of six measurements and the error bars show the minimum and maximum of the six measurements. 95
7.2 Cost of changing the CPU frequency: the bars show the mean of 12 measurements and the error bars show the minimum and maximum of 12 measurements. 97
7.3 Cost of reactive internal adaptation: the bars show the mean of 50 measurements and the error bars show the minimum and maximum of 50 measurements. 98
7.4 Cost of proactive internal adaptation for constructing the speed schedule: the bars show the mean of 6 measurements and the error bars show the minimum and maximum of 6 measurements. 98
7.5 Cost of soft real-time scheduling: the bars show the average of 5,000 measurements and the error bars show the 95% confidence intervals. 99
7.6 Cost of new system calls: the bars show the mean of ten measurements and the error bars show the minimum and maximum of the ten measurements. 100
7.7 Comparing GRACE-OS with other systems for maximum-utility global adaptation: the bars show the mean of five measurements and the error bars show the minimum and maximum of the five measurements. 104
7.8 Comparing GRACE-OS with other systems for desired-lifetime global adaptation: the bars show the mean of five measurements and the error bars show the minimum and maximum of the five measurements. 107
7.9 Comparing different internal adaptation approaches: the bars show the mean of five measurements and the error bars show the minimum and maximum of the five measurements. 111

Chapter 1

Introduction

The design and implementation of *GRACE-OS*, an energy-efficient mobile multimedia operating system, is motivated by the emergence of multimedia-enabled mobile devices, such as camera phones, and their demands for supporting multimedia Quality of Service (QoS) under limited system resources, especially battery energy. In this chapter, we introduce the motivation of our research in mobile multimedia operating systems, discuss the challenging research problems, present the major features and contributions of GRACE-OS. Finally, we outline the rest of the thesis.

1.1 Motivation

Battery-powered mobile devices are becoming increasingly important platforms for processing multimedia data such as image, audio, and video. For example, we can already use a cell phone to take and send pictures and use an iPAQ pocket PC to watch TV. Compared to conventional desktop and server systems, such multimedia-enabled mobile systems need to save energy and hence extend the battery life while supporting multimedia QoS requirements. There is a conflict in the design goals for QoS provisioning and energy saving. For QoS provisioning, system resources often need to provide high performance, typically resulting in high energy consumption. For energy saving, system resources should consume low energy. As a result, the operating system of mobile devices needs to manage resources in QoS- and energy-aware manner and provides the flexibility to trade off QoS and energy based on the user's preferences.

1.1.1 What Is Available

Although the requirement of high QoS and low energy is challenging, now it becomes achievable due to the strong advances in the *adaptable* system layers, ranging from hardware to applications. First, system resources are being designed with the ability to trade off performance for energy. For example, mobile processors on the market today (such as Intel Pentium-M [51], AMD Athlon [5], and Transmeta Crusoe [98]) can already change the speed and power at runtime. Second, multimedia applications can gracefully adapt to resource changes while keeping the user's perceptual quality meaningful. That is, multimedia applications allow a tradeoff between output quality and resource demands. Finally, the operating system can also provide flexible resource management to support the tradeoff between QoS and resource demands or to balance the demands on different resources (e.g., CPU time and network bandwidth).

Based on the observation of adaptability, researchers have proposed *adaptation* approaches to address the high QoS and low energy challenge in mobile devices. Adaptation can happen in different layers from hardware to operating system to applications[1]. The hardware adaptation dynamically reconfigures hardware resources such as the processor to save energy while providing the requested resource service and performance [15, 45, 49, 53, 59, 102]. The operating system adaptation changes the policies of allocation and scheduling in response to application and resource variations [11, 34, 60, 75, 106, 110]. The application layer adaptation, possibly with the support of the operating system or middleware, changes the QoS parameters such as rate to trade off output quality for resource usage or to balance usage of different resources [41, 47, 61, 72, 78, 91].

The above adaptation approaches have been shown to be effective for both QoS provisioning and energy saving. However, most of them adapt only a single layer or two joint layers (e.g., the operating system and applications [31, 89] or the operating system and hardware [68, 83, 87]), as shown in Figure 1.1-(a).

[1]There is also a lot of research on adaptation in the network protocols [56, 55, 6]. In this thesis, however, we focus on three layers: hardware, operating system, and applications, in stand-alone mobile devices. Furthermore, we consider middleware such as Puppeteer [34] and Dynamo [75] as a part of the operating system.

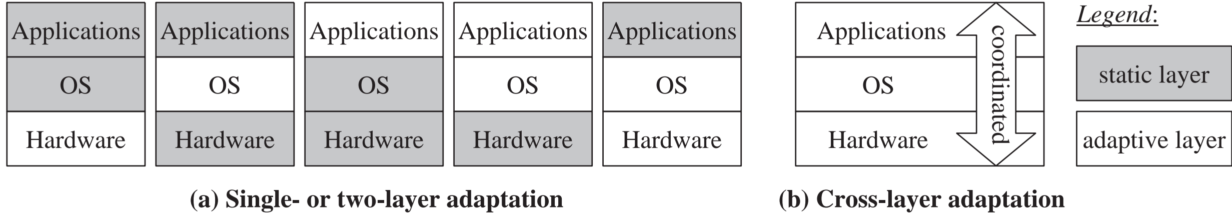


Figure 1.1: Adaptation in various system layers: previous work adapts one or two layers at a time (a), while we consider coordinated cross-layer adaptation (b).

1.1.2 What Is Missing

We argue that to trade off QoS for energy more efficiently, we should consider *cross-layer adaptation*, in which all layers adapt together in a coordinated manner, as shown in Figure 1.1-(b). The cross-layer adaptation is not a simple combination of the above adaptation techniques, as they exist in individual layers. There are at least four reasons:

1. Independent adaptive layers that are unaware of each other may result in adaptation conflicts or miss system-wide optimization opportunities. For example, when the CPU slows down to save energy, applications may increase their CPU demand.

2. Independent adaptations in various layers may cause instability (i.e., large fluctuation) of multimedia quality, which is annoying to the end user. For example, when an MPEG player adapts by increasing its frame rate, the total CPU demand correspondingly increases. This increase will trigger hardware adaptation to speed up the CPU. Such a hardware adaptation may result in more CPU resource available, and consequently may trigger the MPEG player to increase its frame rate again.

3. Various runtime scenarios require different adaptation objectives. For example, we may need to coordinate all layers to maximize application quality when resources are sufficient and minimize energy consumption when the battery is low. These adaptation objectives can be achieved only through a coordinated cross-layer coordination.

4. Adaptations in different layers have different benefits and cost. As a result, we need to trigger

different adaptations to balance the benefits and cost. For example, the operating system can adapt CPU allocation with low overhead to handle a small variation in application CPU demand. When the CPU is heavily overloaded, however, applications may need to degrade their quality and CPU demand.

Hence, if all adaptive layers deploy adaptation techniques simultaneously, we need to consider carefully their integration and coordination. In particular, there are two research problems: First, how to design each adaptive system layer. Second, given all adaptive layers, how to coordinate their adaptation for a system-wide optimization such as maximizing multimedia quality under the CPU and energy constraints.

1.2 GRACE-OS: An Energy-Efficient Multimedia OS

To address the problem of adapting multiple system layers together, the *Illinois GRACE (Global Resource Adaptation through CoopEration)* project is developing a cross-layer adaptation framework for mobile devices that primarily run multimedia applications. In the GRACE framework, all adaptive layers cooperate with each other to trade off QoS for energy based on the user's preferences, such as maximizing multimedia quality or achieving a desired battery life [4, 109]. GRACE is a multi-disciplinary project with members from different areas such as architecture, operating system, network, and coding.

In this thesis, we focus on the operating system of the GRACE framework. The operating system plays a key role in the cross-layer adaptation since it coordinates the adaptation of all layers based on the system-wide states, such as application requirements and resource availability, and adapts the process scheduling in the operating system layer. We next describe the research challenges addressed by GRACE-OS, give the overview of the solution of GRACE-OS, and present the major contributions made by GRACE-OS.

1.2.1 Research Problems

The operating system of mobile devices needs to manage resources in a QoS- and energy-aware manner. In particular, to enable the tradeoff between QoS and energy in a cross-layer adaptive system, the operating system needs to address the following two challenging problems:

- *How to coordinate multiple layers at different time granularity*. As discussed above, most of previous adaptation approaches consider only a single layer or two joint layers. More recently, some groups have also proposed cross-layer adaptation approaches [75, 79, 82, 90]. These related approaches, however, adapt only at coarse time granularity, e.g., when an application joins or leaves the system. To fully exploit the benefits of adaptation, we need to adapt multiple layers to system changes at different time granularity.

- *How to schedule applications when the hardware resources dynamically change*. Soft real-time scheduling is a common mechanism, typically with predictable resource allocation such as proportional sharing and reservation, to support multimedia QoS [20, 23, 29, 40, 76, 54, 88, 25]. Previous scheduling algorithms, however, often assume that the hardware resources are static (e.g., the CPU runs at a constant speed). The adaptive hardware brings new challenges for real-time scheduling, e.g., how to allocate and enforce processing time when the CPU speed changes dynamically.

1.2.2 Solution Overview

To address the above two challenges, we develop a novel mobile multimedia operating system, called *GRACE-OS*. GRACE-OS currently manages the CPU and energy resources and is being extended to manage other resources such as network bandwidth. To enable the tradeoff between QoS and energy, GRACE-OS adapts the CPU frequency and voltage in the hardware layer, process scheduling in the operating system layer, and multimedia quality in the application layer. By integrating and coordinating all these adaptations together, GRACE-OS solves the above two challenging problems as follows:

- To answer the first question on how to coordinate the adaptation in multiple layers at different time scales, GRACE-OS employs a *global* and *internal* adaptation hierarchy. Global adaptation coordinates the CPU hardware, operating system, and application layers together in response to large variations (such as application entry or exit) at coarse time granularity. The goal of global adaptation is to achieve user-specified system-wide optimization such as maximizing current multimedia quality (e.g., when recording an important video) or achieving a desired battery lifetime (e.g., when watching a two-hour movie).

  On the other hand, internal adaptation adapts a single layer to small variations at fine granularity, e.g., when an MPEG decoder changes CPU demand for different frames. Internal adaptation enables each layer to enforce the globally coordinated QoS with minimum energy. Limited within a single layer, internal adaptation may miss the system-wide optimization, but incurs much lower overhead than global adaptation. By combining global and internal adaptation together, GRACE-OS can achieve the benefits of the cross-layer adaptation with acceptable overhead.

- To answer the second question on how to schedule applications on dynamic CPU hardware, GRACE-OS extends traditional real-time CPU scheduling by adding another dimension—*speed*. That is, the scheduler decides *what speed to execute applications* in addition to when to execute what applications. This extended scheduling algorithm provides flexibility for enforcing the coordinated allocation when the CPU speed changes dynamically and adapting the speed to handle small variations without affecting multimedia quality.

  GRACE-OS makes these scheduling decisions based on the soft real-time CPU demand of multimedia applications and the total power consumed by the device at different CPU speeds. Multimedia applications present *soft real-time* resource demands: On one hand, unlike hard real-time applications, multimedia applications require only *statistical* performance guarantees (e.g., meeting 96% of deadlines). On the other hand, unlike best-effort applications, if multimedia applications complete a job (e.g., a video frame decoding) by deadline, the

actual completion time does not matter from the QoS perspective. The soft real-time nature of multimedia applications provides an opportunity to save energy without degrading QoS significantly.

1.2.3 Major Contributions

The major contributions of GRACE-OS are as follows:

- **Hierarchical adaptation framework**. GRACE-OS provides a hierarchical framework to integrate and coordinate the adaptation in different system layers of mobile devices to trade off multimedia quality against energy. This hierarchical framework consists of *global* and *internal* adaptation, balancing the benefits and cost of the cross-layer adaptation. Global adaptation coordinates all three layers in response to large system changes such as application entry and exit. The operating system performs the global coordination to achieve a system-wide optimization based on the user's preferences, e.g., maximizing multimedia quality under the CPU and energy constraints.

  Internal adaptation adapts each individual layer in response to small system changes. The operating system also monitors the system states, such as application CPU usage, and triggers internal adaptation in individual layers. GRACE-OS develops a set of algorithms for the integrated internal adaptation in the CPU hardware and operating system. These algorithms adjust the CPU speed to handle small variations in the CPU usage of individual applications. In doing so, GRACE-OS minimizes energy while enabling applications to operate at the globally coordinated quality level.

- **Speed-aware soft real-time scheduling**. Unlike previous real-time scheduling algorithms that often assume a constant CPU speed, GRACE-OS provides a soft real-time scheduling service to support multimedia QoS requirements on a variable-speed CPU. In particular, we extend traditional real-time scheduling with another dimension, *speed*, for mobile devices

with a variable-speed processor. That is, the scheduler decides *how fast to execute applications* in addition to when to execute what applications. This extended scheduling algorithm provides soft deadline guarantees to multimedia applications and provides flexibility for internal adaptation to change the CPU speed without affecting multimedia quality.

- **Kernel-based profiling and prediction**. GRACE-OS provides a kernel-based profiling technique to monitor the CPU usage of individual applications and uses a histogram-based algorithm to predict the probability distribution of cycle demand of individual applications. The demand distribution is used for both global and internal adaptation. First, the global adaptation allocates each application cycles based on its demand distribution, rather than the worst-case demand. This differs from most previous work that assumes known CPU demand but does not specifically address how to determine the demand. Second, the internal adaptation adjusts the CPU speed based on the demand distribution of each application to minimize energy while not affecting multimedia quality.

- **Implementation and case study**. We have implemented GRACE-OS in the Linux kernel and a prototype of the GRACE cross-layer adaptive system with adaptive CPU and video codecs. To the best of our knowledge, GRACE is the first real system that coordinates the adaptation in the CPU hardware, operating system, and application layers. The GRACE prototype is a concrete case study of a generic cross-layer adaptation framework, where many components and parameters at each layer can adapt. The investigation of this prototype shows the impact of cross-layer adaptation on QoS and energy and justifies our hierarchical adaptation approach.

Our experimental results show that GRACE-OS efficiently trades off QoS against energy based on the user's preferences with acceptable overhead. Specifically, compared to previous systems that adapt only some system layers, GRACE-OS's global adaptation achieves the user-desired battery lifetime and saves energy by up to 59% while providing better or the same multimedia quality. Compared to previous systems that adapt only at coarse time

granularity, GRACE-OS's internal adaptation saves energy by 2% to 8.9% without affecting multimedia quality.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 introduces the models of adaptive CPU and multimedia applications, as well as system changes GRACE-OS adapts to. Chapter 3 presents an overview of the GRACE cross-layer adaptive system and describes the role of the operating system in the GRACE system. Chapter 4 and 5 discuss the two major operations, global and internal adaptation, of GRACE-OS. In particular, Chapter 4 presents the global coordination problem for a system-wide optimization, shows that the coordination problem is NP hard, and describes how to heuristically solve the NP hard coordination problem. Chapter 5 introduces an energy-aware soft real-time scheduling algorithm, which enforces the coordinated allocation on a variable-speed CPU, motivates the need to handle small variations in the CPU usage of applications, and presents a set of internal adaptation algorithms in the CPU hardware and operating system. Chapters 6 and 7 present the implementation and experimental results of GRACE-OS, respectively. Chapter 8 compares GRACE-OS with related work. Finally, Chapter 9 summarizes the thesis and describes the future work.

Chapter 2

System Models

GRACE-OS manages CPU and energy resources for a cross-layer adaptive system. This chapter introduces adaptive models in the hardware, operating system, and application layers for GRACE-OS. In particular, we discuss what knobs we can tune in each layer to trade off QoS and energy. We then describe changes in mobile systems; these changes trigger the cross-layer adaptation. It is important to stress that although GRACE-OS is currently built on these specific models, it can be extended to support other adaptive models such as adaptive architecture [49] and network protocols [6, 56]. Such an extension is a part of our future work.

2.1 CPU Frequency and Voltage Adaptation

In the hardware layer, we consider mobile devices with a single adaptive CPU that supports multiple speeds (frequencies), $\{f_1, \cdots, f_K\}$, trading off performance for energy. We currently focus on reducing CPU energy for two reasons: First, the CPU is one of the highest energy consumers in our target stand-alone mobile devices. For example, we measured energy consumption of a laptop and found that the CPU consumes 15% to 52% of the total energy, depending on the application workload. Second, mobile processors on the market today (e.g., Intel Pentium-M [51], AMD Athlon [5], and Transemta Crusoe [98]) allow software, typically the operating system, to change the speed through the Advanced Configuration and Power Interface (ACPI) standard [26], thereby enabling a cross-layer system control.

In general, there are two approaches to reduce CPU energy consumption. The first one is dynamic power management (DPM) [15], which puts the idle processor into the lower-power sleep state. The second approach is dynamic frequency/voltage scaling (DVS) [81], which lowers the operating speed and voltage of the active processor. DPM, however, is not suitable for our targeted multimedia applications, which access the CPU periodically. As a result, the idle interval in each period is often shorter than the wake-up time (e.g., 160 milliseconds for StrongARM SA-1100 [15]); so the processor cannot be put into the sleep state in the short idle intervals. We therefore consider only the DVS approach in this thesis. The basic idea behind DVS is as follows.

The CPU power consumption typically consists of three major parts: the dynamic power, short circuit power, and leakage power, as shown follows:

$$\underbrace{C \times f \times V^2}_{\text{dynamic power}} + \underbrace{V \times I_{sc}}_{\text{short circuit power}} + \underbrace{V \times I_{leak}}_{\text{leakage power}} \tag{2.1}$$

where C is the loading capacitance, f is the speed, V is the voltage, I_{sc} is the short circuit current, and I_{leak} is the leakage current [24]. When the speed decreases, the CPU can operate at a lower voltage and thus reduce its power. Further, the CPU power is generally a convex function of the speed. Consequently, the CPU energy (i.e., the product of the power and time) also decreases as the speed decreases even at the cost of longer execution time. In particular, for ideal processors, we assume that their power is dominated by the dynamic power and the voltage is proportional to the speed; that is, the CPU power is proportional to the cube of the speed.

The above assumptions may not hold for some mobile processors such as Intel Pentium-M and AMD Athlon, whose power is not proportional to the cube of the speed. Furthermore, a lower speed may increase the energy consumption of other resources such as memory and display. For example, when the CPU operates at a lower speed, an application needs to run for a longer time; consequently, it needs to use the memory for a longer time, often increasing the memory energy. Since our goal is to save the total energy of the whole device, we are more interested in the total power of the device. In general, the relationship between the speed f and the total device power

Table 2.1: Speed-power relationship for an HP N5470 laptop.

Speed f (MHz)	300	500	600	700	800	1000
Power $p(f)$ (Watt)	22.25	25.84	28.24	31.05	35.44	39.06

$p(f)$ can be obtained via measurements. For example, we used an Agilent 54621A oscilloscope to measure the power of an HP Pavilion N5470 laptop at different CPU speeds, and Table 2.1 shows the results. Without loss of generality, we assume that the total power decreases as the CPU speed decreases. Otherwise, we will never run the CPU at the speed that consumes more power but provides lower performance than another speed.

2.2 Application Quality Adaptation

We consider each process or thread in a multimedia application as a *periodic task*[1] that manipulates media streams, e.g., decodes video frames periodically. Each task consumes CPU resource and generates an output to other tasks or the end user. *Adaptive* tasks can trade off output quality for CPU demand [19, 89]. Unlike best-effort tasks such as TCP communication, multimedia tasks often adapt in a discrete manner and hence support a discrete set of QoS levels, $\{q_1, ..., q_m\}$. For example, an audio encoder can change the sample rate, and a multi-layered video player can decode different number of layers. The QoS levels may correspond to the sample rate for the audio encoder and the decoded layers for the video player.

Utility functions are a flexible tool to capture task adaptation behavior and are commonly used in previous literature to optimize QoS among multiple tasks [19, 62, 82, 89]. In this thesis, we also use utility, $u(q)$, to measure the perceptual quality at a QoS level from the user's point of view. In general, utility definition is user-specific; e.g., different users may perceive different quality for the same task running at the same QoS level. Some objective or subjective assessment techniques [57, 69, 74] are helpful to define the utility from the user's point of view. Note that

[1]Although not explicit in this task model, we can handle aperiodic tasks such as best-effort and interactive applications with a periodic server [28, 106].

this thesis focuses on the quality levels with different utility and CPU demand. Applications may change quality finely around these quality levels without affecting utility [91]. Such changes can be handled by applications internally, which is oblivious to the operating system.

Since we target to mobile devices that are often user-centric, we assume that the user defines utility functions for each task and normalizes utility among different tasks. Consequently, we use *total utility* to measure the overall quality of all concurrent tasks. Specifically, if there are n tasks and each task runs at the QoS level q_i with utility $u_i(q_i)$, then the total utility is

$$\sum_{i=1}^{n} u_i(q_i) \tag{2.2}$$

The utility functions may also change overtime, e.g, when the user changes his/her focus of concurrent applications. This can also be handled by the GRACE cross-layer adaptation framework, but requires the user's interaction, which is out of the scope of this thesis.

2.3 Operating System Allocation Adaptation

To provide a certain output quality, each periodic task releases a job (e.g., a frame decoding) every period. The released job has a soft deadline, typically defined as the end of the period, and consumes a certain amount of CPU cycles during its execution. By *soft* deadline, we mean that the job should, but does not have to, complete by this time. In other words, a job may miss its deadlines. Multimedia tasks need to meet some percent of job deadlines since they present soft real-time performance requirements.

More formally, for each QoS level of a task, we represent its corresponding CPU demand by a triple-tuple (ρ, P, C):

- **Statistical performance requirement** ρ denotes the probability that a task should meet job deadlines; e.g., if $\rho = 0.96$, then the task needs to meet 96% of deadlines. In general, the application developer or user can specify the parameter ρ, based on application characteristics

(e.g., audio streams have a higher ρ than videos) or user preferences (e.g., a user may tolerate some deadline misses when the CPU is overloaded or when the battery energy is low).

- **Period** P is the minimum interval of consecutive job release [65]. The period of a task can be directly calculated from its application QoS level q, given, e.g., by the parameter 'rate', via the equation $P(q) = \frac{1}{rate}$.

- **Statistical cycle demand** C is the number of cycles the task demands to meet its statistical performance requirement. For example, if an MPEG decoder has $\rho = 0.96$ and 96% of its frame decoding needs less than 5×10^6 cycles, then its parameter C is 5×10^6. We assume that the number of cycles demanded by a job is roughly constant for different speeds. This assumption holds for typical multimedia applications since the number of cycles they spend on memory stalls is often small and the remaining CPU performance scales with CPU speed [70, 48]. If the number of cycles demanded by a job changes largely with the speed, e.g., due to cache [93], we can use an average number of cycles at different speeds as the parameter C.

  In general, the statistical cycle demand of a task is in between its average and worst-case demand. We characterize task cycle demand based on the statistical, rather than the worst-case, demand for two reasons. First, it is difficult to precisely estimate the worst-case demand of multimedia tasks, because several low-level operating system mechanisms (e.g., caching and interrupts) and semantics of multimedia content (e.g., scene changes in MPEG video) introduce an uncertainty in task execution. The statistical cycle demand, however, can typically be predicted via online or off-line profiling [67, 99, 107] and is often stable across the processed stream. Second, allocating cycles based on the statistical demand of individual tasks delivers statistical performance guarantees, which is sufficient for soft real-time multimedia applications.

2.4 Cross-Layer Configuration

When we put the adaptive CPU, operating system, and application models together, we get a cross-layer adaptation problem, as shown in Figure 2.1. Specifically, we need to tune the CPU speed in the hardware layer, the QoS level for each task in the application layer, and the CPU allocation for each task in the operating system layer. The major problem addressed in this thesis is given these adaptation parameters, how to adapt them in a coordinated manner.

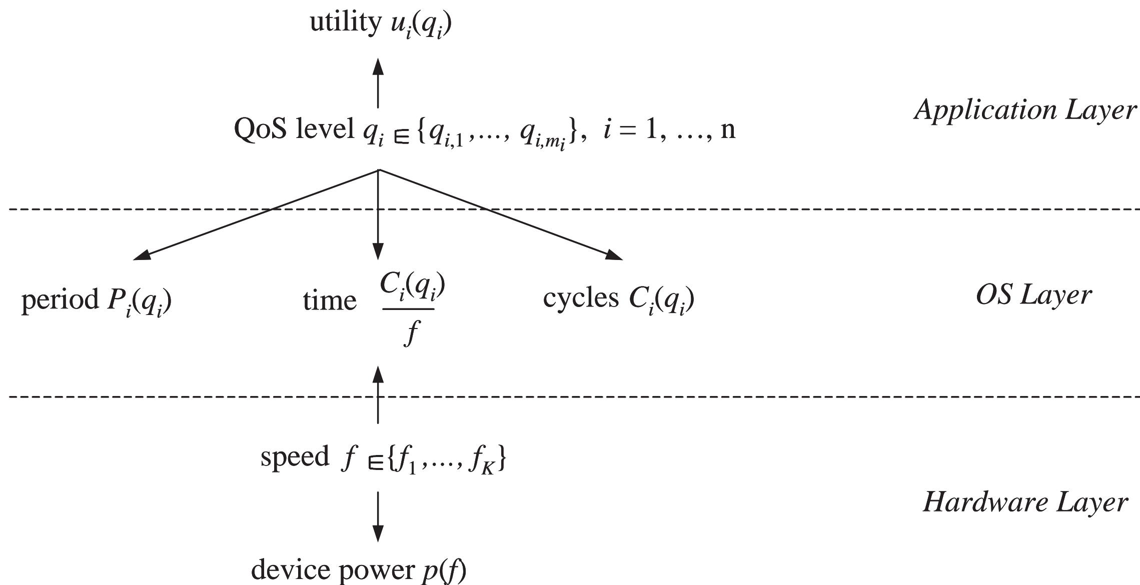


Figure 2.1: Cross-layer adaptation: Integrating the adaptation of CPU speed, operating system allocation, and multimedia quality.

2.5 Adaptation Triggers

At runtime, mobile multimedia systems must adapt to system or application changes. There can be several reasons for a change in resource demands and/or resource availability, serving as triggers for adaptation. In GRACE-OS, we consider follows[2]:

[2]Other adaptation triggers include, for example, changes of context, user preference, and wireless network bandwidth. As a part of our future work, we will consider these triggers.

- Changes in manipulated media streams which cause fluctuation of CPU usage, e.g., changes of MPEG frame type (I, P, B) or scene changes.

- Joining and leaving of multimedia tasks which cause large changes in total CPU demand.

- Low energy availability; i.e., when the battery of the mobile device cannot last for the desired lifetime at the current rate of energy consumption. The desired lifetime is the time until which the battery needs to last without recharging. This is often known by the user of a mobile device (e.g., the time length of a flight or a lecture presentation) [35, 27].

All the above adaptation triggers represent a change in resource availability and/or demand, but may occur at different time scales. Changes of the first type may happen frequently in a short-term (e.g., in tens of milliseconds or per-job) or in the medium-term (e.g., in several seconds or across multiple jobs for a scene change). In contrast, changes of the second and third types occur infrequently in long-term intervals (e.g., in minutes or per-task).

Chapter 3

GRACE-OS: An Overview

GRACE-OS is done as a part of the Illinois GRACE project, which is developing a cross-layer adaptation framework for mobile multimedia systems [4]. In the GRACE framework, all system layers, ranging from hardware to applications, are adaptive and cooperate with each other to achieve a system-wide optimization, such as maximizing multimedia utility under the constraints of CPU time and battery energy. The operating system is responsible for coordinating all adaptive layers; in particular, it allocates applications resources and enforces the allocation for the system-wide optimization.

In this chapter, we briefly introduce the GRACE system as the background of GRACE-OS. We then describe the role of the operating system in GRACE and present the architecture of GRACE-OS. In the next two chapters, we will describe GRACE-OS in more detail.

3.1 Background— The GRACE System

Our target systems are multimedia-enabled mobile devices such as camera phones. Such mobile systems present both new *challenges* and new *opportunities*. New challenges arise because these multimedia-centric devices need to support multimedia QoS and save battery energy at the same time. There is an inherent conflict behind these two design goals. On one hand, for QoS support, system resources such as the CPU should provide high performance, typically resulting in high power consumption. On the other hand, for energy saving, however, system resources should

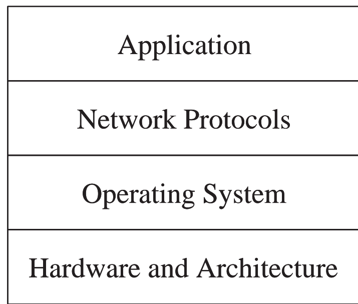
consume low power, often operating at low performance.

New opportunities lie in the *adaptability* of the hardware and software components of mobile devices. First, hardware resources are being designed with the ability to trade off performance for energy. For example, mobile processors, such as Intel Pentium-M [51], AMD Athlon [5], and Transmeta Crusoe [98], can already change the operating speed and power consumption at runtime. Second, multimedia applications require *soft* real-time performance guarantees and can gracefully adapt performance and resource demands based on the available resources. Finally, the operating system and network protocols can also provide flexible resource management, e.g., by using different scheduling policies or by trading off CPU time for network bandwidth.

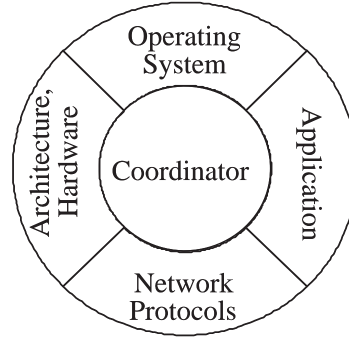
Based on the above two observations, namely, new challenges and new opportunities, researchers have introduced QoS- and energy-aware adaptation into various system layers, such as hardware [45, 49, 53, 59, 102], operating system [11, 34, 60, 75, 106, 110], network protocols [56, 55, 6], and applications [41, 47, 61, 72, 78, 91]. These adaptation techniques have been shown to be effective for both QoS provisioning and energy saving. However, most of the prior adaptation work focuses on adapting a single layer or two joint layers at a time.

We believe that a key to meeting the challenges of our target mobile systems is to design *all* system layers with an ability to *adapt* in response to system changes. Further, to reap the full benefits of these adaptations, all system layers must *cooperate* with each other to determine a system-wide globally optimal configuration. The *Illinois GRACE— Global Resource Adaptation through CoopEration—* project is designing such a cross-layer adaptive system [4]. Figure 3.1 captures the vision of the GRACE project. Figure 3.1-(a) shows the current systems with mostly fixed and isolated system layers. Figure 3.1-(b) shows the vision where all system layers cooperatively adapt as a community to trade off QoS for energy based on the user's preferences, such as maximizing multimedia QoS or achieving a desired battery life.

Currently, the GRACE system considers adaptation in the hardware layer for the CPU (e.g., voltage and frequency scaling and architecture adaptations) and wireless interface card (e.g., adapting transmission power), network protocol layer (e.g., adapting the reliability methods among



(a) Current Layered Adaptation



(b) GRACE Cross-Layer Adaptation

Figure 3.1: The GRACE approach to trade off QoS for energy: Moving from fixed isolated adaptive layers a cross-layer adaptive system.

ARQ, FEC, or a hybrid for reliability), the CPU and the network scheduler in the operating system (adapting CPU and bandwidth allocation), and multimedia applications (e.g., changing video encoding algorithm to trade off computation, communication, and energy). We next describe the architecture and major operations of GRACE for coordinating the above adaptations.

3.1.1 GRACE Architecture

A key challenge in a cross-layer adaptive system is to enable the global communication among different layers while preserving the existing independence of different layers. In particular, the design of the GRACE system has the following two goals:

1. GRACE must perform the cross-layer optimization without exposing implementation internals of a layer to other layers. That is, the coordinator is only responsible for setting a global contract among different layers (e.g., the utility each task should provide and the performance the CPU should provide). The coordinator does not care how an optimal configuration is reached within individual layers (e.g., how a task provides the coordinated utility).

2. GRACE must localize adaptation decisions specific to a layer within that layer. That is, each layer adapts internally without the knowledge of the internals of other layers. Each layer is also free to adapt internally as long as they do not violate the globally coordinated contract.

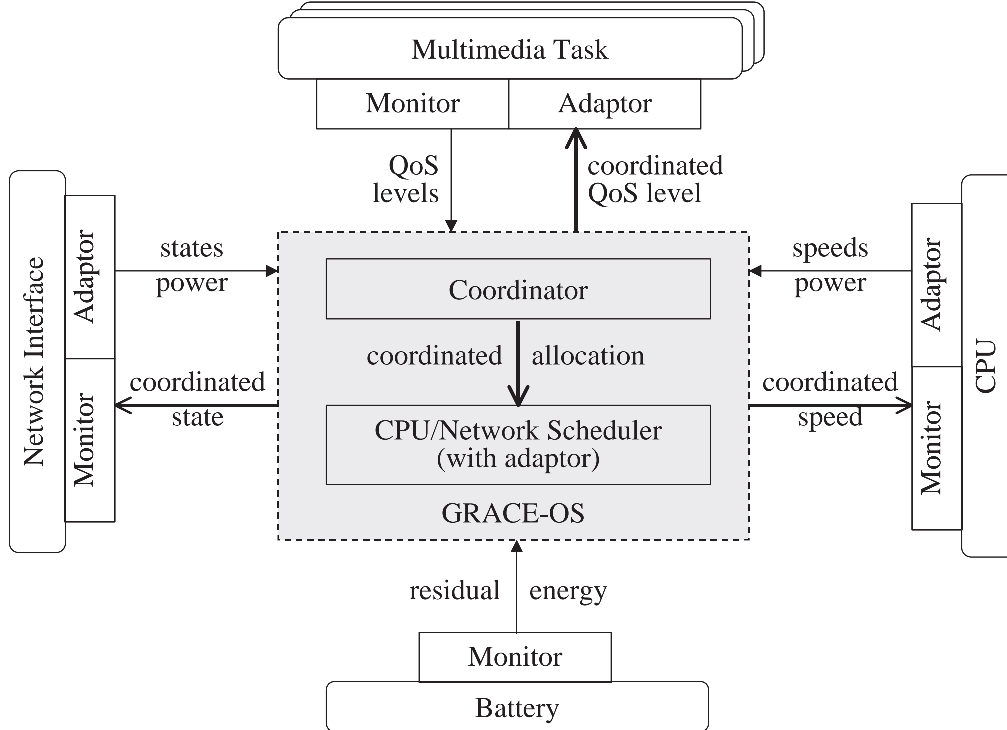


Figure 3.2: Architecture of the GRACE system: Each individual layer has a specific adaptor and monitor; the coordinator mediates the adaptation of all layers based their monitored information.

For example, the CPU can internally adapt its speed and architecture such as instruction window size to minimize energy while providing the coordinated performance.

To meet the above design goals, we modularize the adaptation of each individual layer. Figure 3.2 shows the architecture of the GRACE system, where the coordinator mediates the adaptation of all layers and each layer has a monitor and adaptor specific to that layer. We next describe each component in the architecture:

- **Monitors**. Each layer has a specific monitor, which measures the resource availability (e.g., residual energy and available network bandwidth) or resource usage (e.g., power consumption of the wireless interface card and a task's CPU and bandwidth demands).

- **Adaptors**. Each layer has an internal adaptor, which tunes the operating parameters of the corresponding layer (e.g., the speed for the CPU and the QoS parameters such as rate for a multimedia task).

- **Coordinator**. There is one coordinator in the system. The coordinator is responsible for coordinating the adaptation of all adaptive layers to achieve a system-wide optimization. The coordinator makes the coordination based on information monitored in each layer— the QoS levels of each task, the operating states and power of the CPU and wireless interface card, and the available battery energy.

  The coordinator is implemented as part of the operating system since the operating system has the access to full knowledge of the system states, such as resource availability and application requirements, and hence can make the system-wide decisions.

- **Schedulers**. The CPU and network schedulers schedule multimedia tasks to share the CPU and network resources. They enforce the global coordination to let tasks provide the utility expected by the coordinator and let hardware resources consume energy expected by the coordinator.

3.1.2 Adaptation Hierarchy

Three key questions in a cross-layer adaptive system are *what layers* to adapt, *when* to adapt them, and *how* to adapt them. An ideal cross-layer adaptive system would adapt all system layers together upon any changes in the system. In this way, it can always find a system-wide optimization. In practice, however, such an ideal system is infeasible since it would incur unacceptably large overhead. For example, consider that the CPU architecture may have tens to hundreds of possible configurations and an adaptive video encoder may use tens of possible encoding algorithms. Exploring all the combinations of the hardware and software configurations is expensive or even impossible. One approach to reducing reduce the overhead is to adapt locally within each individual layer. This approach, however, may result in poor or even conflicting configurations in different layers since each layer is unaware of other adaptive layers.

It is therefore important to balance between the scope and temporal granularity of adaptation in the cross-layer adaptation. GRACE takes a *hierarchical* approach to solve this problem: GRACE

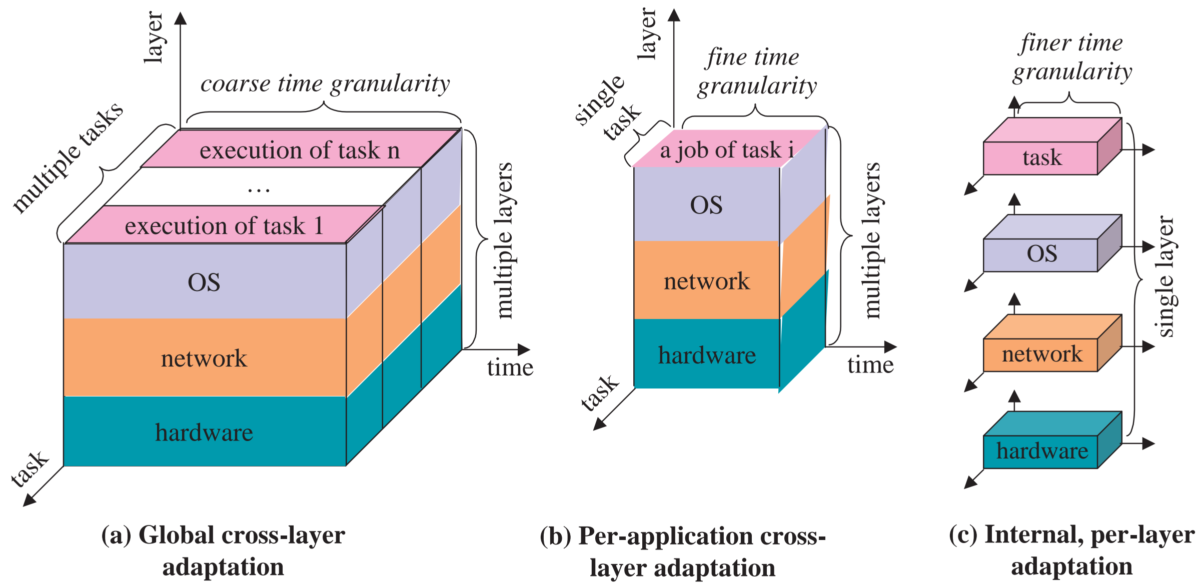


Figure 3.3: Hierarchical adaptation: GRACE uses three levels of adaptation with different scope and temporal granularity.

performs expensive global adaptations occasionally and limited-scope but inexpensive adaptations frequently. The purpose of the adaptation hierarchy is to achieve most of the benefits of cross-layer adaptation with acceptable overhead.

Specifically, GRACE identifies and supports three levels of adaptation, which increasingly adapt at finer scope and temporal granularity (Figure 3.3):

- *Global* adaptation adapts all system layers in response to large system changes (e.g., when a task joins or leaves the system).
- *Per-application* adaptation adapts a multimedia task and other system layers when the task starts a new job.
- *Internal* adaptation adapts only a single system layer, possibly at a granularity finer than a job (e.g., every packet in the network or every scheduling slice in the scheduler).

Both global and per-application adaptations are cross-layer adaptation and are performed by the coordinator. However, the coordinator invokes them at different time granularity and makes decisions based on different information. Specifically, global adaptation is based on the long-term

Table 3.1: Summary of global, per-application, and internal adaptations in GRACE.

Adaptation	what layers	when	how
Global	all layers	upon large changes	cross-layer multi-application coordination
Per-application	all layers	start of a job	cross-layer single-application coordination
Internal	single layer	upon small changes	enforcing coordinated decisions

prediction of the resource availability and demands, while per-application adaptation is based on the short-term (i.e., the next job) prediction.

Table 3.1 summarizes the answers of the three adaptations to the key questions in cross-layer adaptation. In combination, the three levels of adaptation are able to respond to all types of changes in resource demands and resource availability in mobile devices. We next describe the three adaptation in more detail.

Global Adaptation. Global adaptation coordinates all system layers. Its goal is to allocate system resources (CPU time, network bandwidth, and energy) among all tasks for a system-wide optimization such as maximizing the total utility of all tasks under the CPU and bandwidth constraints or achieving a desired battery lifetime. As a result, global adaptation happens when the resources need to be reallocated in response to large system changes such as task entry or exit.

The coordinator in the operating system performs the resource allocation by comparing the combinations of all possible configurations of the different system layers and choosing the combinations that achieve the system-wide optimization. By doing so, the coordination defines the utility and resource allocation for each task and the performance and power consumption of the hardware resources. This is a global contract among all layers, which must be respect by the subsequent internal adaptations.

The long time interval between global adaptations, however, implies its configuration choices could be sub-optimal in short time intervals. The reason is that the long-term resource predictions are based on average statistical behavior and hence may not be accurate for short-term variations between the interval of global adaptations. As a result, the consequent configuration choices may not be optimal due to the small variations. The per-application and internal adaptations compensate

for this sub-optimality as discussed next.

Per-Application Adaptation. Per-application adaptation is invoked when a task starts a new job. At this time, there is more accurate information about the task's resource demand than available to the global adaptation (e.g., the resource demand for the next job is well correlated to the demand of the last few jobs, and hence can be derived by maintaining limited history). Similarly, information on currently available resources is also more accurate. As a result, at the start of a job, a more informed choice can be made for the job. However, it is infeasible to perform a global adaptation at the start of each job based on the more accurate information since the overhead may be too large. Therefore, the per-application adaptation does not attempt to reallocate resources among different tasks. Instead, it only adapts the current task and other system layers and tries to minimize energy during the job execution while providing the utility expected by the global adaptation.

It may be possible for the per-application adaptation to increase the utility for the next job; however, we do not allow this since it could potentially introduce rapid fluctuations in the quality of the multimedia stream, which could be annoying to the user.

Internal Adaptation. Internal adaptation adapts a single layer to enforce the coordinated contract in response to small and frequent changes (e.g., when a task changes the CPU demand for different jobs). Since internal adaptation does not need to consider a cross-product of configurations of different layers, it is significantly more efficient and can potentially be invoked at a very fine time granularity. Internal adaptation is useful in many ways.

- First, recall that CPU and network bandwidth allocations are made globally assuming an average resource demand for each job. However, a given job of a task may underrun or overrun these allocations. The wireless channel quality may also change temporarily. In response to such variations, the CPU and network schedulers can adapt by redistributing the CPU and bandwidth allocation more optimally. For example, if a job underruns, the CPU scheduler can distribute its residual time to overrunning jobs of other tasks.

- Second, internal adaptation can respond to variations in resource usage and resource avail-

ability that occur *within* a given job. For example, different parts of the job may use different function units of the CPU. Under-utilized units could be deactivated for saving energy, while providing the requested CPU performance.

- Third, during the process of global and per-application adaptation, a system layer may apply an internal adaptation process to determine its minimal energy configuration. This can significantly reduce the overhead of the cross-layer adaptation. In particular, internal adaptation allows each system layer to locally integrate the effect of any intra-job internal adaptations; exposing those adaptations to the global or per-application adaptation would expose the internals of the individual layers.

3.2 Overview of GRACE-OS

In the previous section, we described the GRACE cross-layer adaptive system. While it is important to design each system layer to be adaptive, this thesis focuses on how the operating system supports the cross-layer adaptation given the adaptability of each layer. In particular, we design, implement, and evaluate a novel operating system, called GRACE-OS, for the GRACE cross-layer adaptive system. GRACE-OS coordinates the adaptation of the CPU speed in the hardware layer, CPU scheduling in the operating system layer, and multimedia quality in the application layer. We next describe the roles of the operating system in the GRACE system and present the architecture of GRACE-OS.

3.2.1 Operating System Roles in GRACE

From the operating system point of view, the cross-layer adaptation is an extended scheduling problem. In particular, given the adaptability of the CPU hardware and multimedia tasks, the operating system needs to decide *what QoS level for each task* (i.e., application adaptation) and *what execution speed for each task* (i.e., CPU adaptation) in addition to when to execute what tasks,

which is often determined by the traditional scheduling algorithm. By making these decisions, the operating system seeks to trade off QoS for energy based on the user's preferences such as maximizing the current QoS or achieving a desired battery life.

To make these decisions, GRACE-OS plays the following roles in the GRACE cross-layer adaptive system:

1. **Coordinator**. GRACE-OS coordinates all system layers to determine a system-wide optimal configuration during the global adaptation. It decides the coordinated QoS level for each task in the application layer, the coordinated CPU allocation for each task in the operating system layer, the coordinated CPU speed and power int the hardware layer.

2. **Real-time Scheduler**. GRACE-OS enforces the global coordination to enable each task to provide the coordinated QoS. This enforcement includes two steps, predictable CPU allocation and scheduling. Unlike previous scheduling algorithms, GRACE-OS needs to perform the scheduling on a variable-speed processor.

3. **Battery and Task Monitors**. GRACE-OS monitors the residual energy of the battery. It also monitors the runtime CPU usage and performance (e.g., the deadline miss ratio) each individual task. This monitoring are used for two purposes. First, multimedia tasks can use the monitoring to determine their CPU demand for each QoS level. Second, the operating system can trigger internal or global adaptation based on the monitored status.

4. **Scheduling and Speed Adaptor**. GRACE-OS performs internal adaptation in the operating system and hardware layers to handle small changes in the system. The purpose of these two internal adaptations is to enable each task to provide the utility expected by the coordinator with minimum energy.

   Note that we put the CPU speed adaptor into the operating system, rather than the hardware layer, since GRACE-OS needs to minimize CPU energy without affecting application performance. So, the speed adaption is tightly integrated with real-time scheduling in the op-

erating system. In the rest of this thesis, we use internal adaptation to indicate the integrated adaptation in the operating system and hardware layers unless specified otherwise.

3.2.2 The GRACE-OS Architecture

Figure 3.4 illustrates the architecture of GRACE-OS, which includes four major components mentioned above. In particular, the monitor observes the energy availability and provides the CPU demand of individual tasks to the coordinator and scheduler. The coordinator makes global adaptation to determine the QoS level for each task in the application layer, the CPU allocation in the operating system layer, and the CPU speed in the hardware layer. The scheduler enforces the globally coordinated decisions by scheduling tasks in a real-time manner. It also invokes internal adaptation in the operating system and hardware layers in response to variations in CPU usage. The speed adaptor changes the CPU speed and power based on the decisions made in the global and internal adaptation.

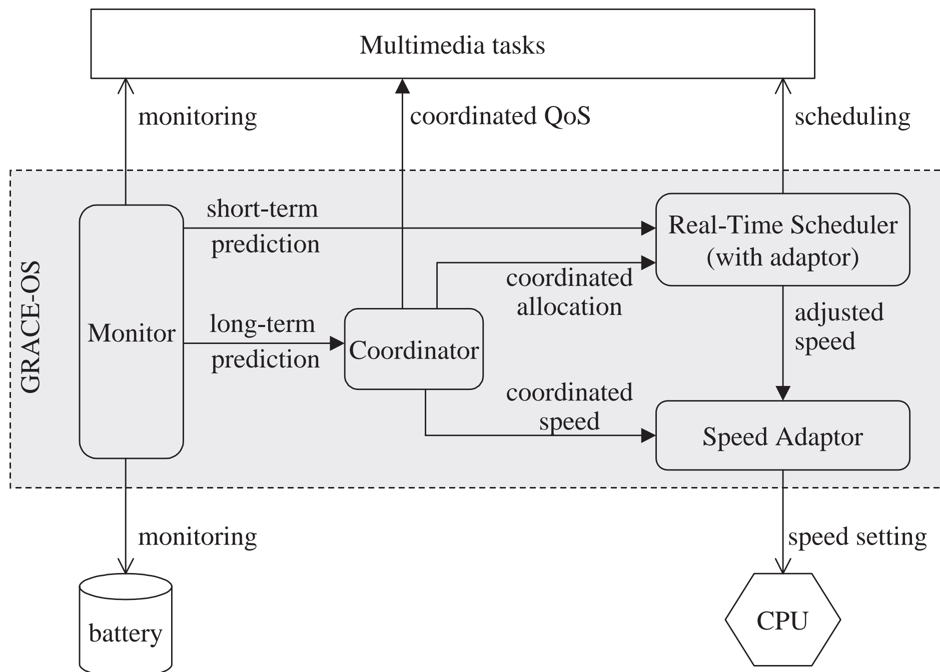


Figure 3.4: Architecture of GRACE-OS.

Operationally, GRACE-OS performs two major operations, global and internal adaptation. The

former handles large system changes such as task entry and exit, while the latter handles small variations in CPU usage of individual tasks. Note that we do not discuss per-application adaptation in GRACE-OS for the following reason. Per-application adaptation is often application-specific (e.g., how a video encoder can adapt its encoding algorithm to trade off CPU demand for QoS in a small range of the globally coordinated QoS level), while GRACE-OS provides general support for cross-layer adaptation. This these therefore focuses on global cross-layer adaptation and internal adaptation in the operating system and CPU hardware.

3.3 Summary

In this chapter, we introduced the GRACE cross-layer adaption framework, which adapts all system layers, ranging from hardware to applications, to system changes for a system-wide optimization. GRACE is a multi-disciplinary project with members from different areas such as architecture, network, coding. While other members are working on the design of each adaptive layer, we focus on how the operating system supports the cross-layer adaptation.

This chapter also described the roles of operating system in the GRACE system and presented the architecture of GRACE-OS to perform these roles. Operationally, GRACE-OS performs global and internal adaptations to handle various system changes to trade off QoS for energy. In the next two chapters, we will describe these two adaptations in more detail.

Chapter 4

Global Adaptation

In the GRACE cross-layer adaptive system, all system layers (the CPU hardware, operating system, and multimedia tasks) can adapt to changes in the system. GRACE-OS invokes global adaptation in response to large system changes at coarse time granularity, for example, when a task joins or leaves the system. The goal of global adaptation is to coordinate all adaptive layers to achieve a user-specified system-wide optimization. To do this, the coordinator considers all combinations of task QoS level, CPU allocation, and CPU speed and chooses the combination that achieves the system-wide optimization.

The outputs of the global adaptation are configurations of all layers, i.e., the QoS level for each task in the application layer, the CPU allocation for each task in the operating system layer, and the CPU speed and power in the hardware layer. The coordinated CPU allocation is based on the long-term prediction of CPU demand of individual tasks.

In this section, we first introduce two representative policies, maximizing the current multimedia utility and achieving a desired lifetime, for the global adaptation. We then formulate these two coordination policies as constrained optimization problems, show that these two optimization problems are NP hard, and present a heuristic algorithm to solve these problems. After describing the algorithm, we describe the communication protocol among different layers during the global adaptation. Finally, we discuss how to automatically predict the long-term CPU demand of individual tasks, which is used in the global adaptation.

4.1 Global Adaptation Problem

Our target multimedia-enabled mobile devices present two challenges, QoS provisioning and energy saving, at the same time. It is therefore important to trade off multimedia QoS for energy to achieve a system-wide optimization based on the preference of the end user. We refer to the user's preference as *coordination policy*. The user may have different coordination policies for different scenarios, for example, maximizing multimedia quality when the battery is high and minimizing the device power consumption when the battery is low. In GRACE-OS, we consider the following two representative policies:

- *Maximum-utility*: The user first wants to maximize the total utility of all tasks, for example, when she/he is recording important audio and video. The secondary goal is to minimize the CPU speed and power consumption while maximizing the total utility.
- *Desired-time*: The user first wants to last the battery for a desired lifetime, for example, when she/he wants to finish a two-hour movie before the battery runs out. The secondary goal is to maximize the total utility of all tasks while achieving the desired battery life.

These two policies both maximizes multimedia QoS, but treat energy differently. The policy *maximum-utility* tries to minimize the energy consumption of the device, while the policy *desired-time* considers energy as a resource constraint when maximizing QoS.

More formally, let E be the remaining battery energy, which is provided by the battery monitor (Figure 3.4), and T be the remaining lifetime (the interval from now to the desired lifetime), which can be specified by the applications or the user, e.g., how long applications should run before recharging the battery. The energy constraint is that the energy consumed by the device in the remaining time is no more than the remaining energy. That is,

$$\int_0^T p(f(t))dt \leq E \tag{4.1}$$

where $f(t)$ is the CPU speed at time t and $p(f)$ is the total power consumed by the whole device when the CPU runs at speed f. To validate this constraint, we need to know the CPU speed from now to the desired lifetime. This future knowledge is often not available in a dynamic mobile device. We therefore simplify the energy constraint by checking if the battery can last for the desired lifetime if the CPU will run at a same speed in the future. That is,

$$p(f) \times T \leq E \tag{4.2}$$

This inequation can be tested without the future knowledge.

The policies *maximum-utility* and *desired-time* both have another CPU constraint. GRACE-OS uses an earliest-deadline-first (EDF) scheduling algorithm (Section 5.2). The EDF schedulability requires that the total CPU utilization of all tasks is no more than one [65]. More formally, if there are n tasks, each task is configured at QoS level q_i and allocated $C_i(q_i)$ cycles per period $P_i(q_i)$, and the CPU runs at speed f, then the CPU constraint is

$$\sum_{i=1}^{n} \frac{\frac{C_i(q_i)}{f}}{P_i(q_i)} \leq 1 \tag{4.3}$$

where $\frac{C_i(q_i)}{f}$ is the processing time at speed f of the ith task.

To achieve a system-wide optimization, the coordinator needs to determine the configurations for all system layers. Specifically, assume that there are n concurrent tasks and each task supports multiple QoS levels $\{q_{i1}, ..., q_{im_i}\}, 1 \leq i \leq n$. The coordinator selects a QoS level q_{ij} for each task in the application layer, allocates $C_i(q_{ij})$ cycles per period $P_i(q_{ij})$ to each task in the operating system layer, and chooses a CPU speed f in the hardware layer. By selecting these variables, we can formulate the global adaptation as a constrained optimization problem.

Specifically, for the *maximum-utility* policy, which first maximizes the total utility and then minimizes power consumption, the global adaptation can be formulated as

$$\text{maximize} \quad \sum_{i=1}^{n} u_i(q_{ij}) \quad \text{(total utility)} \tag{4.4}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{\frac{C(q_{ij})}{f}}{P(q_{ij})} \leq 1 \quad \text{(CPU constraint)} \tag{4.5}$$

$$q_{ij} \in \{q_{i1}, ..., q_{im_i}\} \quad i = 1, ..., n \tag{4.6}$$

$$f \in \{f_1, ..., f_K\} \tag{4.7}$$

$$\text{minimize} \quad f \quad \text{(CPU speed/energy)} \tag{4.8}$$

For the *desired-time* policy, which maximizes the total utility under the CPU and energy constraints, the global adaptation can be formulated as

$$\text{maximize} \quad \sum_{i=1}^{n} u_i(q_{ij}) \quad \text{(total utility)} \tag{4.9}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{\frac{C(q_{ij})}{f}}{P(q_{ij})} \leq 1 \quad \text{(CPU constraint)} \tag{4.10}$$

$$q_{ij} \in \{q_{i1}, ..., q_{im_i}\} \quad i = 1, ..., n \tag{4.11}$$

$$f \in \{f_1, ..., f_K\} \tag{4.12}$$

$$p(f) \times T \leq E \quad \text{(energy constraint)} \tag{4.13}$$

where E and T denote the remaining battery energy and lifetime, respectively.

The above two constrained optimization problems are similar to the QoS management formulation in Q-RAM [89] and DQM [19], which also coordinate the resource allocation among multiple tasks to maximize the total utility of all tasks. GRACE-OS differs from these two approaches in that GRACE-OS considers energy in the optimization problems. This introduces two issues: First, we need to consider the energy constraint or minimize energy. Second, we need to consider the CPU constraint on a variable-speed processor.

4.2 Solution

In this section, we show that the global adaptation problems for *maximum-utility* and *desired-time* policies are NP hard, and then present a heuristic dynamic programming algorithm, which provides an approximate solution for these two problems.

4.2.1 NP Hardness

The above constrained optimization problems for the policies *maximum-utility* and *desired-time* are NP hard. To show this, it is sufficient to show that their common sub-problem

$$\text{maximize} \quad \sum_{i=1}^{n} u_i(q_{ij}) \qquad \text{(total utility)} \tag{4.14}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{\frac{C(q_{ij})}{f}}{P(q_{ij})} \leq 1 \qquad \text{(CPU constraint)} \tag{4.15}$$

$$q_{ij} \in \{q_{i1}, ..., q_{im_i}\} \quad i = 1, ..., n \tag{4.16}$$

$$f \in \{f_1, ..., f_K\} \tag{4.17}$$

is NP hard. To do this, we show that the multi-choice 0-1 Knapsack problem (MCKP) [84], which is known to be NP hard, is an instance of the constrained optimization problem in Equations (4.14)-(4.17). The MCKP can be described as follows.

Given a knapsack of capacity c and n groups $G_i, 1 \leq i \leq n$, of items, each item j with value v_j and weight s_j, select exactly one item j from each group, such that the total value of the selected items is maximized while their total weight is below the knapsack capacity. That is,

$$\text{maximize:} \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} x_{ij} v_{ij} \tag{4.18}$$

$$\text{subject to:} \quad \sum_{i=1}^{n} \sum_{j=1}^{|G_i|} x_{ij} s_{ij} \leq c \tag{4.19}$$

$$\sum_{j=1}^{|G_i|} x_{ij} = 1 \quad 1 \leq i \leq n \tag{4.20}$$

$$x_{ij} \in \{0, 1\} \quad 1 \leq i \leq n, j \in G_i \tag{4.21}$$

To show that the MCKP is an instance of the problem in Equations (4.14)-(4.17), we take two steps. First, we rewrite the problem in Equations (4.14)-(4.17) as the follows

$$\text{maximize:} \quad \sum_{i=1}^{n} \sum_{j=1}^{m_i} x_{ij} u_i(q_{ij}) \tag{4.22}$$

$$\text{subject to:} \quad \sum_{i=1}^{n} \sum_{j=1}^{m_i} x_{ij} \frac{C_i(q_{ij})}{P_i(q_{ij})} \leq f_K \tag{4.23}$$

$$\sum_{j=1}^{m_i} x_{ij} = 1 \qquad i = 1, ..., n \tag{4.24}$$

$$x_{ij} \in \{0, 1\} \qquad i = 1, ..., n, j = 1, ..., m_i \tag{4.25}$$

where f_K is the maximum CPU speed and $m_i, 1 \leq i \leq n$, is the number of QoS levels the ith task can operate.

In the second step, we set

$$G_i = \{q_{i1}, ..., q_{im_i}\} \quad i = 1, ..., n \tag{4.26}$$

$$v_{ij} = u_i(q_{ij}) \qquad i = 1, ..., n, j = 1, ..., m_i \tag{4.27}$$

$$s_{ij} = \tfrac{C_i(q_{ij})}{P_i(q_{ij})} \qquad i = 1, ..., n, j = 1, ..., m_i \tag{4.28}$$

$$c = f_K \tag{4.29}$$

That is, the MCKP is an instance of the rewritten problem in Equations (4.22) - (4.25). As a result, the rewritten problem is NP hard, since the MCKP is NP hard. Consequently, the global adaptation problems of the policies *maximum-utility* and *desired-time* are NP hard.

Note that $v_{ij} = u_i(q_{ij})$ and $s_{ij} = \frac{C_i(q_{ij})}{P_i(q_{ij})}$ for $1 \le i \le n$ and $1 \le j \le m_i$.

1. Initialization
   - 1.1 Set the capacity to $c = f_K$ for the *maximum-utility* policy, and $c = \max\{f : f \in \{f_1, ..., f_K\}$ and $p(f) \times T \le E\}$ for the *desired-time* policy.
   - 1.2 Sort the QoS levels $\{q_{i1}, ..., q_{im_i}\}$ for each task $i, 1 \le i \le n$, by the ascending order of utility.
   - 1.3 Select the first QoS level for each task (i.e., set $x_{i1} = 1, x_{ij} = 0$ for $2 \le j \le m_i$ and $1 \le i \le n$). Define the chosen aggregate CPU bandwidth demand (cycles per second) and aggregate utility as $B = \sum_{i=1}^{n} s_{i1}$ and $U = \sum_{i=1}^{n} v_{i1}$, respectively.
   - 1.4 For all QoS levels $j \ne 1$ define the slope $\lambda_{ij} = \frac{v_{ij} - v_{i,j-1}}{s_{ij} - s_{i,j-1}}, 1 \le i \le n$ and $2 \le j \le m_i$. This slope measures the ratio of utility to bandwidth demand by selecting QoS level j, rather than QoS level $j-1$, for the ith task.
   - 1.5 Sort the slopes $\{\lambda_{ij}\}$ in non-descending order.

2. Check each QoS level in the order of $\{\lambda_{ij}\}$
   - 2.1 If $B + s_{ij} - s_{i,j-1}$ is greater than the capacity c goto Step 3.
   - 2.2 Otherwise, we upgrade the QoS level of the ith task to the next higher one. That is, we set $x_{ij} = 1, x_{i,j-1} = 0$ and update $B = B + s_{ij} - s_{i,j-1}$ and $U = U + v_{ij} - v_{i,j-1}$. Repeat Step 2.

3. Set global adaptation results
   - 3.1 Set the ith task's QoS level to q_{ij} where $x_{ij} = 1$.
   - 3.2 Set the CPU speed to $\min\{f : f \in \{f_1, ..., f_K\}$ and $f \ge B\}$. Note that if $B = c$, we get an optimal solution.

Figure 4.1: Dynamic programming algorithm to solve the global adaptation problems for the *maximum-utility* and *desired-time* policies.

4.2.2 Heuristic Algorithm

If the number of tasks, n, and the number of QoS levels, m_i, of each task are small, we can use exhaustive search to find the optimal solution for these two NP hard problems. In general, however, these two numbers may be large; for example, a video encoder may have up to hundreds of QoS levels by changing the parameters for quantization and motion search. As a result, exhaustive search, whose computational complexity is exponential, is not feasible to solve the global adaptation problems.

We therefore develop a dynamic program algorithm, based on the Knapsack algorithm pro-

posed by Pisinger [84], to provide a heuristic solution for the global adaptation problems. Specifically, we sort all QoS levels of all tasks in a non-decreasing order of a *slope*, which is defined as the utility-to-demand ratio by increasing a task's QoS level to the next higher level. We initially set all tasks to the lowest QoS level and increase each task's QoS level by visiting the sorted slope list until we meet the CPU constraint in Equation (4.23). Figure 4.1 illustrates the algorithm. The complexity of our developed algorithm is $O(M \log M)$ due to the sorting of the slopes (Step 1.5), where M is the sum of the number of QoS levels of all tasks, i.e., $M = \sum_{i=1}^{n} m_i$.

4.3 Global Adaptation Protocol

The global adaptation happens when the CPU resource needs to be reallocated among tasks, for example, when a task joins or leaves the system. The process of the global adaptation involves all three adaptive layers (the CPU hardware, operating system, and multimedia tasks). We next describe the inputs and outputs of the global adaptation and the communication among different layers during the global adaptation.

To perform the global coordination, the coordinator takes the following inputs:

- The residual battery energy, which is observed by the battery monitor.
- The CPU speed options and the speed-power relationship of the device, which are typically available through measurement.
- The QoS levels each task can operate as well as the utility and CPU demand of each QoS level. These are typically either specified by the application developers or the user or obtained through profiling.

After solving the global adaptation problem, the coordinator generates the following outputs:

- The operating QoS level for each task in the application layer.
- The CPU allocation for each task in the operating system layer.

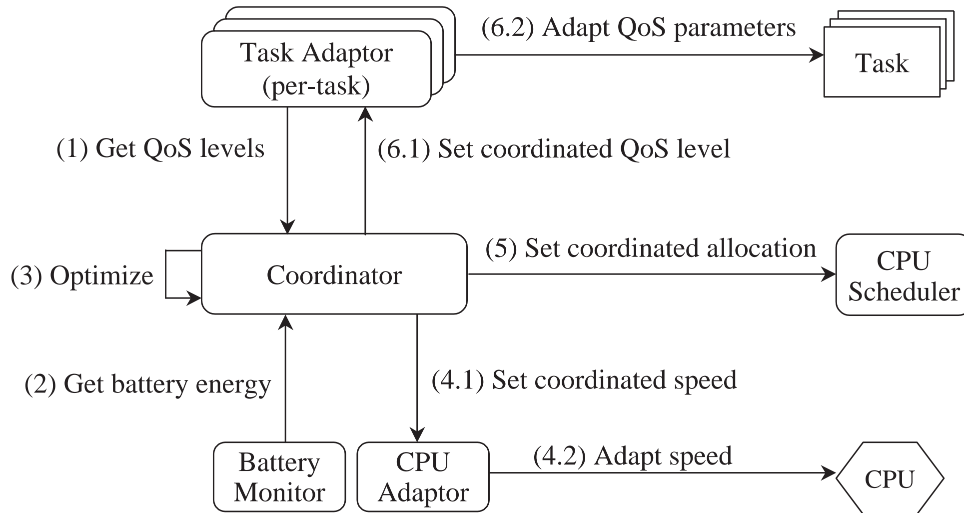


Figure 4.2: The global adaptation protocol: The coordinator in the operating system makes the global decisions based on the system states collected from multiple layers.

- The CPU speed and expected device power in the hardware layer.

Based on these outputs, individual layers can further optimize the corresponding configuration locally, as long as they do not violate the global decisions.

To support the global adaptation, GRACE-OS provides mechanisms (e.g., system calls) and protocol for a global communication among the hardware, operating system, and application layers. In Section 6, we will describe the communication mechanisms in detail. Here, we outline the coordination protocol, as shown in Figure 4.2.

(1) Each task adaptor tells the coordinator the task's QoS levels by specifying the utility and CPU demand of each QoS level. The utility can be specified by the user; the CPU demand can profiled with the support of the operating system (see Section 4.4).

(2) The battery monitor tells the coordinator the residual battery energy. For recent smart batteries, the software, e.g., the operating system, can monitor the battery energy availability through the Advanced Configuration and Power Interface (ACPI) standard [26].

(3) The coordinator solves the constrained optimization problem for the *maximum-utility* or *desired-time* policy, depending the user's preference, and consequently determines a global

configuration for the task QoS level, CPU allocation, and CPU speed.

(4) The coordinator tells the coordinated CPU speed to the CPU adaptor (4.1), which, in turn, correspondingly adjusts the CPU speed (4.2).

(5) The coordinator tells the coordinated CPU allocation to the CPU scheduler, which enforces the allocation to deliver multimedia tasks performance guarantees.

(6) The coordinator tells the coordinated QoS level to each task adaptor (6.1), which, in turn, correspondingly adjusts the QoS parameters, such as frame rate, of the task (6.2).

4.4 Long-Term Demand Prediction

As discussed in Section 4.3, the coordinator takes the following inputs: (1) residual battery energy, (2) CPU speed levels and device power consumption, and (3) QoS levels of tasks, and (4) the long-term CPU demand of tasks. The first three inputs can be monitored or specified by the device manufacturer or application developer. The last input, i.e., long-term CPU demand, is dynamic and typically depends on the processed multimedia stream. Therefore, global adaptation requires predicting the long-term CPU demand for each individual task.

For each QoS level q, the CPU demand of a task has two parameters, demanded cycles $C(q)$ and period $P(q)$. The period can be directly calculated from the application-level QoS parameters; for example, if the frame rate is r, then the period is $\frac{1}{r}$. The challenging problem is how to determine the parameter $C(q)$ for each QoS level of each task. While the problem of CPU allocation for QoS provisioning has been studied extensively in the literature [19, 25, 54, 71, 79, 89, 90], the problem of *how much* CPU to allocate to each task has received relatively little attention.

In this section, we propose an approach that automatically predicts the long-term CPU demand of each task for global adaptation. This prediction involves three steps: First, we use a kernel-based profiling technique to monitor the CPU cycle usage of individual tasks. Second, we estimate the probability distribution of cycle demand of each task based on the monitored cycle usage. Finally,

For each QoS level q of the task

1. Profile the cycle usage of each job when the task operates at QoS level q.
2. Estimate the probability distribution of cycle demand of the task when it operates at QoS level q.
3. Determine the parameter $C(q)$ of the task based on the demand distribution and statistical performance requirement of the task.

Figure 4.3: Outline of prediction of long-term CPU demand for each QoS level of a task.

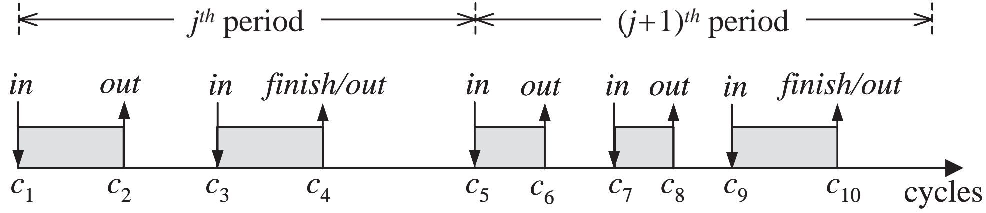
we determine how many cycles to allocate to each task based on its cycle demand distribution. Figure 4.3 outlines the process of the prediction of long-term cycle demand. We next describe these steps in detail.

4.4.1 Kernel-Based Profiling of Cycle Usage

Recently, a number of demand profiling mechanisms, ranging from kernel based approach to instrument based approach, have been proposed. GRACE-OS takes the kernel-based approach for two reasons. First, the kernel-based approach can work with any existing application and requires no changes to the source or binary code. This is especially important to release the burden of the application developers. Second, accurate profiling of a task's CPU usage requires detailed information on when and how many cycles the application uses at a fine time granularity. This information can be easily accessed in the kernel.

GRACE-OS profiles the cycle usage, rather than the time usage, of multimedia tasks for the following reason. On a dynamic-speed CPU, the processing time demanded by a job execution (e.g., frame decoding) is dependent on the underlying CPU speed, while the number of cycles demanded by a job is roughly constant with different speeds. In particular, the cycle usage of a multimedia job typically includes two parts, memory access and computation. The number of cycles spent on memory access is often negligible, and other computation cycles are roughly independent of the speed [49].

To profile the cycle usage for each QoS level, we add a *cycle counter* into the process control block of each task. This cycle counter monitors the number of cycles the task consumes when the



in	the profiled task is switched in for execution
out	the profiled task is switched out for suspension
finish	the profiled task finishes a job

cycles for the jth job $= (c_2 - c_1) + (c_4 - c_3)$

cycles for the $(j+1)^{th}$ job $= (c_6 - c_5) + (c_8 - c_7) + (c_{10} - c_9)$

Figure 4.4: Kernel-based cycle profiling: monitoring the number of cycles elapsed between each task's switch-in and switch-out during context switches.

task is executed. In particular, this counter measures the number of cycles elapsed between the task's switch-in and switch-out during context switches. The sum of these elapsed cycles during a job execution gives the number of cycles the job uses. Figure 4.4 illustrates this kernel-based online profiling technique.

Note that multimedia tasks tell the kernel about their jobs via system calls; e.g., when an MPEG decoder finishes a frame decoding, it may call `sleep` to wait for the next frame. As a result, the kernel can know the start and end of a job for the cycle profiling. Further, when used with resource containers [13], our proposed profiling technique can be more accurate by subtracting cycles consumed by the kernel (e.g., for interrupt handling). We currently do not do this since the number of cycles consumed by the kernel is typically negligible relative to the number of cycles consumed by a multimedia job, which is often in millions.

Our proposed profiling technique is distinguished from others [7, 99, 111] for three reasons. First, it profiles during runtime, without requiring an isolated profiling environment (e.g., as in [99]). This is especially important since multimedia tasks often run concurrently with other applications. Second, it is customized for counting the number of cycles consumed by each job. It is therefore simpler, thus with lower overhead, than general profiling systems that assign counts to different

program functions [7, 111]. Finally, it incurs small overhead, which happens *only* when updating cycle counters before a context switch. There is no additional overhead, e.g., due to the sampling interrupts [7, 111].

4.4.2 Estimation of Demand Distribution

After profiling the cycle usage for each QoS level, we next need to decide how many cycles to allocate to each task. Multimedia tasks are soft real-time tasks and hence do not require worst-case deadline guarantees. Therefore, GRACE-OS does not need to allocate the worst-case number of cycles to each task. Instead, GRACE-OS allocates cycles statistically, thereby improving the CPU and energy utilization. To perform the statistical allocation, we first need to estimate the probability distribution of the cycle demand for each QoS level.

To do this, we employ a simple yet effective *histogram* technique. Specifically, we use a profiling window to keep track of the number of cycles consumed by n jobs of the task for a specific QoS level. The parameter n can either be specified by the application or be set to a default value (e.g., the last 100 jobs). Let C_{min} and C_{max} be the minimum and maximum number of cycles, respectively, in the window. We obtain a histogram from the cycle usage as follows:

1. We use $C_{min} = b_0 < b_1 < \cdots < b_r = C_{max}$ to split the range $[C_{min}, C_{max}]$ into r groups. We refer to $\{b_0, b_1, ..., b_r\}$ as the group boundaries.

2. Let n_i be the number of cycle usage that falls into the ith group $(b_{i-1}, b_i]$. The ratio $\frac{n_i}{n}$ represents the probability that the task's cycle demand is in between b_{i-1} and b_i, and $\sum_{j=0}^{i} \frac{n_j}{n}$ represents the probability that the task needs no more than b_i cycles.

3. For each group, we plot a rectangle in the interval $(b_{i-1}, b_i]$ with height $\sum_{j=0}^{i} \frac{n_j}{n}$. All rectangles together form a histogram, as shown in Figure 4.5.

From a probabilistic point of view, the above histogram of a task approximates the cumulative

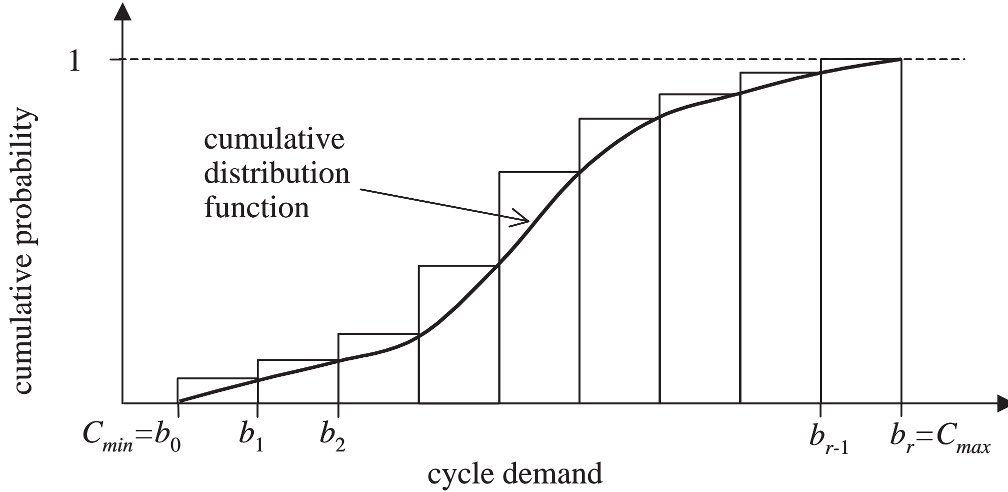


Figure 4.5: Histogram-based estimation: the histogram approximates the cumulative distribution function of a task's cycle demand for a QoS level.

distribution function of the task's cycle demands, i.e.,

$$F(x) = \mathcal{P}[X \leq x] \tag{4.30}$$

where X is the random variable of the task's demands. In particular, the rectangle height, $\sum_{j=0}^{i} \frac{n_j}{n}$, of a group $(b_{i-1}, b_i]$ approximates the cumulative distribution at b_i, i.e., the probability that the task demands no more than b_i cycles for each job execution. In this way, we can estimate the cumulative distribution for the group boundaries of the histogram, i.e., $F(x)$ for $x \in \{b_0, b_1, ..., b_r\}$.

The above kernel-based profiling tries to support general multimedia applications and requires knowledge as little as possible from the applications. With the help of the applications (e.g., the frame type of MPEG videos), the profiling can predict the cycle demand more precisely. Furthermore, unlike distribution parameters such as the mean and standard deviation, the above histogram describes the property of the full demand distribution. This property is necessary for internal adaptation (see Section 5.4). On the other hand, we use the above histogram technique, rather than distribution functions such as normal and gamma (e.g., in [67]), because the former is simple. First, the histogram describes a task's demand distribution without needing to configure function parameters, such as the mean and standard deviation of a normal distribution. Second, it is easy

to update the histogram during the task execution. For example, an MPEG video decoder may change its demand distribution due to video scene changes. In such a case, either the decoder or the scheduler can initiate to update the demand distribution, since the decoder knows the semantic changes in its input stream, while the scheduler can check if the decoder's recent CPU usage matches the previously estimated distribution.

4.4.3 Determining Long-Term Demand

After determining the demand distribution for each QoS level of each task, we next discuss how to determine the long-term cycle demand— the parameter $C(q)$— for each QoS level, i.e., determining how many cycles to periodically allocate to each task for the corresponding QoS level. This is a challenging problem— over-allocation may waste cycles and energy, while under-allocation may degrade application performance. For example, the worst-case-based allocation results in low CPU utilization since a task seldom needs the worst case, while the average-based allocation may miss many deadlines since a task often needs more than the average.

GRACE-OS instead takes a *stochastic* approach that decides the long-term demand based on the statistical performance requirement and demand distribution of each task. Compared to the worst-case-based allocation, the stochastic allocation can improve the utilization of the CPU and energy resources. On the other hand, compared to the average-based allocation, the stochastic allocation can deliver statistical performance guarantees to individual tasks.

Specifically, let ρ be the statistical performance requirement of a task— the task needs to meet ρ percent of deadlines. In other words, each job of the task should meet its deadline with a probability ρ. To support this requirement, the scheduler allocates the task C cycles per period, so that the probability that a job of the task requires no more than the allocated C cycles is at least ρ; i.e.,

$$\mathcal{P}[X \leq C] \geq \rho \tag{4.31}$$

To find such a parameter C for a task, we can use its demand histogram. In particular, we

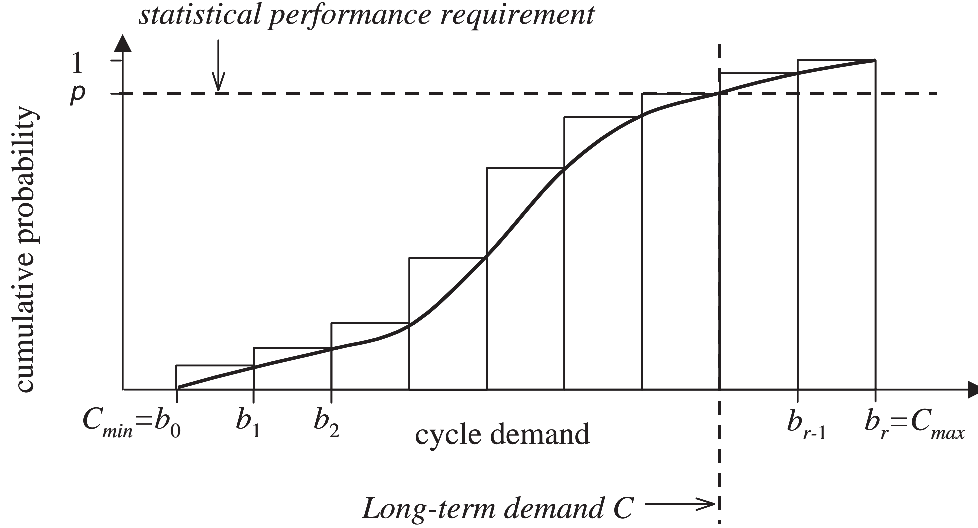


Figure 4.6: Determine the statistical, long-term cycle demand for each QoS level of a task based on its demand distribution.

search the histogram group boundaries, $\{b_0, b_1, ..., b_r\}$, to find the smallest b_m whose cumulative distribution is at least ρ, i.e., $F(b_m) = \mathcal{P}[X \leq b_m] \geq \rho$. We then use this b_m as the parameter C. Figure 4.6 illustrates the stochastic allocation process.

4.5 Summary

Our work in this chapter has made two contributions. First, we formulated the global adaptation problem which coordinates the CPU allocation among concurrent tasks. Although the problem of coordinated CPU allocation among multiple adaptive tasks has been addressed before. Our contribution is to apply the coordinated allocation on a variable-speed CPU and with energy consideration. In particular, the global adaptation coordinates all three layers (the CPU hardware, operating system, and multimedia tasks) for a user-specified system-wide optimization.

The second contribution is an automatic prediction technique to determine the long-term cycle demand for each QoS level of each task. This prediction technique involves three steps: kernel-based profiling, histogram-based estimation, and statistical allocation. While there are several profiling and estimation approaches to determine application's resource demand, our approach is

simple and customized for frame-based multimedia tasks. Furthermore, the estimated probability distribution of cycle demand can be used in internal adaptation (which will be discussed in the Chapter 5) in addition to global adaptation.

Chapter 5

Internal Adaptation

5.1 Overview

In Chapter 4, we discussed how GRACE-OS performs global adaptation in response to large system changes such as task entry and exit. The coordinated decisions made in the global adaptation are the QoS level for each task in the application layer, the CPU allocation for each task in the operating system layer, and the CPU speed and power consumption in the hardware layer. After the global adaptation, GRACE-OS needs to enforce the global contract on multimedia utility and energy. In particular, in the time interval between global adaptations, each task should operate at the coordinated QoS level, and the device should consume no more than the coordinated power.

To enforce these two global decisions simultaneously, GRACE-OS integrates soft real-time scheduling and dynamic voltage scaling together, thereby saving energy while provisioning QoS. Specifically, it extends traditional real-time scheduling by adding another scheduling dimension—*speed*. That is, the scheduler decides *what CPU speed* to execute tasks in addition to what tasks to execute and when to execute them. The purpose of the speed-aware real-time scheduling algorithm is to support multimedia QoS while saving energy.

The scheduler allocates CPU cycles periodically according to the coordinated CPU allocation, which, in turn, is based on the long-term prediction of task's cycle demand. Specifically, at the global adaptation time, we predict each task's demand as a statistical number of cycles (e.g., the 95th percentile of previous jobs) per job. At runtime, multimedia tasks often vary their CPU

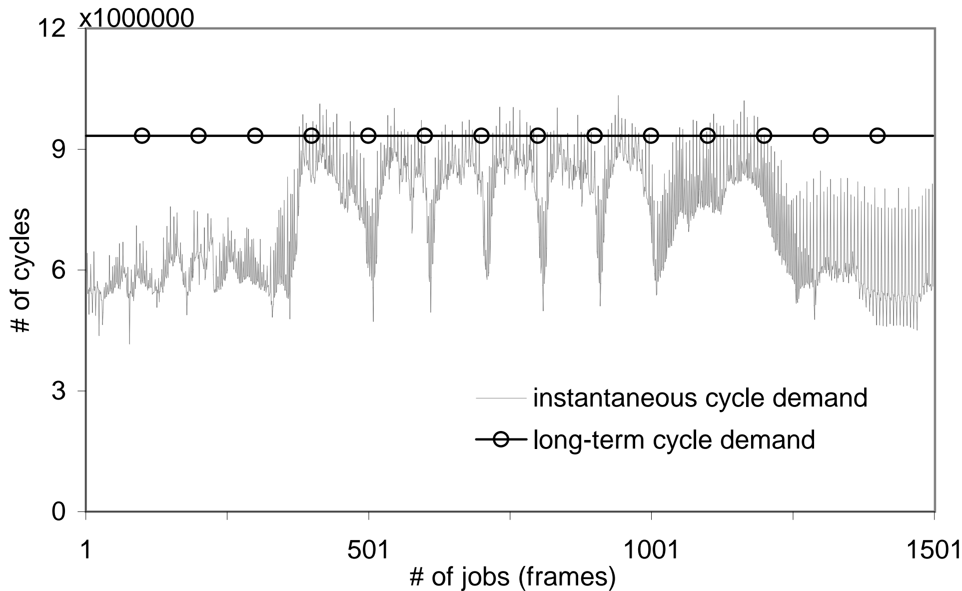


Figure 5.1: Variations of instantaneous cycle demand of an MPEG video decoder at fine time granularity: Frame decoding may need more or less cycles than the long-term prediction (95th percentile of all frames).

demand dynamically. An MPEG decoder, for example, needs different amount of cycles to decode I, P, and B frames. Figure 5.1 shows the long-term and instantaneous demand of an MPEG player when decoding the `4dice.mpg` video with frame size 352×240 pixels.

The dynamic nature of the instantaneous demand of multimedia tasks means that they often overrun or underrun the coordinated allocation (i.e., need more or less cycles than the allocated). An overrun may cause the task or other tasks to miss a deadline, while an underrun may result in CPU slack time, thus wasting energy. As a result, we need to handle overrun and underrun at fine time granularity. To do this, the GRACE system performs internal adaptation in each layer. The goal of internal adaptation is to minimize energy while providing the multimedia utility expected by the coordinator.

In general, internal adaptation can happen in each system layer. For example, in the application layer, multimedia tasks can adapt QoS parameters, such as rate, within an acceptable range of the coordinated QoS level through rate control or media scaling [34, 61, 78, 91]. The application internal adaptation is often application-specific. Consequently, we do not discuss it in detail in

this thesis since we focus on the operating system to support general multimedia applications. Instead, we concentrate on internal adaptation in the operating system and hardware layers, which is independent of applications.

Internal adaptation in the operating system layer dynamically adjusts the policy of CPU allocation and scheduling to maintain multimedia QoS in the presence of the variations in CPU usage. The BERT operating system [14], for example, allows multimedia applications to borrow CPU time from best-effort applications when the former needs more time. Similarly, internal CPU adaptation dynamically adjusts the CPU speed and power without affecting application performance. GRACE-OS integrates internal adaptation in the hardware and operating system layers together and we simply refer to the integrated internal adaptation in these two layers as internal adaptation in the rest of the thesis until specified otherwise.

There are two reasons for this integration: First, the decisions on internal CPU adaptation are often made by the operating system, typically the scheduler, since it has the knowledge on application demands and hence can save energy while meeting application demands. Second, by integrating adaptation in the CPU and operating system, GRACE-OS seeks to enable multimedia tasks to provide the utility expected by the coordinator while minimizing the energy consumption[1]. We minimize energy here since energy is a conservable resource (i.e., can be saved for the future). As a result, if less energy is used than that expected by the coordinator, energy can be saved for later jobs which could demand more CPU and energy or for admitting more tasks later.

GRACE-OS supports two kinds of internal adaptation:

- *Reactive adaptation*. GRACE-OS first runs the CPU at the coordinated speed and adjusts the speed based on the prediction of the instantaneous demand of a job when the job overruns or underruns. Upon an overrun, the scheduler speeds up the CPU to allocate more cycles to the job and hence avoids the job to miss deadline. Upon an underrun, the scheduler slows down the CPU to reclaim the residual cycles, thus saving energy.

[1] In internal adaptation, it is possible to provide higher utility than the expected by the coordinator. GRACE-OS does not do this since utility changes often result in rapid fluctuation of the perceived multimedia quality, which is annoying to the end user.

- *Proactive adaptation*. GRACE-OS changes the CPU speed at finer time granularity within each job execution based on the prediction of the statistical cycle demand of each multimedia task. The proactive adaptation is motivated by an observation on the CPU usage pattern of multimedia tasks: Although multimedia tasks change their instantaneous cycle demand largely, the probability distribution of their cycle demand is often stable. By setting speed based on the demand distribution, GRACE-OS can minimize the energy consumption while providing multimedia tasks statistical performance guarantees.

In the next sections, we introduce the extended soft real-time scheduling algorithm, which enables each task to operate at the coordinated QoS level and provide the utility expected by the coordinator. We then describe an example of the scheduling algorithm; this example motivates the need for internal adaptation to handle the variations of CPU usage at fine time granularity. Finally, we discuss the reactive and proactive internal adaptation methods in detail.

5.2 Soft Real-Time Scheduling

Multimedia tasks present demanding computational requirements that must be met in soft real time, e.g., decoding a frame within a period. Soft real-time scheduling is a common mechanism to support such timing requirements, typically by combining predictable allocation (such as proportional sharing [23, 29, 40, 76] and reservation [25, 54, 88]) and real-time scheduling algorithm (such as earliest deadline first and rate monotonic [65, 66]).

Previous soft real-time scheduling algorithms, however, often assume that the CPU runs at a constant speed. This assumption does not hold for our target mobile devices with a variable-speed CPU. As a result, we cannot directly use existing scheduling algorithms in GRACE-OS. We therefore extend traditional real-time scheduling algorithms by adding another dimension— *speed*. That is, the scheduler also sets the CPU speed when executing a task and hence enforces the CPU allocation on a variable-speed CPU [106]. We next describe this scheduling algorithm in detail, followed by an scheduling example.

5.2.1 The Scheduling Algorithm

GRACE-OS uses an energy-aware EDF scheduling algorithm, which enforces the globally coordinated CPU allocation on a variable-speed CPU [106]. To provide soft EDF schedulability[2], the scheduler needs to make admission control to ensure that the total CPU utilization of all concurrent tasks is no more than one, i.e.,

$$\sum_{i=1}^{n} \frac{\frac{C_i(q_i)}{f}}{P_i(q_i)} \leq 1 \tag{5.1}$$

where there are n tasks, each allocated $C_i(q_i)$ cycles per period $P_i(q_i)$, and the coordinated CPU speed is f. Note that this admission test is made at the global adaptation. The test condition is CPU constraint in the global adaptation (Equation (4.3)).

In this scheduling algorithm, each task has a deadline and a cycle budget:

- The deadline of the task equals to the end of its current period. That is, when a task begins a new period, its deadline is postponed by the period.
- The budget of a task is recharged periodically. In particular, when a task begins a new period, its budget is recharged to the coordinated number of cycles.

The scheduler schedules all tasks based on their deadline and budget. In particular, the scheduler always dispatches the task that has the earliest deadline and a positive budget. As the task is executed, its budget is decreased by the number of cycles it consumes. That is, if the task executes for Δt time units at speed f, its budget is decreased by $\Delta t \times f$. When the budget of a task is decreased to 0, the task is preempted to run in best-effort mode until its budget is replenished again at the next period. This preemption provides temporal and hence performance isolation among tasks; i.e., a task's performance is not affected by the behavior of other tasks [54, 76, 110].

This algorithm also handles the overrun part in a bounded time. Specifically, assume that a multimedia task T_i, allocated C_i cycles per period P_i, is preempted at time t for overrun protection

[2] By soft schedulability, we mean that GRACE-OS provides soft deadline guarantees. This is different from schedulability in hard real-time systems [65], which guarantees that all jobs meet their deadline.

1. Assume each task T_i is allocated $C_i(q_i)$ cycles every period $P_i(q_i)$ in the global coordination.

2. When a task T_i begins a new period at time t,
   The task's deadline is updated as $t + P_i(q_i)$
   The task's budget is recharged as $C_i(q_i)$

3. Dispatch the task T_i with the earliest deadline and positive budget.

4. Decrease the task T_i's budget by the number of cycles it consumes.
   If the task T_i's budget becomes 0, preempt it into best-effort mode and goto step 3.

Figure 5.2: The scheduling algorithm.

and the overrun part demands c^o cycles. To complete the overrun part, the task needs to consume its budget by an amount of c^o cycles. This consumption takes at most $\left\lceil \frac{c^o}{C_i} \right\rceil$ periods since the task is allocated C_i cycles per period P_i. Therefore, the task will finish its overrun part no later than

$$t + \left\lceil \frac{c^o}{C_i} \right\rceil \times P_i \tag{5.2}$$

Figure 5.2 summarizes this scheduling algorithm. Note that in this algorithm, we do not specify how to set the CPU speed. The advantage of the scheduling algorithm is that it can enforces the allocation regardless of the CPU speed changes, which happen in the global adaptation. This scheduling algorithm also provides flexibility for changing the CPU speed in internal adaptation.

5.2.2 An Scheduling Example

We next give an example to illustrate the above scheduling algorithm. This example includes two multimedia tasks, T_1 and T_2, which join the system at time 0 and 20, respectively. When a task joins the system, GRACE-OS invokes a global adaptation to allocate CPU to all concurrent tasks. Assume these two tasks are allocated 7.5×10^6 cycles per period 30 milliseconds and 5×10^6 cycles per period 20 milliseconds, respectively. The scheduler executes tasks using the EDF-based scheduling algorithm at the coordinated speed.

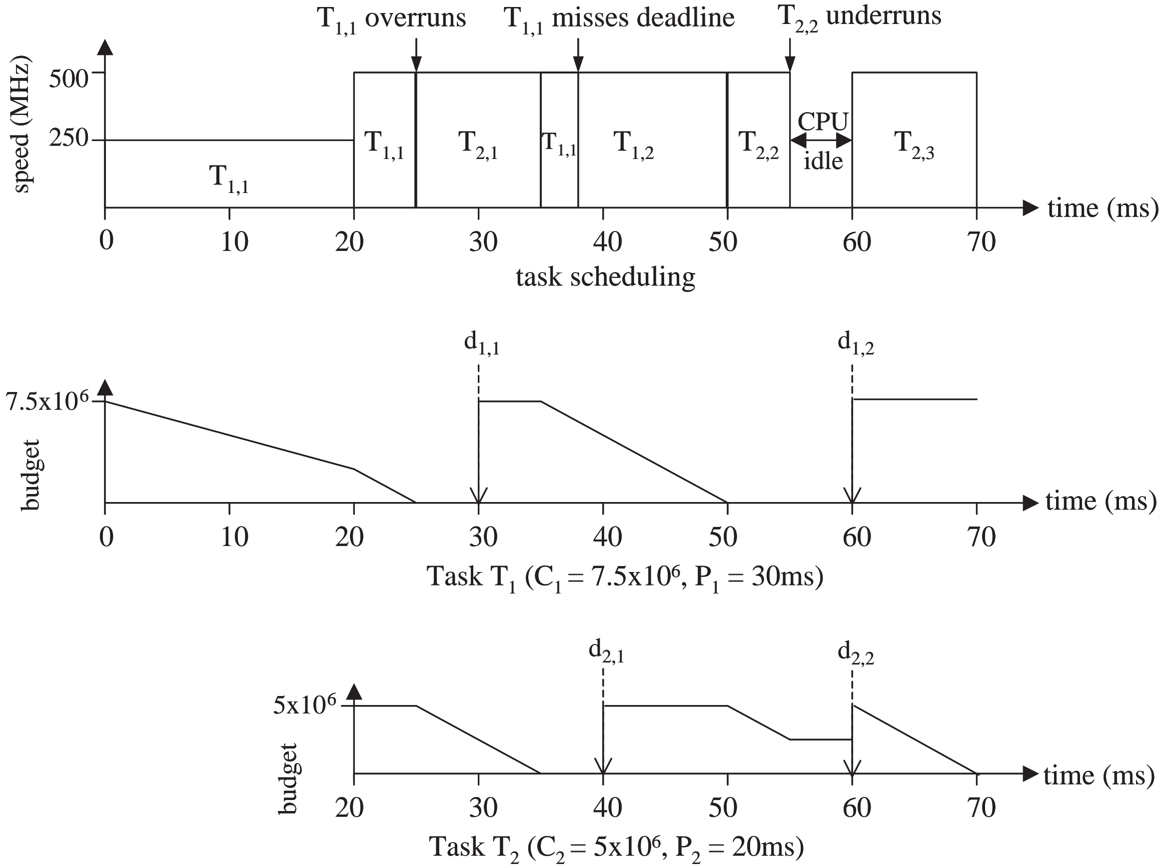


Figure 5.3: An example of the speed-aware, EDF-based scheduling algorithm.

In particular, initially at time 0, there is only one task T_1. T_1 is executed at the speed 250 MHz. When task T_2 joins at time 20, T_1 still has the earliest deadline and hence continues to execute until its budget is exhausted at time 25. At this time, T_1 overruns (i.e., its current job $T_{1,1}$ does not finish but its budget is exhausted). The scheduler hence preempts T_1 and dispatches T_2. At time 30, T_1 begins a new period and the scheduler updates its deadline and recharges its budget. Since T_2 still has the earliest deadline, it continues to execute until time 35. As a result, task T_1 misses the deadline for its job $T_{1,1}$. Similarly, at time 55, task T_2 underruns (i.e., it completes its job earlier with a residual budget). As a result of this early completion, the CPU is idle in the interval from 55 to 60, thus wasting energy.

The above example illustrates that the EDF-based scheduling algorithm enforces the coordi-

nated CPU allocation at the coordinated CPU speed and protects temporal isolation among tasks. This algorithm, however, cannot efficiently handle overruns and underruns, which result from variations in runtime CPU usage of tasks. In particular, when a task overruns, it is preempted to run in best-effort mode. This may result in a deadline miss, which typically degrades the application performance. On the other hand, an underrun may cause the CPU to be idle and hence waste energy. We next discuss how GRACE-OS uses reactive and proactive internal adaptations to handle overruns and underruns, thus avoiding (or reducing) the deadline misses and energy waste.

5.3 Reactive Internal Adaptation

In the reactive internal adaptation method, the scheduler monitors the cycle usage of jobs of each task and adapts the CPU allocation and speed in response to the variations of the cycle usage. In particular, GRACE-OS considers two kinds of reactive internal adaptations with different time granularity:

- **Per-job adaptation** at per-job time granularity. When the scheduler detects a job overrun or underrun, it allocates an extra budget to or reclaims the residual budget from the task. This adaptation applies to *only the current overrun or underrun job of the task.*

- **Multi-job adaptation** at multi-job time granularity in case of consistent overruns or underruns, e.g., due to video scene changes. The scheduler adjusts the task's cycle allocation based on its recent CPU usage (e.g., using feedback control [61, 97, 21]). This adaptation applies to *all later jobs of the task* until other adaptation happens.

5.3.1 Per-job Adaptation

While scheduling tasks, the scheduler also monitors the cycle usage for each job execution and adjusts the cycle budget when a job needs more or less cycles than the allocated. Specifically, consider that a task T_i underruns at time t with a residual budget of b_i cycles. Let t' be the beginning

of T_i's next period, at which T_i releases a new job and gets its budget recharged. The task's residual budget is wasted since the task has no job to execute until time t'. To avoid this waste, the scheduler reclaims the residual budget from the task by slowing down the CPU at the time interval $[t, t')$.

In particular, at the current speed, f_{cur}, the processor provides a total CPU budget (for all concurrent tasks) of

$$f_{cur} \times (t' - t) \tag{5.3}$$

cycles in the interval $[t, t')$. However, the actual total budget demand is

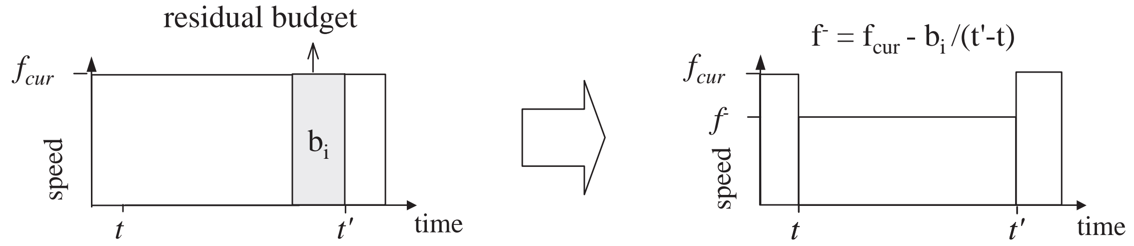
$$f_{cur} \times (t' - t) - b_i \tag{5.4}$$

cycles because of T_i's underrun. Hence, during the interval $[t, t')$, the processor can slow down to a lower speed:

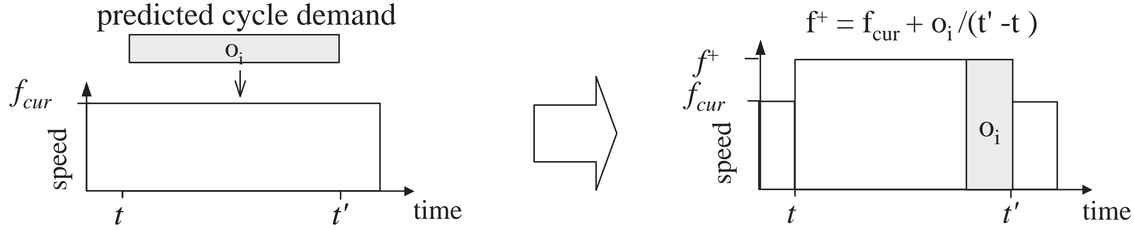
$$f^{-} = f_{cur} - \frac{b_i}{t' - t} \tag{5.5}$$

Figure 5.4-(a) illustrates the process for underrun handling.

On the other hand, consider that a task T_i overruns at time t, and t' is T_i's deadline. The basic idea behind overrun handling is to allocate T_i with an extra budget, so that T_i can finish the overrun job by its deadline. The scheduler, however, does not know the actual number of cycles for the overrun part, which is available only after the job completes. Hence, we use some heuristics, such as AVG(k)— the average of the recent k jobs' and LAST OVERRUN— the number of cycles of the last overrun job, to predict the overrun CPU demand [102, 49]. Since the task has executed a part of the job, the task can provide some useful information (e.g., MPEG frame type) to help the prediction. For a predicted overrun of o_i cycles, the total budget demand (across all tasks) over the



(a) Reclaim residual budget and slow down the CPU to handle underrun



(b) Allocate extra budget and speed up the CPU to handle overrun

Figure 5.4: Per-job adaptation to handle underrun and overrun.

interval $[t, t')$ is

$$f_{cur} \times (t' - t) + o_i \tag{5.6}$$

cycles. Thus, in this interval, the processor needs to run at a higher speed:

$$f^+ = f_{cur} + \frac{o_i}{t' - t} \tag{5.7}$$

Figure 5.4-(b) illustrates the process for overrun handling.

The idea of underrun handing is similar to previous reclamation approaches, which also first run the CPU fast and slow down upon early completion [9, 83, 104]. The idea of accelerating the CPU to handle overrun is novel. The per-job adaptation illustrates the flexibility of our speed-aware real-time scheduling algorithm. This algorithm extends traditional real-time scheduling algorithms by adding CPU speed as another scheduling dimension. That is, the scheduler can change the speed to handle underrun and overrun.

5.3.2 Multi-job Adaptation

As discussed in Chapter 4, the coordinator allocates CPU cycles to each task based on its stochastic cycle demand. Multimedia tasks may change the stochastic demand over time due to variations in the input data. Figure 5.5, for instance, plots the variations of the instantaneous and stochastic cycle demands of an MPEG decoder, which plays video `4dice.mpg` with frame size 352×240 pixels. The decoder's stochastic cycle demand, defined as the 95th percentile of the job cycles, changes for different video segments; e.g., the 95th percentile of all jobs is much higher than that of the first and last 300 jobs, but is lower than that of the middle 300 jobs.

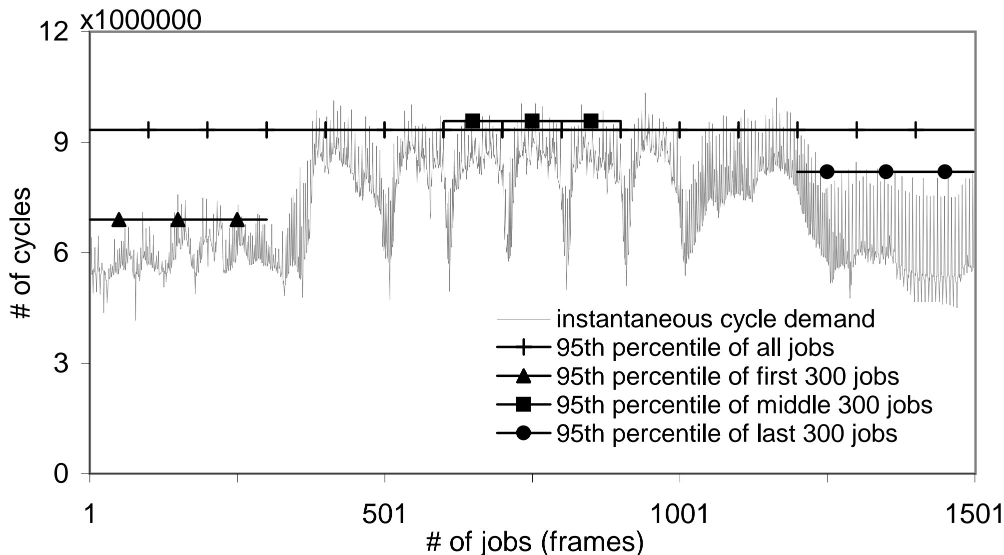


Figure 5.5: Variations of instantaneous and statistical demand of an MPEG video decoder.

The dynamic nature of the stochastic demand implies that the decoder may consistently underrun or overrun its coordinated allocation. The consistent underruns or overruns would trigger the above per-job adaptation frequently. Such frequent adaptation is inefficient since it may changes the CPU speed frequently and hence incur large overhead (as shown in Section 7.2). To avoid this, GRACE-OS triggers *multi-job* adaptation in case of consistent underruns or overruns. The multi-job adaptation updates the statistical demand of the task (and hence the number of cycles allocated to all later jobs of the task) according to its recent CPU usage.

Specifically, for each task, the scheduler uses a profiling window to keeps track of the number

of cycles the task has consumed for its recent W jobs, where W is the profiling window size (W is set to 100 in our implementation). When the overrun or underrun ratio of a task exceeds a threshold, the scheduler calculates a new statistical cycle demand, e.g., as the 95th percentile of the job cycles in the profiling window. Let C' be the new statistical cycle demand. The scheduler then uses an exponential average strategy, commonly used in control systems [61, 96], to update the task's statistical demand C as

$$\alpha \times C + (1 - \alpha) \times C' \tag{5.8}$$

where $\alpha \in [0, 1]$ is a tunable parameter and represents the relative weight between the old and new cycle demands (α is set to 0.2 in our implementation).

Note that here GRACE-OS adjusts only the statistical cycle demand for a task, but does not change its period. The reason is that period adjustment often changes the task's QoS level, which in turn may cause fluctuation in the perceptual quality. In GRACE-OS, the goal of the internal adaptation is to enable multimedia tasks to maintain their QoS and provide the utility expected by the coordinator.

When the multi-job adaptation updates a task's statistical cycle demand, the total CPU demand of all concurrent tasks changes accordingly. If the total demand exceeds the maximum CPU speed, the multi-job adaptation fails. After reaching a certain failure threshold, the scheduler can either tell the task to degrade its quality and CPU demand or trigger a global adaptation to reallocate the CPU among all tasks. GRACE-OS takes the latter approach since it can potentially achieve a better configuration. For example, if an important task, such as a user-focused video, consistently overruns and the CPU already runs at the maximum speed, GRACE-OS can allocate more cycles to this important task by decreasing the allocation to other less important tasks.

In summary, to handle variations in task CPU usage, GRACE-OS integrates three different adaptations: per-job adaptation, multi-job adaptation and global adaptation, and applies them at different time scales. Figure 5.6 illustrates this integration.

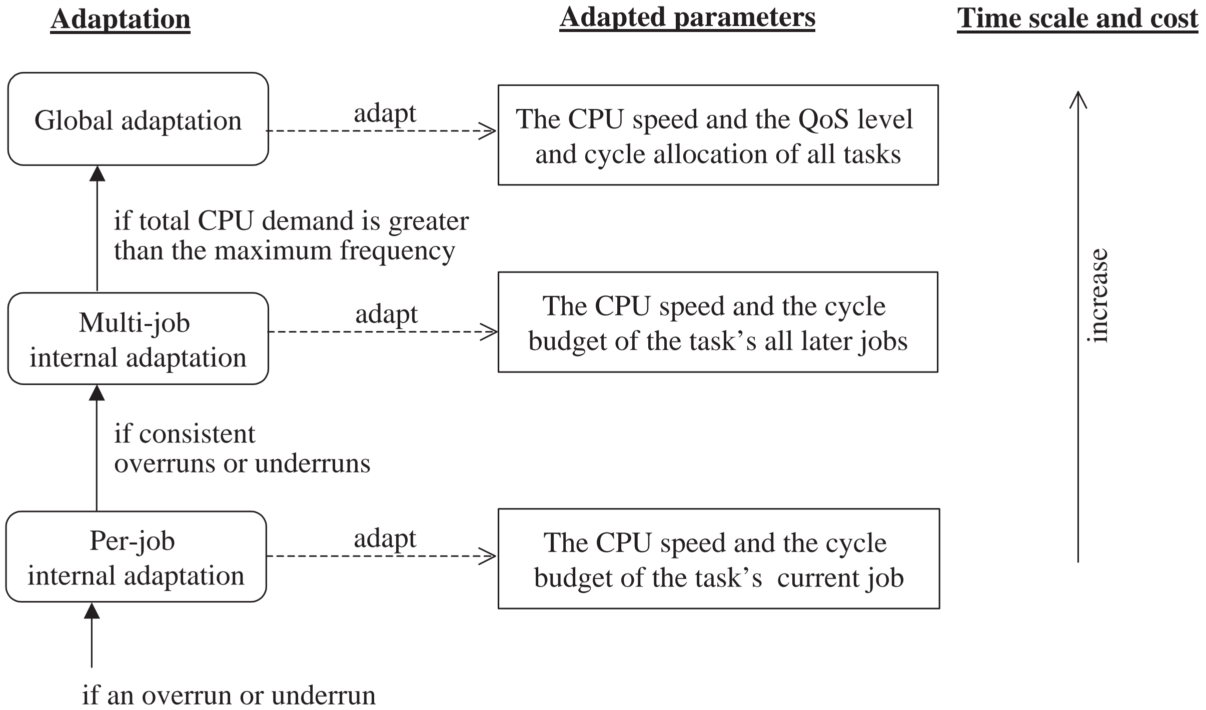


Figure 5.6: Applying reactive internal adaptation and global adaptation at different time scales to handle CPU usage variations.

5.4 Proactive Internal Adaptation

In the previous section, we discussed two reactive internal adaptation methods, per-job adaptation and multi-job adaptation, to handle overruns and underruns. The basic idea of reactive adaptation is to (1) set a constant CPU speed for a job execution (e.g., decoding a frame) based on the prediction of the instantaneous cycle demand of the job and (2) adjust the speed in response to an overrun or underrun. As we mentioned before, the instantaneous cycle demand of multimedia tasks often change largely due to the variations in the input data (e.g., I, P and B frames). It is therefore difficult to precisely predict the instantaneous cycle demand.

Proactive internal adaptation is an alternative approach to handling overruns and underruns. It does not detect and react to overruns and underruns, but exploits the statistical cycle demand of multimedia tasks. Specifically, it adapts the CPU speed *within* each job execution based on the probability distribution of cycle demand. This speed adaptation typically starts a job slowly

and accelerates the job as it progresses. The goal is to minimize the energy consumption while delivering statistical deadline guarantees to multimedia tasks.

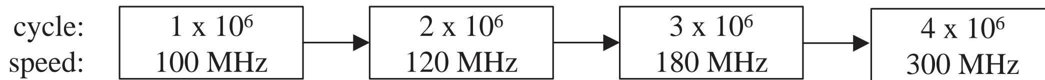
Compared to reactive adaptation, proactive adaptation is more aggressive by assuming that a job will use fewer cycles than allocated. As a result, if a job finishes early, it does not need to execute the high speed (and high power) part. This is especially useful to save more energy. Recall that the coordinated CPU allocation is based on the long-term, statistical cycle demand. For example, if the allocation is based on the 95th percentile of cycle demand of all jobs, then about 95% of jobs underrun the allocation. This means that in the average case, the proactive adaptation can save more energy.

In this section, we discuss how to perform proactive adaptation during each job execution. Specifically, we use a *speed schedule* for each task. The speed schedule of a task defines when to change the speed and what speed to change to when executing each job of the task. We then discuss how to calculate the speed schedule based on the probability distribution of cycle demand of each individual task. For the calculation, we first consider a simple case by assuming an ideal processor, which can change speed in a continuous manner and whose power consumption is proportional to the cube of the speed; we then consider non-ideal processors that support a discrete set of speeds, rather than a continuous range, and their power consumption does not scale in a cubic manner. Finally, we investigate the stability of the demand distribution of common multimedia tasks. The stability justifies the feasibility of the proactive adaptation.

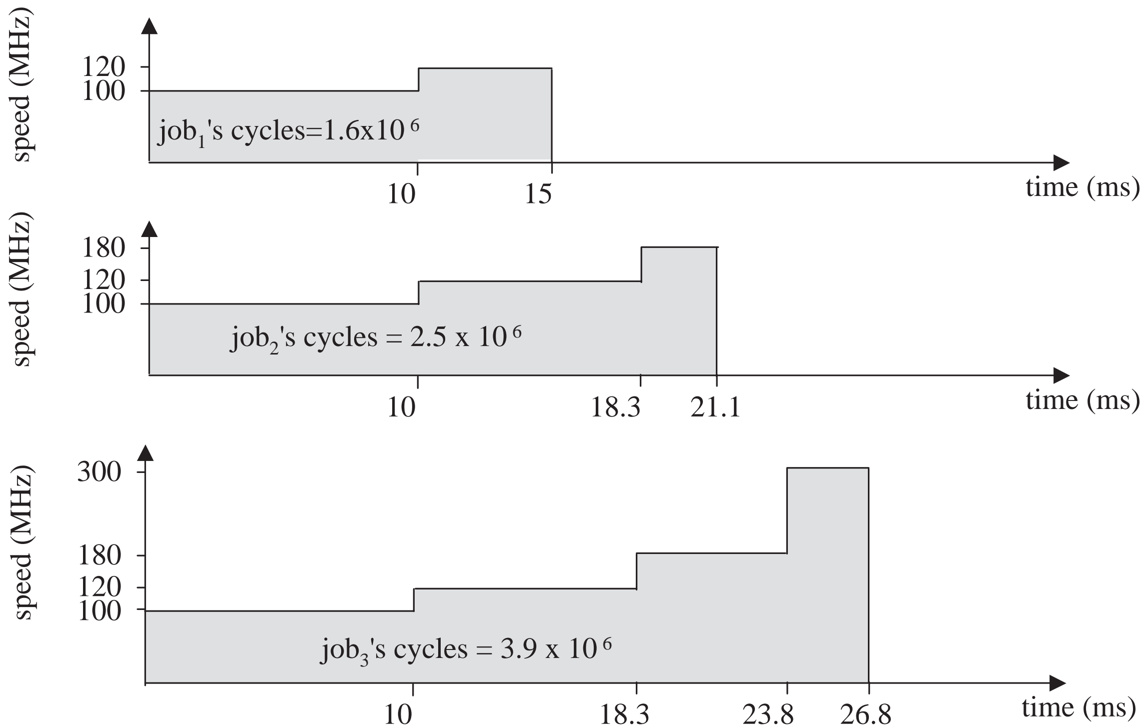
5.4.1 Adaptation with Speed Schedule

To perform the proactive internal adaptation, we define a *speed schedule* for each multimedia task. The speed schedule is a list of scaling points, where the execution speed of the task changes. Each point (x, y) specifies that a job of the task is executed at the speed y until the job uses x cycles.

To enforce the speed schedule, we extend the process control block (PCB) of multimedia tasks by adding two attributes, the speed schedule and current speed. Whenever a task is dispatched,



(a) Speed schedule with four scaling points



(b) Speed scaling for three jobs using speed schedule in (a)

Figure 5.7: Example of speed schedule and corresponding speed scaling for job execution: the scheduler dynamically changes the speed during a job execution.

the scheduler sets the CPU speed to the current speed of the task. The current speed of a task is maintained as follows:

- When the task begins a new job, its current speed is set to the speed of the first point of its speed schedule.
- As the task executes its job, if its cycle usage is equal to the cycle of the next point of its speed schedule, then its current speed advances to the speed of its next scaling point.

Figure 5.7-(a) shows an example of a task's speed schedule with four scaling points. Figure 5.7-(b) shows the corresponding speed scaling for three jobs of the task. Each job starts at speed 100 MHz and accelerates as it progresses. If a job needs fewer cycles, it avoids the high speed

execution. For example, the first job requires 1.6×10^6 cycles and thus needs to execute at speed 100 and 120 MHz only.

Note that the adaptation with the speed schedule is integrated with real-time scheduling. Specifically, the speed schedule of different tasks may be different. the scheduler changes the speed in three cases:

- **Context switch**. The scheduler dispatches another task to execute, it always sets the CPU speed based on the speed schedule of the current task. This provides isolation of speed adaptation between different tasks.

- **New job release**. When the current task releases a new job, its current speed is reset to the speed of the first point in its speed schedule.

- **Job progress**. The scheduler also monitors the progress of each job execution and changes the CPU speed when the job reaches the next scaling point.

This is significantly different from the reactive adaptation method whose speed adaptation affects all concurrent tasks. Figure 5.8 shows an example of the scheduling and adaptation process with two concurrent tasks.

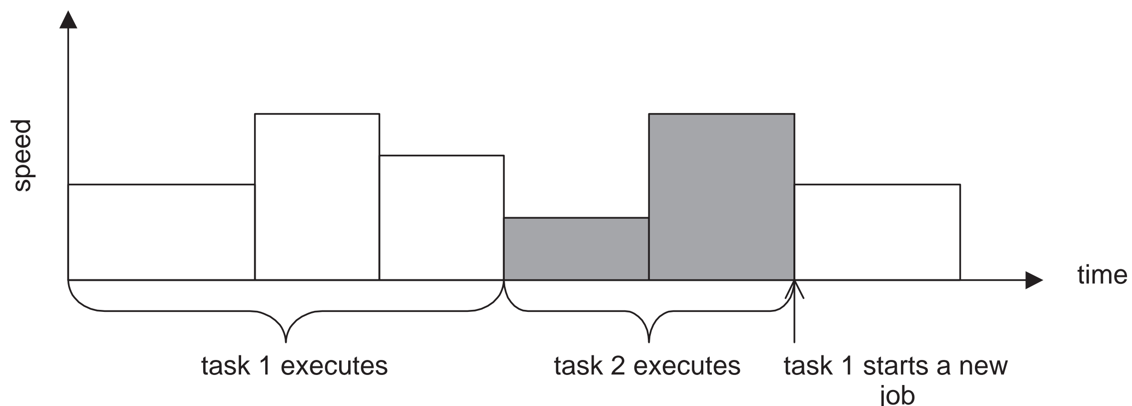


Figure 5.8: Scheduling of two tasks: The CPU speed changes during the task execution and in a context switch.

5.4.2 Speed Schedule for Ideal Processors

Now, we discuss how to construct the speed schedule for each task based on its demand distribution, similar to the stochastic DVS techniques proposed by Lorch and Smith [67] and Gruian [42]. The goal is to minimize the total energy consumed during the job execution while bounding the job's execution time. The reason for bounding the execution time is not to miss the deadline of the job or other jobs executed after the job. In other words, each job should finish within a certain amount of time.

We therefore allocate a time budget for each job. Specifically, if there are n concurrent tasks and each task is allocated C_i cycles per period P_i, then the ith task is allocated

$$T_i = \frac{C_i}{\sum_{i=1}^{n} \frac{C_i}{P_i}} \tag{5.9}$$

time units per period P_i (i.e. for each of its jobs). That is, we distribute the time among all tasks based on their cycle demands. Intuitively, if there is only a single task, then its time budget equals to its period; if multiple tasks run concurrently, they need to share time with each other and hence get a shorter time budget. By using cycle and time allocation together, we can adapt the execution speed for each job as long as the job can use its allocated cycles within its allocated time.

With the allocation of cycles and time, we formulate the problem of constructing speed schedule as a constrained optimization problem. We then use the Lagrange method to solve the problem.

Problem Formulation

The speed schedule construction problem thus becomes, for each task, to find a speed for each of its allocated cycles, such that the total energy consumption of these allocated cycles is minimized while their total execution time is no more than the allocated time. To perform the construction, here we assume an ideal CPU that can change the speed in a continuous way and whose power consumption is proportional to the cube of the speed. Specifically, if a cycle x executes at speed

f_x, its execution time is

$$\frac{1}{f_x} \tag{5.10}$$

and its energy consumption is proportional to

$$\frac{1}{f_x} \times f_x^3 = f_x^2 \tag{5.11}$$

Since a task requires cycles statistically, it uses each of its allocated cycles with a certain probability. Therefore, each allocated cycle x is executed with a certain probability; consequently, its average energy consumption is proportional to

$$(1 - F(x))f_x^2 \tag{5.12}$$

where $F(x)$ is the cumulative distribution function defined in Equation (4.30). In this way, constructing the speed schedule for a task is equivalent to:

$$\text{minimize:} \quad \sum_{x=1}^{C}(1 - F(x))f_x^2 \tag{5.13}$$

$$\text{subject to:} \quad \sum_{x=1}^{C}\frac{1}{f_x} \leq T \tag{5.14}$$

where C and T are the task's allocated cycles and allocated time per period, respectively.

To solve the above constrained optimization, we need to know the cumulative distribution $F(x)$ for each allocated cycle. However, our histogram-based estimation provides the cumulative distribution for only the group boundaries of the histogram; i.e., we know $F(x)$ for only $x \in \{b_0, b_1, ..., b_m\}$, where $b_m = C$ is the cycle group boundary that is equal to the number of allocated cycles (i.e., the ρth percentile of the task's cycle demands fall into the first m groups of its histogram).

We therefore use a piece-wise approximation technique that divides the allocated cycles into groups and finds a speed for each cycle group, rather than for each individual cycle. In particular, we find the speed for the group boundaries and use a uniform speed within each group. That is, we rewrite the above constrained optimization as:

$$\text{minimize:} \quad \sum_{i=0}^{m} s_i \times (1 - F(b_i)) f_{b_i}^2 \tag{5.15}$$

$$\text{subject to:} \quad \sum_{i=0}^{m} s_i \times \frac{1}{f_{b_i}} \leq T \tag{5.16}$$

where s_i is the size of the ith group, i.e.,

$$s_i = \begin{cases} b_0 & : \quad i = 0 \\ b_i - b_{i-1} & : \quad 0 < i \leq m \end{cases} \tag{5.17}$$

Solution

We next use the Lagrange method to solve the constrained optimization problem in Equations (5.15)-(5.16). To do this, we define the Lagrange multiplier

$$L(\lambda) = \sum_{i=0}^{m} s_i (1 - F(b_i)) f_{b_i}^2 + \lambda \left(\sum_{i=0}^{m} s_i \frac{1}{f_{b_i}} - T \right) \tag{5.18}$$

To minimize Equation (5.15), we need

$$\frac{\partial L}{\partial f_{b_i}} = 2 s_i (1 - F(b_i)) f_{b_i} - \lambda s_i f_{b_i}^{-2} = 0 \quad i = 0, ..., m \tag{5.19}$$

$$\lambda \geq 0 \tag{5.20}$$

$$\lambda \left(\sum_{i=0}^{m} s_i \frac{1}{f_{b_i}} - T \right) = 0 \tag{5.21}$$

$$\sum_{i=0}^{m} s_i \frac{1}{f_{b_i}} \leq T \tag{5.22}$$

We have two possible cases for the Equations (5.19)-(5.22):

(1) $\lambda = 0$

This means that

$$\frac{\partial L}{\partial f_{b_i}} = 2s_i(1 - F(b_i))f_{b_i} = 0 \quad i = 0, ..., m \tag{5.23}$$

Since

$$s_i > 0 \tag{5.24}$$

$$1 - F(b_i) > 0 \tag{5.25}$$

we get

$$f_{b_i} = 0 \quad i = 0, ..., m \tag{5.26}$$

This is contract with

$$\sum_{i=0}^{m} s_i \frac{1}{f_{b_i}} \leq T \tag{5.27}$$

Therefore, this case does not hold.

(2) $\lambda > 0$

This means that

$$\frac{\partial L}{\partial f_{b_i}} = 2s_i(1 - F(b_i))f_{b_i} - \lambda s_i f_{b_i}^{-2} = 0 \quad i = 0, ..., m \tag{5.28}$$

$$\sum_{i=0}^{m} s_i \frac{1}{f_{b_i}} - T = 0 \tag{5.29}$$

That is

$$f_{b_i} = \sqrt[3]{\frac{\lambda}{2(1 - F(b_i))}} \tag{5.30}$$

$$\sum_{i=0}^{m} s_i \frac{1}{f_{b_i}} - T = 0 \tag{5.31}$$

$$\sqrt[3]{\lambda} = \frac{\sum_{i=0}^{m} s_i \sqrt[3]{2(1 - F(b_i))}}{T} \tag{5.32}$$

so

$$\begin{aligned} f_{b_i} &= \frac{\sum_{i=0}^{m} s_i \sqrt[3]{2(1 - F(b_i))}}{T \sqrt[3]{2(1 - F(b_i))}} \\ &= \frac{\sum_{i=0}^{m} s_i \sqrt[3]{1 - F(b_i)}}{T \sqrt[3]{1 - F(b_i)}} \end{aligned} \tag{5.33}$$

Equation (5.33) gives the speed for each of the cycle group boundaries, i.e., f_{b_i} for each group boundary $b_i, i = 0, ..., m$. This solution has two properties:

- As the time allocation T decreases, the speed $f_{b_i}, i = 0, ..., m$, increases. This is easy to understand: When a job is allocated with less time, it needs to run fast to catch the deadline.
- As the cycles b_i increases, the speed $f_{b_i}, i = 0, ..., m$, increases. This means that each job is executed slowly first and is accelerated as the job progresses. This is similar to the finding in PACE [67].

Based on the solution in Equation (5.33), we can construct the speed schedule of a task by adding a scaling point for each group boundary. That is, the speed schedule consists of $m + 1$ scaling points. Each point has cycle number b_i and speed $f_{b_i}, 0 \leq i \leq m$. According to the constructed speed schedule of a task, the scheduler executes each job of the task in the following manner. The first group of cycles $[0, b_0)$ of the job are executed at the speed f_{b_0}, the next group of

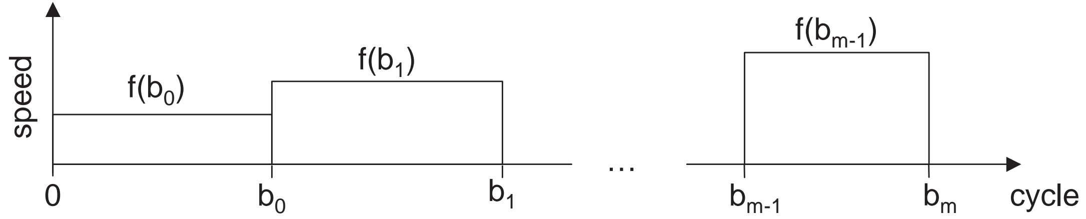


Figure 5.9: Adaptation based on the speed schedule: Each job starts slowly and accelerate as it progresses.

cycles $[b_0, b_1)$ are executed the speed f_{b_1}, and so on. Figure 5.9 illustrates this execution process.

Next, we give a simple example to illustrate why the proactive internal adaptation based on the demand distribution can save energy. Assume that (1) an MPEG decoder is allocated 2×10^6 cycles and 10 ms per period, (2) the coordinated CPU speed is 200 MHz, and (3) a cycle consumes $f^2 \times 10^{-12}$ joule energy at the speed f MHz. Further, assume that 80% of frames demand 10^6 cycles to decode and 20% of frames demand 2×10^6 cycles. In another word, the allocated 2×10^6 cycles are executed in the following manner: The first 10^6 cycles are executed with probability 1 and the second 10^6 cycles are executed with probability 0.2.

We can decode each frame according two speed schedule (Figure 5.10):

- Execute the whole frame at the coordinated speed 200 MHz. The total execution time is 10 ms and within the allocated time budget. The expected energy consumption for a frame is $10^6 \times 200^2 \times 10^{-12} + 0.2 \times 10^6 \times 200^2 \times 10^{-12} = 4.8 \times 10^{-2}$ joule.

- Adapt the speed of a frame based on the demand distribution. Specifically, we execute the first 10^6 cycles at the speed 158 MHz and the second 10^6 cycles at the speed 272 MHz. The total execution time is still 10 ms. The expected energy consumption for a frame is $10^6 \times 158^2 \times 10^{-12} + 0.2 \times 10^6 \times 272^2 \times 10^{-12} = 3.98 \times 10^{-2}$ joule.

The second approach saves energy by 17% compared to the first approach. This clearly illustrates that the proactive internal adaptation based on the demand distribution can save more energy while not affecting application performance.

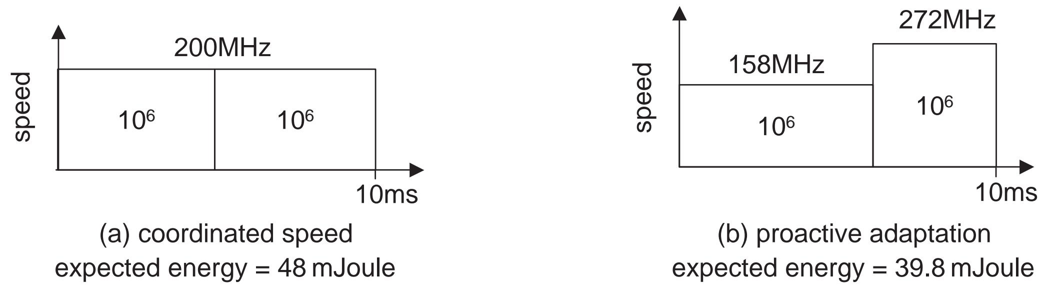


Figure 5.10: Comparsion of expected energy of (a) executing each frame at the coordinated speed and (b) adapting the speed based on the demand distribution.

5.4.3 Speed Schedule for Non-Ideal Processors

Our previous speed schedule calculation (and generally most of previous DVS algorithms [68, 83, 107]) assumes an ideal CPU: (1) the CPU can change speed continuously, (2) the CPU power is dominated by the dynamic power, which is proportional to the speed and square of the voltage, and (3) the voltage is proportional to the speed. That is, a lower speed yields a cubic power reduction and a quadratic energy reduction.

In practice, however, mobile devices often have a *non-ideal* processor. First, mobile processors support a discrete set of speeds, rather than a continuous range. For example, the StrongARM SA-1110 CPU supports 11 different speeds, from 59 MHZ to 206 MHz in steps of 14.7 MHz. Second, a lower speed does not yield a cubic power reduction, since the static power also has a significant effect and the voltage does not scale linearly to the speed. For example, our measurements on an HP laptop with an Athlon CPU [5] show that a lower speed saves much less power than the ideal cubic power-speed relationship (Figure 5.11).

In general, there are three approaches to handle the discrete set of speed options of non-ideal processors:

(1) Calculate the speed schedule by assuming an ideal processor and then round the calculated speed to the upper bound of the supported speeds [83, 107].

(2) Emulate the calculated speed with two bounding supported speeds [42, 52, 67]. This approach distributes cycles that need to be executed at the calculated speed into two parts, one

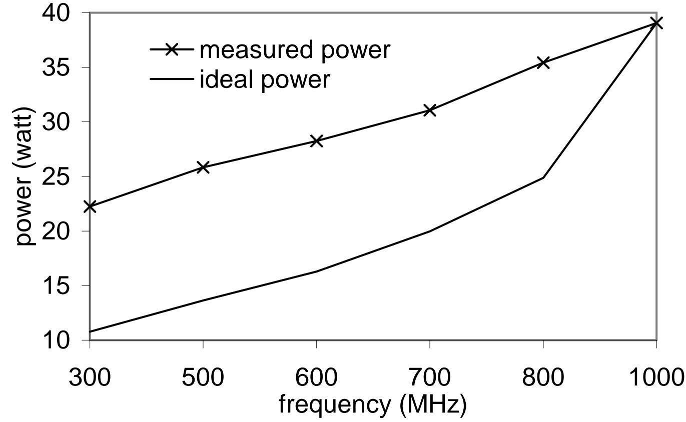


Figure 5.11: Measured and ideal power on an HP N5470 laptop with an Athlon CPU: The measured power is obtained with an oscilloscope, while the ideal power is calculated by assuming that the power is proportional to the cube of the speed.

for the lower bound and the other for the upper bound. Specifically, assume that x cycles need to be executed at the calculated speed f and the lower and upper speed bounds are f_l and f_h, respectively. This emulation executes x_1 cycles at speed f_l and x_2 cycles at speed f_h, such that

$$x_1 + x_2 = x \tag{5.34}$$

$$\frac{x}{f} = \frac{x_1}{f_l} + \frac{x_2}{f_h} \tag{5.35}$$

This emulation approach has been shown to be effective in simulations. It, however, may potentially result in large overhead when used in real implementations since it changes the speed more frequently.

(3) Calculate the speed schedule by explicitly considering the discrete speed levels of the CPU and the total device power (rather than the CPU power only) at different speeds [108]. By doing so, this approach minimizes the total energy consumed by the device, rather than the CPU energy only, while delivering soft deadline guarantees to multimedia tasks.

Next, we discuss the third approach in detail. We formulate the problem with these considerations and then describe the solution for this new problem.

Problem Formulation

If a cycles $x, 1 \leq x \leq C$, is executed at speed $f(x)$, its execution time is $\frac{1}{f(x)}$. The energy consumed by the device during this time interval is

$$\frac{1}{f(x)} \times p(f(x)) = \frac{p(f(x))}{f(x)} \tag{5.36}$$

where $p(f(x))$ is the power consumed by the whole device at speed $f(x)$. Since the cycle x is executed with a probability and its *expected energy* is

$$(1 - F(x)) \times \frac{p(f(x))}{f(x)} \tag{5.37}$$

where $F(x)$ is the cumulative distribution function.

Recall that the cycle usage of a job is divided into m groups $[b_i, b_{i+1}), 0 \leq i \leq m-1$. We set a speed for each cycle group in a way that minimizes the total energy consumed during the job execution while bounding the job's execution time. More formally, we formulate the speed adaptation problem as

$$\text{minimize} \quad \underbrace{\sum_{i=0}^{m} (1 - F(b_i)) s_i \frac{p(f_{b_i})}{f_{b_i}}}_{\text{busy energy}} + \underbrace{\left(T - \sum_{i=0}^{m} (1 - F(b_i)) s_i \frac{1}{f_{b_i}} \right) p_{idle}}_{\text{idle energy}} \tag{5.38}$$

$$\text{subject to} \quad \sum_{i=0}^{m} s_i \frac{1}{f_{b_i}} \leq T \tag{5.39}$$

$$f_{b_i} \in \{f_1, ..., f_K\}, \quad 0 \leq i \leq m \tag{5.40}$$

where T is the time budget allocated to the job and p_{idle} is the device power when the CPU is idle at the lowest speed.

In Equation (5.38), the first part is the energy consumed when executing all allocated cycles; the second part is the energy consumed during the residual time, which equals to the time budget minus the expected execution time of all allocated cycles. During this residual time, the CPU is

often idle since the task needs to wait until next job is available[3]. This idle slack is often very short; so we cannot put the CPU into the lower power *sleep* state due to overhead (which is, e.g., 160 ms for StrongARM SA-1100 [15]). We therefore set the CPU to the lowest speed during the idle slack. Note that Equation (5.39) bounds the worst-case, rather than the expected, execution time of all allocated cycles.

The above constrained optimization is similar to the energy optimization in previous statistical DVS algorithms [42, 67, 107] in that all of them find a speed for each of the allocated cycles to minimize their total energy. However, our approach differs substantially from previous statistical DVS algorithms (and generally most of previous DVS algorithms) for two reasons: First, our proposed approach explicitly considers the discrete set of speeds. Second, our proposed approach considers the energy consumed when the CPU is idle and also minimizes the total energy consumed by the whole device, rather than CPU energy only.

Solution

The optimization problem in Equations (5.38)-(5.40) is NP hard since one can easily prove that the multi-choice 0-1 Knapsack problem [84], which is known to be NP hard, is an instance of the optimization problem in Equations (5.38)-(5.40). Specifically, we can rewrite this optimization problem as

$$\text{minimize:} \quad \sum_{i=0}^{m}\sum_{j=1}^{K} x_{ij} \times (1 - F(b_i)) s_i \frac{p(f_j) - p_{idle}}{f_j} + T \times p_{idle} \tag{5.41}$$

$$\text{subject to:} \quad \sum_{i=0}^{m}\sum_{j=1}^{K} x_{ij} \times \frac{s_i}{f_j} \leq T \tag{5.42}$$

$$\sum_{j=1}^{K} x_{ij} = 1, \quad i = 0, ..., m \tag{5.43}$$

$$x_{ij} \in \{0, 1\}, \quad i = 0, ..., m, j = 1, ..., K \tag{5.44}$$

[3] Although the EDF algorithm allows other tasks to share the residual time, the CPU may be idle eventually. As a part of future work, we are investigating how GRACE-OS can utilize the residual time; e.g., we can allocate the residual time, ΔT, to the next task and hence relax its time constraint to $\Delta T + T$ in Equation (5.39).

Note that $v_{ij} = (1 - F(b_i))s_i \frac{p(f_j) - p_{idle}}{f_j}$ and $w_{ij} = \frac{s_i}{f_j}$ for $0 \le i \le m$ and $1 \le j \le K$.

1. Initialization
   - 1.1 Set the lowest speed for each cycle group (i.e., set $x_{i1} = 1, x_{ij} = 0$ for $2 \le j \le K$ and $0 \le i \le m$).
   - 1.2 Define the total execution time as $B = \sum_{i=1}^{n} w_{i1}$.
   - 1.3 Define the slope $\lambda_{ij} = \frac{v_{ij} - v_{i,j-1}}{w_{ij} - s_{i,j-1}}$, $0 \le i \le m$ and $2 \le j \le K$. This slope measures the ratio of energy to time by changing the speed of a cycle group from f_{j-1} to f_j.
   - 1.4 Sort the slopes $\{\lambda_{ij}\}$ in non-descending order.

2. Check the speed in the order of $\{\lambda_{ij}\}$
   - 2.1 If $B - w_{ij} + w_{i,j-1}$ is less than the time budget T, then goto Step 3.
   - 2.2 Otherwise, set $x_{ij} = 1, x_{i,j-1} = 0$ and update $B = B - w_{ij} + w_{i,j-1}$. Repeat Step 2.

3. Set the speed for each cycle group
   - 3.1 Set the speed of the ith cycle group to f_j where $x_{ij} = 1$. Note that if $B = T$, we get an optimal solution.

Figure 5.12: Dynamic programming algorithm to calculate the speed schedule for non-ideal processors with a discrete set of speed options.

where K is the number of supported speeds, $\{f_1, ..., f_K\}$. One can easily reduce the multi-choice Knapsack problem into an instance of the above optimization problem.

Being NP hard, the optimization problem does not have an optimal yet feasible solution. GRACE-OS provides a heuristic solution with a dynamic programming algorithm, based on the algorithm proposed by Pisinger [84]. Specifically, we sort the combinations of all speed options for all cycle groups in the non-decreasing order of a slope, which is defined as the energy-to-time ratio by increasing a group's speed to the next higher speed. We initially set all groups to the lowest speed and increase each group's speed by visiting the sorted slope list until we meet the time constraint in Equation (5.39). Figure 5.12 shows the dynamic programming algorithm. The complexity of this algorithm is $O(mK\log(mK))$, where m is the number of cycle groups and K is the number of speeds. The output of this algorithm is a speed f_{b_i} for each cycle group $i, 0 \le i < m$. That is, we get a speed schedule $(b_0, f_{b_0}), (b_1, f_{b_1}), \cdots, (b_{m-1}, f_{b_{m-1}})$.

5.4.4 Stability of Demand Distribution

The proactive internal adaptation algorithms depend on the probability distribution of cycle demand of multimedia tasks. If a task's demand distribution is stable, the scheduler can estimate it with a small profiling window; otherwise, the scheduler can either estimate the demand distribution with a large profiling window or update it when it changes. Now, we empirically analyze the stability of demand distribution.

To do this, we use the prediction method discussed in Section 4.4 to profile the cycle usage of typical multimedia codecs of speech, audio, and video during various time intervals of their execution (e.g., during the first 50 and 100 jobs) and estimate the demand distribution from the cycle usage. We then compare the demand distributions of different time intervals. Although we report the results for specific input streams, we have also experimented other input streams for each application and found similar results.

Figure 5.13-(a) depicts the cycle usage of the `MPGDec` application, an MPEG video decoder, for the whole video clip *lovebook.mpg* with frame size 320×240 pixels and 7691 frames. Figure 5.13-(b) plots its demand distribution for decoding different parts of the video (e.g., the first 50 and 100 frames). The figure shows two important characteristics of the `MPGDec`'s CPU usage.

- First, its instantaneous cycle demand is bursty and most jobs do not need the worst case cycles; e.g., for the first 100 jobs, the worst-case demand is 9.9×10^6 cycles, but 99% of jobs require less than 9.4×10^6 cycles. This indicates that compared to worst-case-based allocation and speed scaling, stochastic allocation and scaling can improve CPU and energy utilization. For example, the scheduler can improve CPU utilization by 5% when delivering the `MPGDec` 99% (as opposed to 100%) deadline guarantees.

- Second, `MPGDec`'s instantaneous cycle demand changes greatly (up to a factor of three), while its demand distribution is much more stable. For example, the cumulative probability curves for the first 50 jobs, the first 100 jobs, and all 7691 jobs are almost the same. This stability implies that GRACE-OS can perform proactive internal adaptation for `MPGDec`

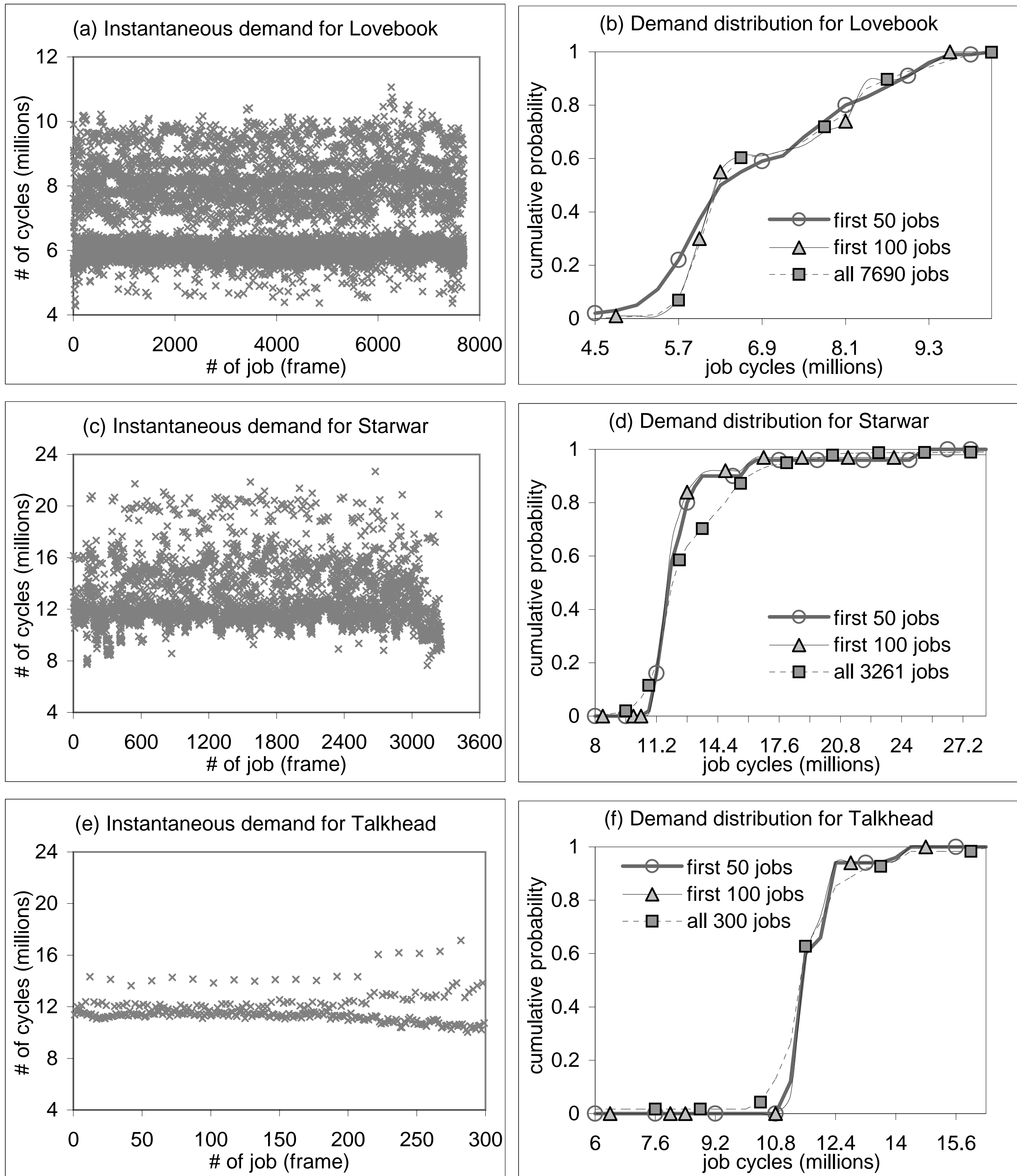


Figure 5.13: Cycle usage and estimated demand distribution of `MPGDec`: its instantaneous cycle demands change greatly, while its demand distribution is much more stable.

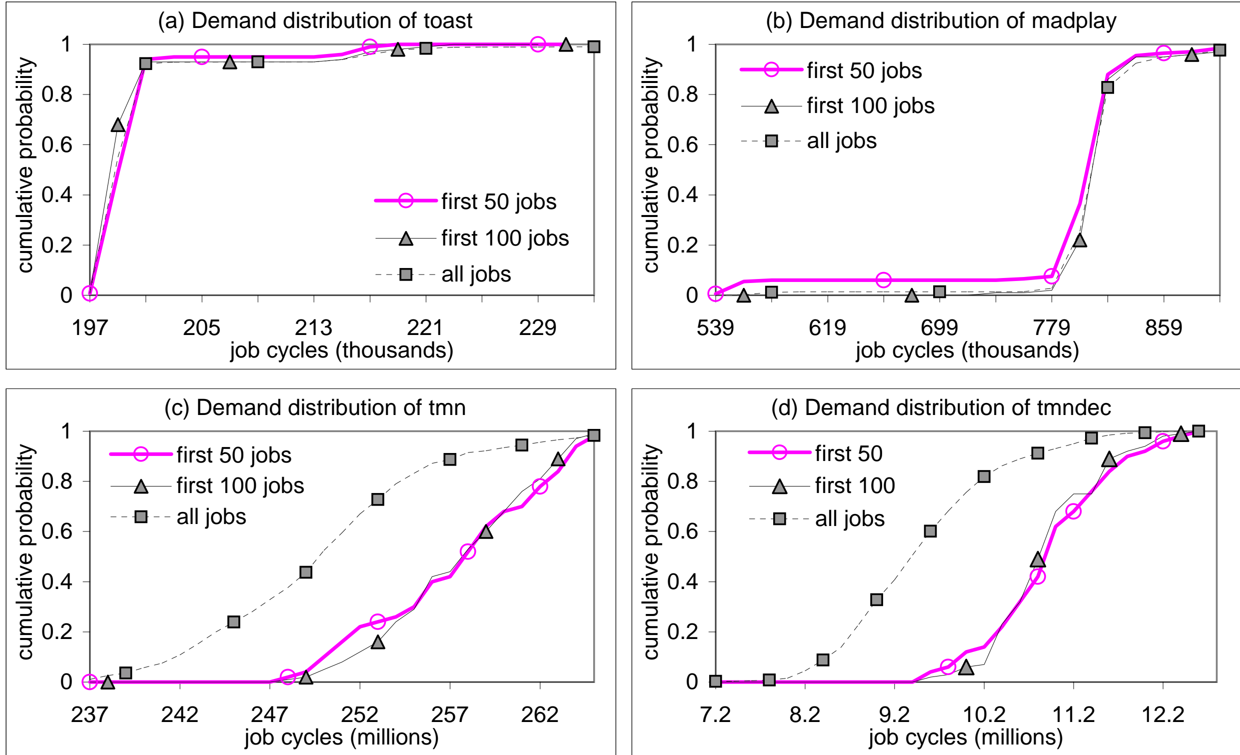


Figure 5.14: Stability of demand distribution of other codecs: *toast* and *madplay*'s are stable, and *tmn* and *tmndec*'s change slowly and smoothly.

based on a small part of its cycle usage history (e.g., cycle usage of the first 50 jobs).

We repeat the experiment for other inputs of the MPEG decoder. Figure 5.13-(c) and (d) plot the instantaneous demand and demand distribution for *Starwar.mpg*, which contains a lot of scene changes. We can also see that the demand distribution of this video is also stable. Figure 5.13-(e) and (f) plot the instantaneous demand and demand distribution for *Talkinghead.mpg*, which contains few scene changes. These results show that the demand distribution of this video is more stable than the instantaneous demand, which does not change substantially.

We also repeat the experiment for other codecs: *toast*, a speech codec, *madplay*, a MP3 audio decoder, *tmn*, an H263 video encoder, and *tmndec*, an H263 video decoder. Figure 5.14-(a) to (d) plot the demand distribution of the *toast*, *madplay*, *tmn*, and *tmndec* codecs, respectively. The results show that *toast* and *madplay* both present low CPU demands; e.g., the 95th percentile of their jobs need less than 2.3×10^5 and 8.6×10^5 cycles, respectively. Further, the probability

distribution of their cycle demands is stable; e.g., the cumulative probability curve for the first 50 jobs is almost the same as that for all jobs.

On the other hand, *tmn* and *tmndec* present high CPU demands; e.g., the 50th percentile of *tmn*'s jobs need more than 2.5×10^8 cycles. Further, their demand distribution changes over time (i.e., for different parts of the input video). The reason is that their input videos have several scene changes and hence require different amount of CPU cycles. Such changes indicate that GRACE-OS needs to dynamically update the demand distribution for *tmn* and *tmndec*. However, the demand distribution of *tmn* and *tmndec* changes in a slow and smooth manner (e.g., there is little variation between the first 50 and 100 jobs). This implies that GRACE-OS only needs to update their demand distribution infrequently (e.g., for every 100 jobs).

5.5 Summary

In this chapter, we discussed how GRACE-OS enforces the coordinated decisions made in the global adaptation, i.e., enables each task to operate at the coordinated QoS level at the coordinated CPU speed. We presented a variable-speed EDF-based scheduling algorithm, which extends traditional real-time scheduling with another dimension— speed. That is, this algorithm decides what tasks to execute, when to execute them, and what speed to execute them.

Although the extended scheduling algorithm provides overrun protection among different tasks, it cannot efficiently handle overruns and underruns. The reason is that this algorithm enforces the coordinated CPU allocation, which is based on the long-term CPU demand prediction. At runtime, multimedia tasks dynamically change their cycle demand due to the variations in the input data. We therefore proposed two sets of algorithms to handle the variations in CPU demand.

The first set of algorithms are *reactive* and adapt the CPU speed and allocation when a job overruns or underruns or when a task consistently overruns or underruns. The second set of algorithms are *proactive* and adapt the CPU speed when executing each job. These two sets of algorithms also adapt at different time granularity and use different prediction methods: The reactive algorithms

Table 5.1: Comparison between reactive and proactive internal adaptation.

	Reactive Adaptation	Proactive Adaptation
granularity	per-job or across multi-job	intra-job
when	in response to overrun or underrun	within job execution
prediction	instantaneous cycle demand	probability distribution of cycle demand

adapt at the granularity of per-job or multi-job execution and are based on the prediction of the instantaneous cycle demand. The proactive algorithms adapt within each job execution and are based on the prediction of the probability distribution of cycle demand. Table 5.1 summarizes the differences between the reactive and proactive adaptations.

These internal adaptation algorithms show the flexibility of GRACE-OS's extended real-time scheduling algorithm. In particular, when used together with internal adaptation, the scheduling algorithm can handle overruns and underruns by changing the CPU speed without violating timing requirements of multimedia tasks. As a result, GRACE-OS can enable each task to provide the utility expected by the coordinator with minimum energy.

Chapter 6

Implementation

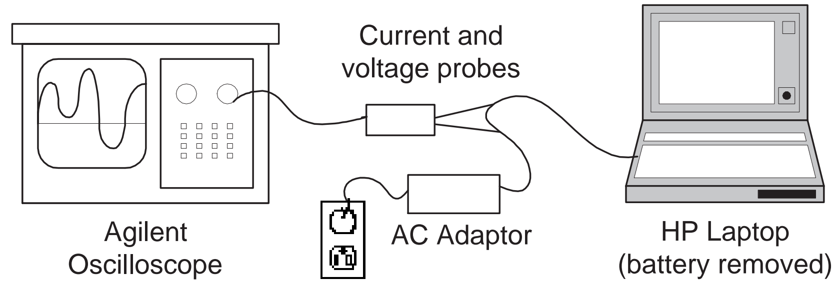
We have implemented GRACE-OS in the Linux kernel on an HP N5470 laptop, which has an adaptive processor. To evaluate GRACE-OS, we have also implemented a prototype of the GRACE cross-layer adaptive system [109]. This prototype integrates and coordinates the adaptation of the CPU speed (frequency/voltage), operating system scheduling, and multimedia quality. The implemented adaptive multimedia tasks are an MPEG decoder and an H263 encoder. In this chapter, we introduce the hardware platform of our implementation and describe the implementation of GRACE-OS and two adaptive multimedia codecs.

6.1 Hardware Platform

The hardware platform for our implementation is an HP Pavilion N5470 laptop. This laptop has a single AMD Athlon processor [5]. To provide energy saving capability, the Athlon processor can dynamically change speed at runtime, trading off performance for power. In particular, this processor supports six different speeds; different speeds have different voltages (Table 6.1). Similar to Intel's *speedstep* technology, AMD's *PowerNow* technology allows the software to change the

Table 6.1: Supported speed and voltage of the Athlon CPU.

Speed (MHz)	300	500	600	700	800	1000
Voltage (Volt)	1.20	1.20	1.25	1.30	1.35	1.40



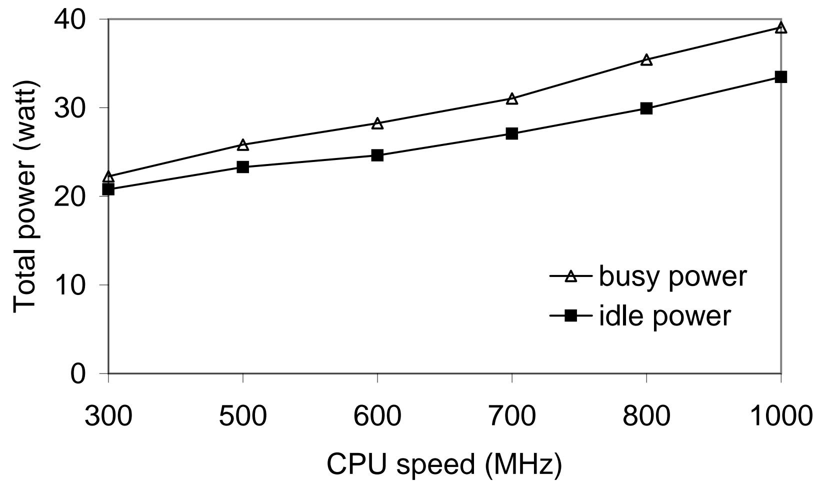


Figure 6.2: Total power consumed by the laptop at different speeds: Each power value is the average of 2000 measurements.

speed and voltage of the CPU through the Advanced Configuration and Power Interface (ACPI) standard [26]. Specifically, the operating system can control the speed and voltage at runtime.

The Athlon CPU consumes different power at different speeds. We are more interested in the total power consumed by the laptop, rather than the CPU power only, at different speed since our goal is to reduce the total energy consumed the laptop. To measure the total power, we use an Agilent 54621A oscilloscope to measure the power consumed by the laptop. Specifically, we remove the battery from the laptop and measure the current and voltage from the input cord of the AC adaptor (Figure 6.1). The product of the current and voltage is the power consumed by the laptop. Figure 6.2 shows the total power of the laptop when the CPU is busy or idle at different speeds.

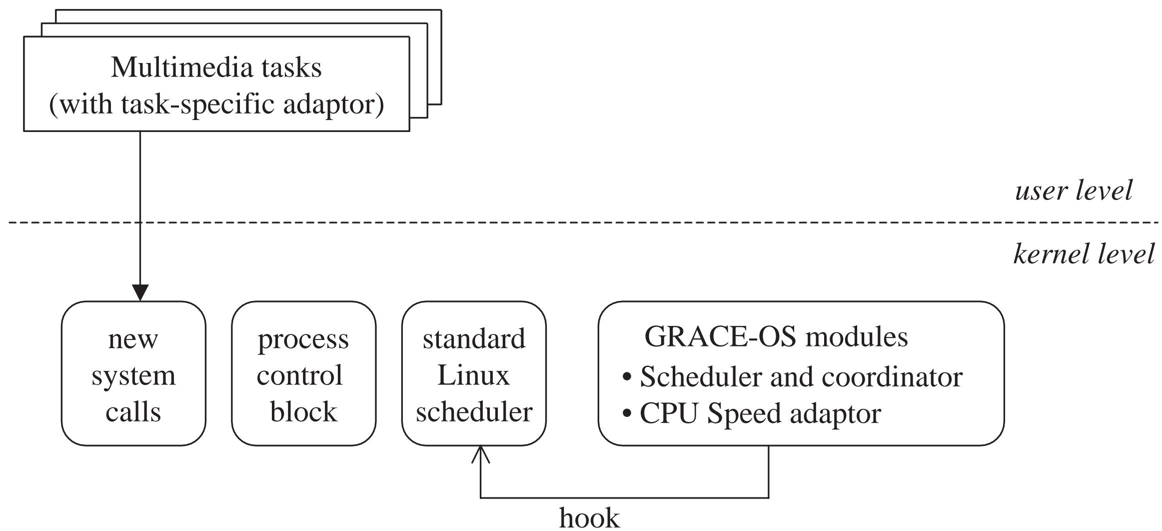


Figure 6.3: Software architecture of GRACE-OS implementation.

6.2 Implementation of GRACE-OS

GRACE-OS is implemented as a set of patches and modules that hook into the Linux kernel 2.6.5. Figure 6.3 illustrates the software architecture of the prototype implementation, which is similar to the design architecture in Figure 3.4. Note that we combine the soft real-time scheduler and coordinator together and implement the CPU adaptor in the operating system. The reason for implementing the CPU adaptor in the operating system, rather than in the hardware layer, is that GRACE-OS makes the decision on the speed adaptation.

The entire implementation contains 2605 lines of C code, including about 185 lines of modification to the Linux kernel, primarily for cycle profiling and speed adaptation during context switches. We next describe the five major issues in the implementation.

6.2.1 Adding New System Calls

We add six new system calls to support soft real-time requirements of multimedia tasks. These system calls enable tasks to communicate with the kernel on the CPU demand and adaptation. Table 6.2 shows these system calls. Table 6.3 gives a sample code on how to use these new calls. We now describe these system calls in detail.

Table 6.2: New system calls for GRACE-OS

System Call	Description
`EnterSRT`	Indicate that the calling task is a soft real-time task.
`SetQoSLevel`	Set the utility and CPU demand for each supported QoS level.
`FinishSetup`	Complete the QoS setup and trigger a global adaptation.
`GetQoSLevel`	Get the coordinated QoS level of the calling task.
`FinishJob`	Indicate that the calling task has finished a job. For the purpose of cycle profiling, this system call returns the number of cycles the job has used.
`ExitSRT`	Indicate that the calling task exits soft real-time mode.

1. A task uses `EnterSRT()` to tell the kernel that the task is a soft real-time task and requires performance guarantees from the operating system.

2. A task uses `SetQoSLevel()` to specify all QoS levels the task supports. This system call tells the kernel the id, utility, CPU demand (period and statistical cycle demand) of each QoS level. The id is associated with the application-level QoS parameters such as frame rate.

3. After setting all QoS levels, a task uses `FinishSetup()` to trigger a global adaptation. Upon this system call, the operating system performs global adaptation to determine the operating QoS level and CPU allocation of the task.

4. A task performs multimedia processing periodically, i.e., executing a job such as frame decoding per period. Before starting a job, a task retrieves the coordinated QoS level using the system call `GetQoSLevel()` and changes the application-level QoS parameters correspondingly. Note that the coordinated QoS level may change due to the joining or leaving of other tasks. If the QoS level changes, the task adapts its QoS parameters again.

5. At the end of job, the task uses `FinishJob()` to tell the kernel that the task has finished a job[1]. So, the kernel can tell if the task misses its deadline and may reclaim the residual cycle budget of the task.

[1] Multimedia tasks often tell the kernel about their jobs via system calls; e.g., when an MPEG decoder finishes a frame decoding, it may call `sleep` to wait for the next frame. Although not explicit here, we can use these system calls to replace `FinishJob`, similar to the approach proposed by Banachowski et al. [12]

Table 6.3: Sample code of an adaptive multimedia task

```
int main()
{
    // 1. Profiling and estimation
    Profile the demand distribution for the input stream.

    // 2. QoS setup
    EnterSRT()
    for each QoS level //set QoS levels
       SetQoSLevel()
    FinishSetup()

    // 3. runtime
    for each job {      //execute jobs periodically
       GetQoSLevel()    //get coordinated QoS level
       ConfigureQoS()   //configure QoS parameters
       DoAJob()         //e.g., decoding a video frame
       FinishJob()
    }
    ExitSRT()           //release CPU
}
```

6. Finally, after finishing all jobs, a task uses `ExitSRT()` to release the CPU reservation. Upon this system call, the operating system reports the statistics (such as deadline miss ratio) about the task and performs another global adaptation for the remaining tasks.

We implement the above system calls (Table 6.2) in the soft real-time scheduler, which is a loadable kernel module. A challenging problem here is that Linux kernel 2.6 does not expose the symbol `sys_call_table` for loadable kernel module. The symbol `sys_call_table` defines the entry for each system call. To address this problem, we take the following steps:

- First, we add an entry for each of these system calls in file `entry.S`. These entries extend the `sys_call_table` for the new system calls.
- Second, we add a dummy implementation in file `sched.c` to compile the kernel.
- Third, the dummy implementation is replaced with the real implementation when the soft

Table 6.4: Modified process control block

```
// file sched.h

struct task_struct {
    ...

#ifdef CONFIG_UIUC_GRACE
    /* for profiling */
    unsigned long long last_sample_cycles;
    unsigned long job_cycles;

    /* for intra-job DVS */
    struct dvsPnt_struct *speed_schedule;
    unsigned short dvsPnt_count, current_dvsPnt;
#endif /*CONFIG_UIUC_GRACE*/
};
```

real-time scheduler module is loaded. To map the entry of a system call to its implementation, we find the address of the unexposed `sys_call_table` in the file `System.map` and use this address as `sys_call_table` in the soft real-time scheduler module.

6.2.2 Modifying the Processor Control Block

We add five new attributes into the process control block (i.e., the `task_struct`), as shown in Table 6.4. The first two attributes are used for cycle profiling. In particular, `job_cycles` records the number of cycles the task consumes for each job. To profile the number of cycles, we accumulate the number of cycles elapsed during the job execution (i.e., the interval from the task's switch-in to its switch out). The attribute `last_sample_cycles` remembers the CPU cycle count, read from a system register, when the task is switched-in last time.

The last three attributes are used for enforcing the speed schedule for proactive internal adaptation. Specifically, `speed_schedule` is a list of speed scaling points, which define how to accelerate the CPU for a job execution; `current_dvsPnt` specifies the current speed point for

Table 6.5: Value of register `FidVidCtl` for different speeds.

Speed (MHz)	300	500	600	700	800	1000
Register Value	1250064	1249540	1248774	1248520	1248266	248270

the task's execution and changes during the task execution. In Section 6.2.5, we will show how to use these new attributes in the standard Linux scheduler.

6.2.3 Implementing the CPU Adaptor

The Athlon CPU in the HP laptop allows the operating system kernel to change its speed at run-time. We therefore implement the CPU adaptor in the kernel by adding a new DVS kernel module. Specifically, we change the CPU speed by writing the frequency and corresponding voltage to a system register `FidVidCtl` using the following statement.

```
wrmsr(FidVidCtl, register_val);
```

Table 6.5 shows the value of this register (`register_val`) for different speeds.

Although the CPU adaptor is implemented for the Athlon CPU, the adaptor provides a simple, clean interface for speed setting, and is separated from the DVS decision maker (the soft real-time scheduler in our case). In doing so, we improve the flexibility and reusability of our implementation: We can apply GRACE-OS to other processors by replacing only the speed setting module. For example, we have successfully ported GRACE-OS to an IBM thinkpad T40 laptop, which has an adaptive Intel Pentium-M processor.

6.2.4 Implementing the Soft Real-Time Scheduler

We integrate the coordinator into the soft real-time scheduler. The real-time scheduler is hooked into the standard Linux scheduler, rather than replacing the latter. We do this for a number of reasons. First, multimedia applications can take advantage of the existing operating system services and libraries such as X server. As a result, we can validate GRACE-OS with typical mul-

Table 6.6: High resolution timer to trigger soft real-time scheduling

```
static struct timer_list timer;
static void reset_mytimer(void) {
    timer.function = timer_tick;
    timer.expires = jiffies;
    timer.sub_expires = get_arch_cycles(timer.expires);
    timer.sub_expires += usec_to_arch_cycle(1000);
    while(timer.sub_expires >= arch_cycles_per_jiffy){
        timer.expires ++;
        timer.sub_expires -= arch_cycles_per_jiffy;
    }
    if (timer_pending(&timer))
        mod_timer(&timer, timer.expires);
    else
        add_timer(&timer);
}
```

timedia applications. Second, we can easily support the coexistence of real-time and best-effort applications. Finally, we can minimize the modification to the operating system kernel, thereby improving the reusability of our implementation. For example, we have easily ported GRACE-OS from Linux kernel 2.4 to kernel 2.6, which changes substantially from kernel 2.4.

To improve the scheduling granularity, we patch the kernel with the *high-resolution-timer* patch [8] and add a periodic, one millisecond resolution timer into the kernel. The real-time scheduler is attached as the call-back function of the timer and hence is invoked every millisecond. Table 6.6 shows the code to do this. When the timer expires, the real-time scheduler is invoked to perform real-time scheduling as follows: (1) it checks the cycle budget of the current task. If the budget is exhausted, it sets the current task's scheduling policy to best-effort mode for overrun protection. (2) It checks if it is necessary to change the speed for the current task by advancing its speed schedule. (3) Finally, it invokes the standard Linux scheduler, which in turn dispatches a real-time task for execution.

In addition to soft real-time scheduling, the real-time scheduler also implements the new system calls (Table 6.2). Specifically, the real-time scheduler triggers global adaptation upon the system

calls `FinishSetup()` and `ExitSRT()`. If a task is admitted, its scheduling policy is set to `SCHED_FIFO`. The standard Linux schedule uses the scheduling policy and real-time priority together to decide the order to execute real-time tasks. The real-time scheduler also sets the real-time priority for each multimedia. To do this, the real-time scheduler suspends a task when it calls `FinishJob()` and wakes up the task when it begins a new period. At this time, the real-time scheduler updates the deadline for the task and adjusts its real-time priority by comparing its deadline with other real-time multimedia tasks.

A challenging problem here is that unlike kernel 2.4, kernel 2.6 uses an O(1) scheduler to schedule all tasks. Specifically, the standard Linux scheduler maintains a run queue and always dispatches the first task in the run queue. This means that when our real-time scheduler changes the real-time priority of a task, the task needs to be put into the proper position in the run queue. To do this, we add a function in `sched.c` to allow the real-time scheduler to set real-time priority and set a task's position in the run queue. In this way, we maintain the O(1) scheduling algorithm of the kernel 2.6 while adding real-time support.

6.2.5 Modifying to Standard Linux Scheduler

We modify the standard Linux scheduler to add cycle profiling and speed setting. When the `schedule()` function is invoked, if a context switch happens, the Linux scheduler does some housekeeping for the *switch-out* task. First, the scheduler increases the cycle counter of the task by the number of cycles elapsed since its last switch-in. Second, the cycle budget of the task is decreased by the same amount. Finally, the scheduler advances the current scaling point of the task if its cycle counter reaches the cycle number of the next scaling point.

The Linux scheduler then sets the speed for the *switch-in* task based on its current scaling point. This task will execute at the new speed after the context switch. Table 6.7 shows the modification to context switch in the standard Linux scheduler.

Table 6.7: Modification to the standard Linux scheduler

```
// file sched.c

asmlinkage void schedule(void) {
     ...
#ifdef CONFIG_UIUC_GRACE
     /* profiling */
     rdtscll(next->last_sample_cycles);
     prev->job_cycles += next->last_sample_cycles
                       - prev->last_sample_cycles;

     /* advancing speed schedule if necessary */
     if(prev->speed_schedule
        && prev->current_dvsPnt < prev->dvsPnt_count - 1
        && prev->job_cycles >=
        prev->speed_schedule[prev->current_dvsPnt].cycle)
        prev->current_dvsPnt++;

     /* Set speed for the next task*/
     if(next->speed_schedule)
        SetSpeed(next->speed_schedule[next->current_dvsPnt].speed);
#endif /*CONFIG_UIUC_GRACE*/
     ...
}
```

6.3 Implementation of Adaptive Multimedia Tasks

To test GRACE-OS, we have also implemented two adaptive multimedia applications, an MPEG decoder and an H263 encoder. These two tasks support multiple QoS levels, trading off multimedia quality for CPU and energy demands.

6.3.1 Adaptive MPEG Decoder

The adaptive MPEG decoder is based on the Berkeley MPEG tools [22]. The original Berkeley MPEG decoder can decode an MPEG video with different dithering methods, such as color and gray, by specifying an option `-dither` when starting the decoder. Different dithering methods

presents different perceptual quality and consumes different amount of CPU cycles to decode a frame.

To support the dithering adaptation at runtime, we modify the Berkeley MPEG decoder as follows. First, we instrument it with the new system calls (Table 6.2). Specifically, the decoder tells GRACE-OS four different dithering methods: `gray`, `mono`, `color`, and `color2`, when the decoder starts playing a video. Second, before decoding a frame, the decoder retrieves its dithering method from GRACE-OS. If the dithering method changes, we need to apply the new dithering method to the next frame. However, it is difficult to change the dithering method directly since the decoder initializes several decoding parameters at the beginning.

We therefore stop the current decoding thread and start a new thread by specifying the new dithering method. When the new thread starts, it initializes the decoding parameters with the new dithering method and then continues to play the video from the current frame number. This QoS adaptation may incur a large overhead due to the thread restart. We therefore adapt the quality only in global adaptation, which happens at coarse time granularity.

Although the adaptive MPEG decoder is single-threaded application, it uses the X server to display the decoded image. As a result, the execution of the MPEG decoder may be delayed due to the synchronization with the X server, which usually runs in the best-effort mode. To address this dependency, we use the *priority inheritance protocol* [94] to increase the priority of the X server. In particular, the MPEG decoder sets the priority of the X server as its own just immediately before calling the X server for displaying the decoded image. In this way, the X server will be executed immediately even if there are some other real-time tasks. When the X server returns, its priority is reset to best-effort.

6.3.2 Adaptive H263 Encoder

The adaptive H263 encoder [91] is based on the TMN (Test Model Near-Term) encoder [1], which encodes standards-compliant H263 streams. We modify the encoder to trade off computational complexity against the number of cycles demanded for encoding. Specifically, the adaptive

encoder can change the quantization parameter at the beginning of each frame.

Similar to the above MPEG decoder, we modify the H236 encoder as follows. First, we instrument it with the new system calls (Table 6.2). Specifically, the encoder tells GRACE-OS three quantization parameters: 5, 18, and 31, before encoding a stream. Second, before encoding a frame, the encoder retrieves its quantization parameter from GRACE-OS and always uses the latest quantization parameter to encode the next frame. Note that unlike the above adaptive MPEG decoder, the adaptive H263 encoder does not need to restart in case of quality adaptation, thus incurring much lower overhead for adaptation.

6.4 Summary

This chapter described the implementation of GRACE-OS in the Linux kernel 2.6.5 on an HP laptop with an AMD Athlon processor. We divided GRACE-OS into several modules and provided clear interfaces among these modules. In particular, we isolated the CPU adaptor, which is dependent on the hardware platform. As a result, we can apply GRACE-OS to other platforms by replacing only the CPU adaptor.

We addressed several challenges when implementing GRACE-OS in the Linux kernel 2.6.

- Linux is a best-effort operating system. GRACE-OS, however, needs soft real-time scheduling to support multimedia QoS.

  We implemented a soft real-time scheduler, which allocates CPU cycles to individual tasks and enforces the allocation for QoS provisioning. This real-time scheduler is hooked to the standard Linux scheduler. As a result, multimedia applications can take advantage of the services and libraries provided by the Linux operating system.

- The standard Linux timer has a resolution of 10 milliseconds, which is too coarse for the scheduling and speed adaptation in GRACE-OS.

  We patched the kernel with the high resolution timer and added a one-millisecond-resolution

timer. This timer periodically triggers the real-time scheduler, thus improving the real-time scheduling granularity to one millisecond.

- The new kernel 2.6 changes much from the kernel 2.4. In particular, the kernel 2.6 uses an O(1) schedule and hides system call table from load kernel modules.

  We allowed the real-time scheduler to access the run queue defined in the kernel to put multimedia tasks into the proper position of the run queue based on their deadline. In this way, we maintained the O(1) scheduling algorithm while adding real-time support. We also exposed the system call table from the `System.map` file to add new system calls in the real-time scheduler module.

Chapter 7

Experimental Evaluation

We have experimentally evaluated GRACE-OS on a real system with adaptive processor and multimedia applications. In this chapter, we describe the experimental setup for the evaluation and then report the experimental results, including overhead, energy saving and QoS provisioning. The overhead results justify our global and internal hierarchial adaptation approach. The energy and QoS results demonstrate the benefits of the cross-layer adaptation supported by GRACE-OS.

7.1 Experimental Setup

In this section, we describe the multimedia applications used in our experiments. We describe their inputs, supported QoS levels, as well as CPU demand and utility for each QoS level. We then introduce our interested metrics for evaluation.

7.1.1 Experimental Applications

Our experiments are performed on the HP N5470 laptop with 256MB RAM. The experimental applications include the adaptive MPEG decoder and H263 encoder (described in Section 6.3) and a non-adaptive H263 decoder. The H263 decoder also uses the X server to display the decoded image; we let the X server inherit the priority from the H263 decoder, same as in the MPEG decoder implementation (Section 6.3). Table 7.1 summarizes these multimedia tasks and their inputs. We have also experimented other audio and speech codecs such as `toast` and `madplay`.

Table 7.1: Experimental multimedia tasks.

Application	Type	Input stream	Jobs	QoS Levels
MPGDec	MPEG video decoder	`Starwars.mpg`	3260	4
H263Enc	H263 video encoder	`Paris.cif`	1065	3
H263Dec	H263 video decoder	`Paris.263`	1065	1

These codecs present a very low CPU demand, e.g., less than 20 million cycles per second (MHz). As a result, there is no much space for energy saving since the CPU can always run at the lowest speed 300 MHz. We therefore do not report the results for these audio and speed codecs in the thesis.

For each codec, we define the CPU demand and utility of each QoS level q as follows:

1. We calculate the period $P(q)$ from the application-level QoS parameters— frame rate using the equation

$$P(q) = \frac{1}{frame\ rate} \tag{7.1}$$

   For a fair comparison among different operating system support in the evaluation, we use the same frame rate and hence same period for all QoS levels of a task.

2. We run the codec and profile its cycle usage for each frame processing off-line. We then use the 95th percentile of cycle usage cross all frames as the statistical cycle demand $C(q)$.

3. We define the utility as a log function of the demanded cycles per second, i.e.,

$$u(q) = w \log \frac{C(q)}{P(q)} \tag{7.2}$$

   where w is the weight of the task. Intuitively, the higher the CPU demand, the higher the utility; the user can also use the weight w to increase the utility of an important application, which may have a low CPU demand. Such a utility function is also commonly used in previous literature [19, 62].

Table 7.2: QoS levels for the three multimedia codecs.

	MPGDec				H263Enc			H263Dec
	dithering				quantization level			non-adptive
QoS level	gray	mono	color	color2	31	18	5	
period (ms)	50	50	50	50	150	150	150	40
cycles ($\times 10^6$)	12.77	16.02	19.73	20.07	55.06	61.76	90.18	7.78
bandwidth (MHz)	255.4	320.4	394.6	401.4	367.1	411.7	601.2	194.5
utility	2.407	2.506	2.596	2.604	2.565	2.615	2.779	2.289

Table 7.2 summarizes the QoS levels of these three codecs.

7.1.2 Metrics

We measure five metrics for the evaluation:

- **Overhead**. We measure the cost for each operation (such as global and internal adaptation) of GRACE-OS. Unless specified otherwise, we set the CPU to lowest speed, perform the operation, and measure the time elapsed during the operation.

  Although we are unable to measure the energy cost during each operation, we found that the energy cost of GRACE-OS's operations is small and negligible for multimedia execution since their corresponding time cost is small.

- **Energy consumption**. We measure the energy consumed in each experiment using the following equation

$$E = \int_0^T p(f(t))dt \tag{7.3}$$

  where T is the execution time of the experiment, $f(t)$ is the CPU speed at time $t, 0 \leq t \leq T$, and $p(f(t))$ is the total power consumed by the laptop at speed $f(t)$ (Figure 6.2). We use the idle power when the idle task is dispatched (i.e., the CPU is idle).

  Although we are unable to use Agilent oscilloscope to measure the actual energy consumption for a long time interval during the experiments. We verified that for a short time interval,

the measured energy with the oscilloscope is the same with the calculated energy with Equation (7.3). We therefore conclude that Equation (7.3) is valid for energy measurement.

- **Achieved lifetime**. Currently, we cannot measure the actual battery lifetime due to the difficulties in precisely measuring the residual energy in Linux. Instead, we assign an initial energy budget before starting each experiment and decrease the budget by the energy consumed by the laptop as in Equation (7.3). When the budget becomes 0, we say that the battery is exhausted and calculate the achieved lifetime as the time interval from the budget assignment to the exhausted time instance.

- **Deadline miss ratio**. This metric shows how well GRACE-OS meets multimedia timing requirement. Intuitively, the lower the deadline miss ratio is (i.e., the better the multimedia QoS is), the better the operating system support.

- **Accumulated utility**. We define the *accumulated utility* of a task as

$$\bar{u} = (1 - \delta) \times \int_0^T u(t)\, dt \tag{7.4}$$

  where δ is the task's deadline miss ratio, T is its execution lifetime— the time interval from its arrival to its leaving, and $u(t)$ is its configured utility (defined in Table 7.2) at time t. The *accumulated total utility* is defined as the sum of the accumulated utility of all tasks executed during the battery life.

  Compared to the utility function in Table 7.2, the accumulated utility captures the perceptual quality better by considering the missed deadlines, configured quality, and runtime of the task. For example, a user may prefer a small and smooth video (with few missed deadlines) to a large but jerky video, and prefer watching the whole movie with a small screen to watching a part of the movie with a full screen. At the global adaptation time, however, GRACE-OS uses the utility function in Equation 7.2 to make global optimization since by definition, the accumulated utility of a task is known only after the task has finished.

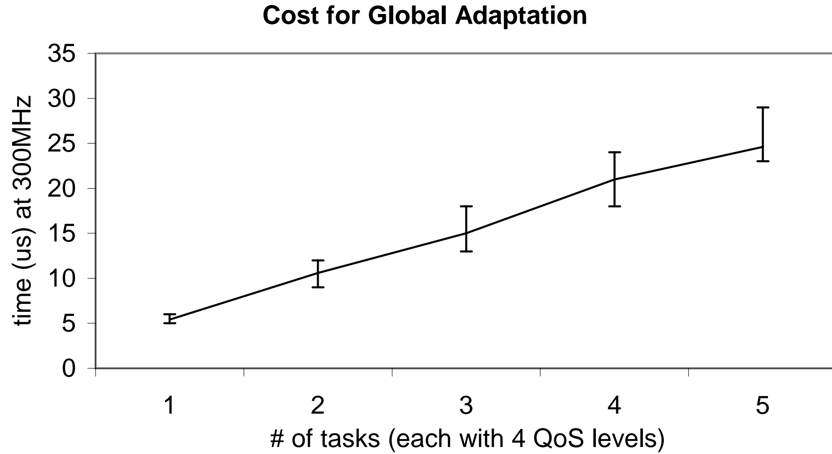


Figure 7.1: Cost of global adaptation: the solid line shows the mean of six measurements and the error bars show the minimum and maximum of the six measurements.

7.2 Overhead

In the first set of experiments, we analyze the overhead of the operations in GRACE-OS. Specifically, we report the cost for global adaptation, internal adaptation in each system layer, real-time scheduling, and the new system calls (Table 6.2).

7.2.1 Cost for Global Adaptation

Global adaptation considers all the combination of the CPU and task QoS level. Since the number of speed options is fixed in our experiments (six speeds for the Athlon CPU), the cost of global adaptation depends on the number of concurrent tasks and the number of their QoS levels. To measure this cost, we run one to five MPEG decoders (mobile devices seldom run more than five active applications concurrently) at a time, set the lowest speed 300 MHz before each global adaptation, and measure the time elapsed during global adaptation based on the optimization policies *maximum-utility* and *desired-life* in the kernel.

Our results show that the policies *maximum-utility* and *desired-life* incur the almost same cost since they use the same dynamic programming algorithm in Figure 4.1. We therefore do not differentiate these two policies in the reported cost (Figure 7.1). We notice immediately that here

the cost of global adaptation is quite small. For example, the cost with five tasks is about 30 microseconds, which is less 0.075% of the time for decoding an MPEG frame.

This seems to indicate that we can invoke global adaptation frequently. However, the cost reported in Figure 7.1 is only for the coordination algorithm. The reported cost does not include time for configuring each layer based on the decisions made in the global adaptation. In particular, the cost for configuring application QoS parameters may be very large, depending on the application. For example, when the adaptive MPEG decoder changes its dithering method, the adaptation cost is in hundreds of milliseconds. This implies that if global adaptation is triggered frequently, it may incur unacceptably large overhead. GRACE-OS hence chooses to trigger global adaptation at coarse time granularity when a task joins or leaves or when the internal adaptation cannot handle the changes in the CPU demand of a task.

Even for multimedia tasks that can adapt QoS with very small overhead (e.g., our implemented H263 encoder), we still cannot invoke global adaptation frequently. The reason is that global adaptation may change the quality of multimedia tasks; frequent quality changes (e.g., fluctuation of a video color) could be annoying to the user.

7.2.2 Cost for Internal Adaptation

Now we analyze the cost for internal adaptation in the CPU and operating system layers. The cost for DVS includes three parts, timer expiration, changing the frequency and stabilizing the voltage. The internal adaptation is driven by a high resolution timer. The cost for the timer expiration is about 1000 cycles and hence very small [100]. To measure the cost for the frequency change, we adjust the CPU from one frequency to another one and measure the time elapsed for each adjustment in the kernel. Figure 7.2 plots the results. The cost for the frequency change is dependent on the destination frequency and is below 40 microseconds. Although we are unable to directly measure the cost for stabilizing the voltage, AMD document [5] reports that this cost is below 100 microseconds. That is, it takes less than 140 microseconds to change the CPU speed once. Therefore, it is acceptable for GRACE-OS to change the speed several (often less than six)

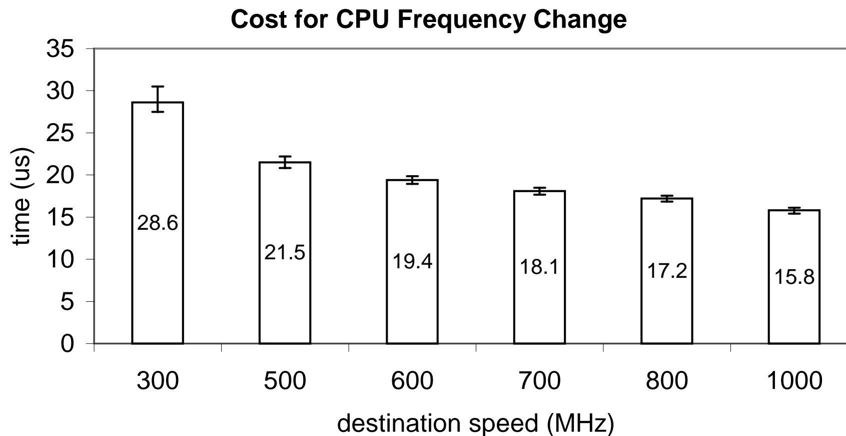


Figure 7.2: Cost of changing the CPU frequency: the bars show the mean of 12 measurements and the error bars show the minimum and maximum of 12 measurements.

times during a job execution, since a job execution often takes tens of milliseconds. Note that with the advances of circuit design, the DVS overhead is becoming smaller; e.g., the lpARM processor can change speed in 1250 cycles and continue operation while changing the speed [81].

Next, we measure the cost for reactive internal adaptation. To do this, we run one `MPGDec`, set the CPU speed to the lowest speed 300 MHz before each per-job and multi-job adaptation, and measure the time elapsed during each per-job and multi-job adaptation in the kernel. The results (Figure 7.3) show that multi-job adaptation has a much larger overhead (in a factor of 100) than per-job adaptation. However, both per-job and multi-job adaptations incur a negligible overhead relative to multimedia processing. For example, the cost of multi-job adaptation is below 22 microseconds, which is less than 0.05% of the time for decoding an MPEG frame.

Finally, we analyze the cost for proactive internal adaptation, primarily for constructing the speed schedule for a task. To do this, we always run the CPU at the lowest speed 300 MHz and measure the time elapsed when using the dynamic programming algorithm to calculate the speed schedule for each task. The results (Figure 7.4) show that this cost is small and negligible relative to multimedia processing. For example, for `H263Enc`, which has 37 cycle groups, the speed schedule calculation takes less than 30 microseconds, while a typical video frame processing takes tens of milliseconds.

We need to point out that GRACE-OS only constructs the speed schedule for a task when its

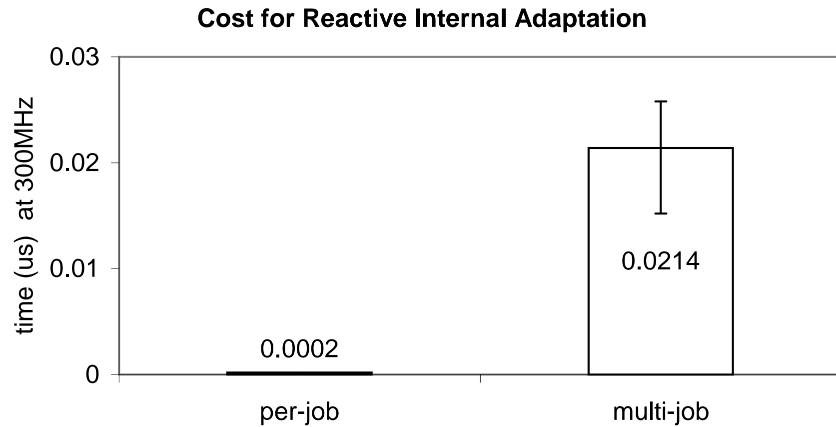


Figure 7.3: Cost of reactive internal adaptation: the bars show the mean of 50 measurements and the error bars show the minimum and maximum of 50 measurements.

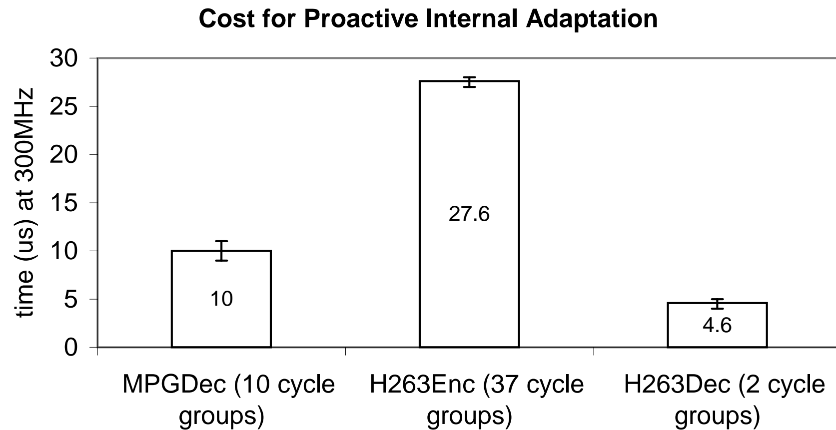


Figure 7.4: Cost of proactive internal adaptation for constructing the speed schedule: the bars show the mean of 6 measurements and the error bars show the minimum and maximum of 6 measurements.

time budget changes due to the entry or exit of other tasks. That is, the construction of speed schedule happens immediately after the global adaptation, which happens infrequently. After the construction, the scheduler only needs to adjust the speed for a task's execution based on the task's speed schedule. This means that the calculation of the speed schedule does not happen frequently.

7.2.3 Cost for Real-Time Scheduling

To measure the cost for soft real-time scheduling, we run different number of tasks concurrently and measure the time elapsed for each invocation of soft real-time scheduling. Figure 7.5 plots

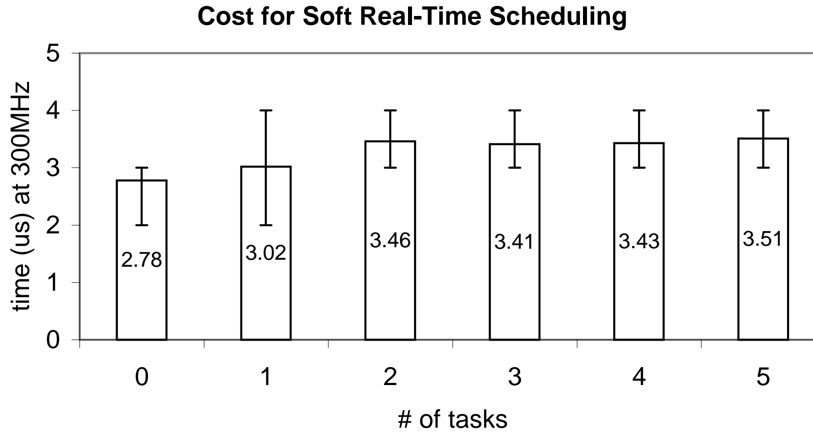


Figure 7.5: Cost of soft real-time scheduling: the bars show the average of 5,000 measurements and the error bars show the 95% confidence intervals.

the results. The scheduling cost is very small and below 4 microseconds, thus negligible during multimedia processing. In terms of relative overhead, the scheduling cost is below 0.4% since the scheduling granularity is 1000 microseconds.

We also found that the cost of soft real-time scheduling does not increase significantly with the number of concurrent tasks. The reason is that like the O(1) scheduling algorithm in Linux kernel 2.6, our soft real-time scheduler also uses an O(1) algorithm. The primary operation of the soft real-time scheduling is to charge the cycle budget of the current task and enforce its speed schedule based on its cycle usage.

7.2.4 Cost for New System Calls

Finally, we measure the cost for each of the new system calls (Table 6.2) in the application level. To do this, we set the CPU to the lowest speed 300 MHz, run the `MPGDec`, and measure the time elapsed during each system call in the application level. Figure 7.6 plots the results, which are negligible relative to multimedia processing, for the following reasons.

- First, although `GetQoSLevel` is called once per job (see the application sample in Table 6.3), the cost per call is very small, about 4 microseconds.

- Second, although `EnterSRT`, `SetQoSLevel`, and `FinishSetup` have a larger cost per

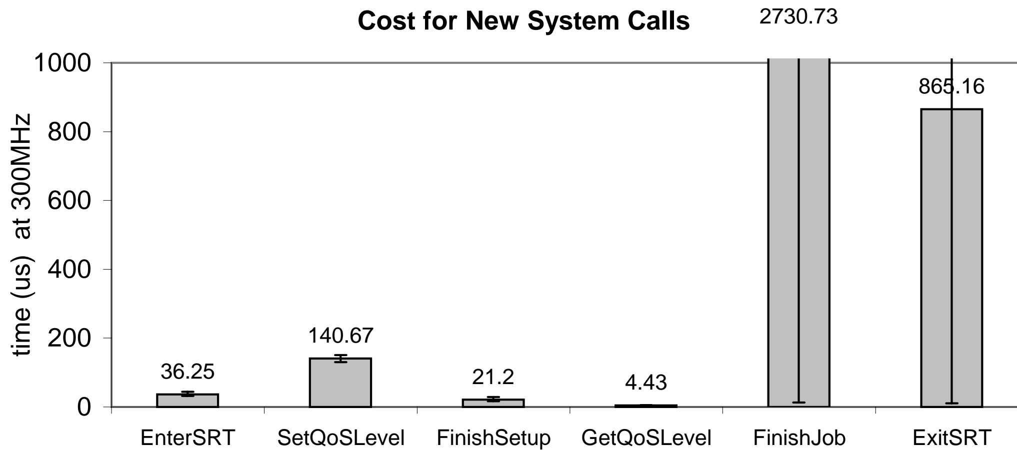


Figure 7.6: Cost of new system calls: the bars show the mean of ten measurements and the error bars show the minimum and maximum of the ten measurements.

call, they are called only once or several times for each task. As a result, their overall cost is still small during task execution.

- Third, although `FinishJob` is called once per job and has a very large cost (in milliseconds) per call, the calling task is suspended until next period anyway. This means that the delayed return of `FinishJob` does not matter from the QoS point of view.

- Finally, although `ExitSRT` has a larger cost per call, it is called only once when the task exits the real-time mode. That is, after calling `ExitSRT`, the calling task enters the best-effort mode; consequently, the delayed return of `ExitSRT` does not affect multimedia QoS.

Another interesting result from Figure 7.6 is that `ExitSRT` and `FinishJob` both exhibit large deviations in their cost. For `ExitSRT`, the calling task is set to best-effort mode and may be dispatched later if there are other soft real-time tasks. As a result, the call `ExitSRT` may return with a large delay. For `finishJob`, the calling task is usually suspended until next period, at which the `finishJob` call returns. However, the task starts a new period immediately, if the task finishes the previous job at or after the deadline.

7.3 Benefits of Global Adaptation

After analyzing the overhead of GRACE-OS and justifying our hierarchical adaptation approach, we now show the benefits of GRACE-OS for QoS provisioning and energy saving. Specifically, we first analyze the benefits of global cross-layer adaptation, and then analyze how much energy can be saved by internal adaptation.

To evaluate the benefits global cross-layer adaptation, we compare GRACE-OS with other systems that adapt only some of the three layers:

- *No-adapt*. No system layer adapts: The CPU runs at the highest speed, tasks run at the highest QoS level, and the operating system scheduler does not handle overruns and underruns.
- *CPU-only*. Same as *no-adapt* except that the CPU adapts when a task arrives or leaves. Specifically, the CPU speed is adjusted based on the total CPU demand of all concurrent tasks, each operating at the highest QoS level.
- *OS-only*. Same as *no-adapt* except that the operating system scheduler uses internal adaptation to handle overruns and underruns. This internal adaptation, however, does not change the CPU speed.
- *App-only*. Same as *no-adapt* except that when a task arrives, it configures its QoS level as high as possible given the currently available CPU resource.
- *CPU-OS*. Joint adaptation in the hardware and operating system layers. The scheduler uses internal adaptation to handle overruns and underruns. The CPU adapts the speed upon task entry or exit and internal adaptation.
- *CPU-app*. Joint adaptation in the hardware and application layers. When a task arrives, it configures its QoS level as high as possible given the currently available CPU resource. The CPU adapts the speed based on the total demand of all concurrent tasks when a task joins or leaves the system.

- *OS-app*. Joint adaptation in the operating system and application layers. When a task arrives or leaves, the operating system coordinates all concurrent tasks to maximize the total utility. The scheduler uses internal adaptation to handle overruns and underruns. In general, *OS-app* represents systems that coordinate and enforce CPU allocation to multiple tasks, such as Q-RAM [89], IRS [38] and DQM [19].

The internal adaptation in the operating system above means the reactive internal adaptation, which changes the CPU allocation to a task. For each of the above adaptation policies, the scheduler also allocates the CPU to individual tasks based on their statistical cycle demand and uses the speed-aware soft real-time scheduling algorithm. If the scheduler cannot allocate CPU resource to a task, the task exits immediately.

Under each of the above adaptive systems, we perform two kinds of experiments:

1. **Single run**. We run each of the three codecs (`MPGDec`, `H263Enc`, and `H263Dec`) one at a time. Each code has the QoS configuration in Table 7.2.

2. **Concurrent run**. We run each of the three codecs together by starting them in the order of `H263Enc`, `MPGEnc`, and `H263Dec` with some time interval. This concurrent run represents the scenario in which we record a video and playback the video to check the quality of multiple segments (e.g., in a multi-video window).

In all experiments, we set the task weight to 1.0. We have also tried other weight values and found that the cross-layer adaptation achieves higher utility than other systems that are oblivious to utility. In particular, the *desired-time* optimization is good to save energy for important applications that start later. We next use the single and concurrent run experiments to evaluate the two global adaptation policies, *maximum-utility* and *desired-time*, in GRACE-OS. In these experiments, GRACE-OS uses the reactive internal adaptation in the CPU and operating system layers.

7.3.1 Maximizing Utility

We first consider the scenario in which the user wants to first the total utility of all current tasks and then minimize energy, e.g., when recording important video and audio. We do the above single and concurrent run experiments and measure the accumulated utility and energy consumption (we are not interested in the achieved lifetime here since the battery has enough energy to finish the experiments). Figure 7.7 reports the utility and energy results. We now use these results to evaluate GRACE-OS.

Compared to *no-adapt*, *CPU-only*, *OS-only*, *app-only*, *CPU-OS*, *CPU-app*, and *OS-app* that adapt only some of the three layers, GRACE-OS achieves similar utility with much less energy for the single runs and achieves much higher utility for the concurrent run. These results clearly show the benefits of cooperative cross-layer adaptation for maximizing multimedia quality. Specifically,

- In the single run cases, GRACE-OS and the zero- or one-layer adaptive systems have a similar utility since they all configure the single task at the highest QoS level. They also reduce the deadline misses for overrun jobs (the deadline miss ratios are below 0.5% and hence are negligible) but use different approaches: GRACE-OS uses internal adaptation to allocate extra cycles, while other systems run overrun jobs in best-effort mode using the unallocated cycles, which exist since the CPU may runs at a higher speed than the total demand due to the discrete frequency options.

  In terms of energy, GRACE-OS saves energy by up to 59%. Relative to *no-adapt*, *OS-only*, and *app-only* that are oblivious to energy saving, the energy benefits of GRACE-OS results from the CPU adaptation since the CPU does not need to always run at the highest frequency. Relative to *cpu-only*, GRACE-OS saves more energy by using internal adaptation to handle underruns. This underrun handling is effective since even with a small deduction of the total demand via the budget reclamation, the CPU may run at the next lower speed.

  Note that GRACE-OS consumes almost the same energy as *CPU-OS* since they have the same adaptation behavior in the single run case. Although GRACE-OS differs from *CPU-*

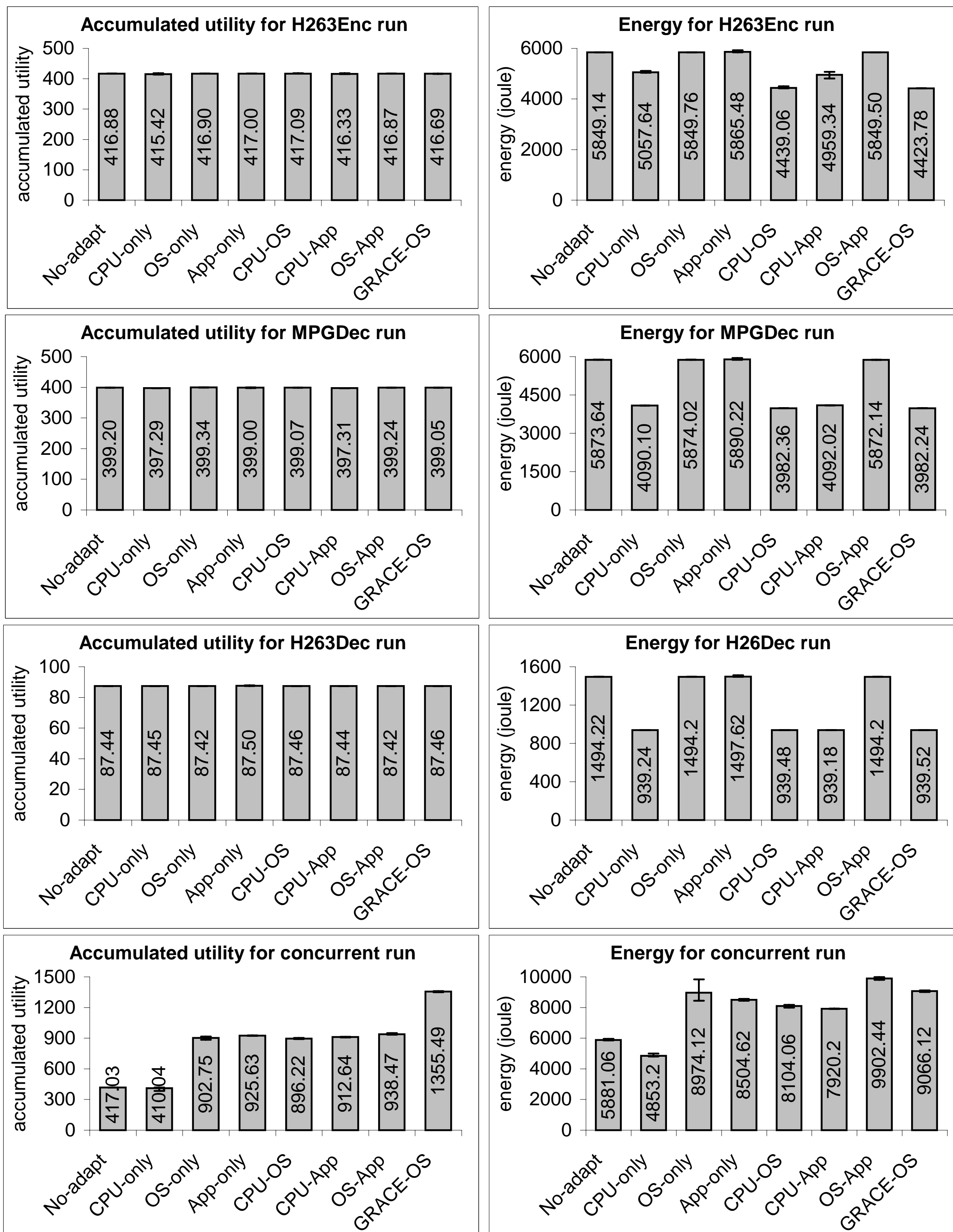


Figure 7.7: Comparing GRACE-OS with other systems for maximum-utility global adaptation: the bars show the mean of five measurements and the error bars show the minimum and maximum of the five measurements.

OS in application adaptation, the single application runs at the highest quality level in these two systems, thus resulting in the same adaptation behavior.

- In the concurrent run case, GRACE-OS improves the utility by 31% to 69% relative to other systems. GRACE-OS accepts all tasks and coordinates them to maximize their total utility under the CPU constraint. In contrast, other zero- or one-layer adaptive systems admit fewer tasks due to the CPU constraint.

  Specifically, compared to *no-adapt*, *CPU-only*, *OS-only*, and *CPU-OS* that are oblivious to utility, the utility benefits of GRACE-OS results from the application adaptation and coordination of multiple tasks for the maximum utility. Compared to *App-only* and *CPU-App* that are not aware of multiple tasks, the utility benefits of GRACE-OS results from coordinating the adaptation of multiple tasks. Compared to *OS-App* that also coordinates multiple tasks, the utility benefits of GRACE-OS results from the lower deadline miss ratio due to the internal adaptation for handling overruns.

  GRACE-OS consumes more energy than other systems only because GRACE-OS admits more tasks and hence needs to run the CPU at a higher speed. This is desirable since in the policy *maximum-utility*, maximizing utility is the primary objective and is more important than saving energy.

7.3.2 Achieving Desired Lifetime

Now we consider the scenario in which the user wants to last the battery for a desired lifetime, e.g., when watching a two-hour movie. The desired lifetime here is defined as the expected runtime of the applications. Table 7.3 shows the desired lifetime for the single and concurrent runs. We repeat the above experiments and measure the achieved lifetime and accumulated utility (we are not interested in energy consumption here since energy is a constraint here for maximizing multimedia utility). When the initial battery energy is high, we always achieve the desired lifetime and get similar utility results as the above experiments in Section 7.3.1. We therefore focuses the cases

Table 7.3: Desired lifetime for the single and concurrent runs.

Experiment	lifetime (seconds)
MPGDec	163
H263Enc	160
H263Dec	43
concurrent	226

when the battery energy is low, i.e., the energy is insufficient for the CPU to run at the highest speed for the whole desired lifetime. Figure 7.8 plots the results.

We first notice that for both the single and concurrent run cases, GRACE-OS almost achieves the desired lifetime and finishes all processed streams. GRACE-OS improves the battery lifetime by up to 57.8% relative to other systems that adapt some of the three layers. The reason is that GRACE-OS considers the energy constraint and is aware of the lifetime, while other systems are oblivious to the lifetime. In particular, GRACE-OS coordinates the CPU hardware, operating system, and application layers for the desired lifetime by limiting the operating CPU speed and hence power (Equation 4.13). This lifetime-aware cross-layer adaptation is especially effective to save energy for important applications that may start later.

In addition to achieving the desired lifetime, GRACE-OS also achieves higher utility than other systems. This clearly shows the benefits of cross-layer adaptation for higher QoS when the battery energy is limited. Specifically, in the single run case, GRACE-OS increases the accumulated utility by up to 45.8% relative to other systems. The reason is that GRACE-OS achieves a longer lifetime (recall that the accumulated utility is the integral of the utility over time). The longer lifetime also explains why *CPU-OS* has a high utility (but less than GRACE-OS).

In the concurrent run case, GRACE-OS increases the utility by 2% to 45.7% than other systems. There are two reasons: First, GRACE-OS has a longer lifetime, as analyzed above. Second, GRACE-OS coordinates multiple tasks to maximize their utility. In particular, the first reason explains why GRACE-OS has a higher utility than *OS-app* that also coordinates tasks but with shorter lifetime; the second reason explains why GRACE-OS has a higher utility than *CPU-OS* that also

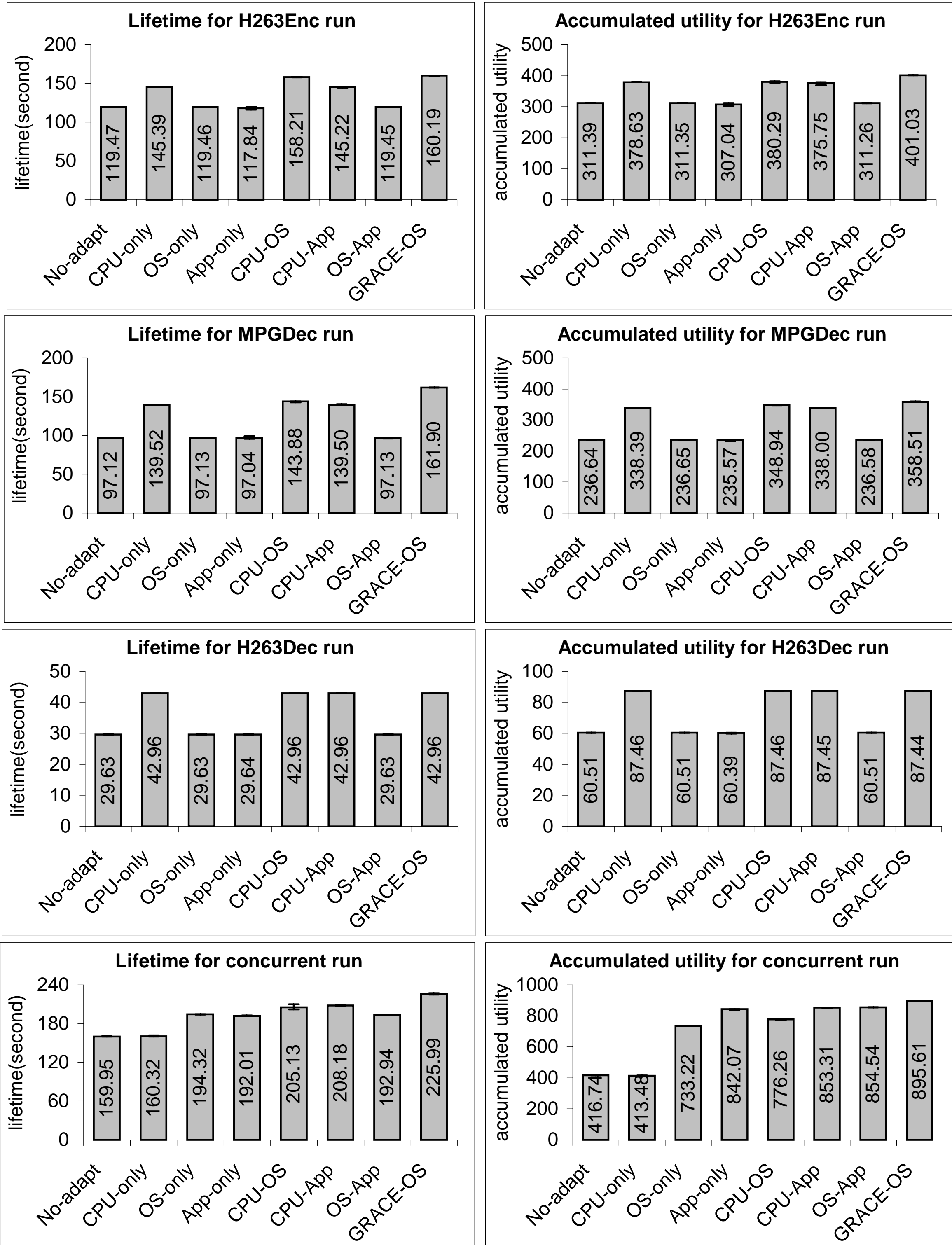


Figure 7.8: Comparing GRACE-OS with other systems for desired-lifetime global adaptation: the bars show the mean of five measurements and the error bars show the minimum and maximum of the five measurements.

has a long lifetime but executes fewer tasks.

Another interesting results is that for the `H263Dec` case, GRACE-OS achieves almost the same lifetime and utility as *CPU-only*, *CPU-OS*, *CPU-app*. The reason is the task `H263Dec` is non-adaptive and can support only one QoS level (Table 7.2). This shows the needs of application adaptation to trade off QoS for energy when the energy is low. On the other, the results also mean that GRACE-OS supports existing non-adaptive application, i.e., GRACE-OS does not perform worse than other systems for non-adaptive applications.

7.3.3 Summary of Global Adaptation Results

Overall, our experimental results show that the global adaptation of GRACE-OS provides significant benefits for multimedia QoS and energy. Compared to previous systems that adapt only some of the layers, GRACE-OS can effectively trade off QoS against energy based on the user's preference: For the policy maximum-utility that first maximizes multimedia utility and then minimizes energy, GRACE-OS improves the accumulated utility by up to 69% or saves energy by up to 59% without affecting multimedia utility. For the policy desired-lifetime that first targets a desired battery lifetime and then maximizes multimedia utility with this lifetime constraint, GRACE-OS always the user-desirable lifetime while increasing the utility by up to 45.8% when the battery energy is low.

7.4 Benefits of Internal Adaptation

After analyzing the benefits of global adaptation of GRACE-OS, we now analyzing the benefits of its internal adaptation. In particular, we evaluate how much energy GRACE-OS saves without substantially degrading multimedia performance. To do this, we focus on the CPU speed adaptation in the internal adaptation and compare the following DVS techniques:

- *No DVS*. This is the baseline technique in which the CPU always runs at the highest speed.

- *Uniform DVS*. It runs the CPU at the coordinated speed for all concurrent tasks until the task set changes. This uniform speed is the lowest speed that is greater than or equal to the total CPU demand $\sum_{i=1}^{n} \frac{C_i}{P_i}$, where there are n tasks and each task is allocated C_i cycles per period P_i. This represents systems that adapt multiple layers only at coarse time granularity [82].

- *Reactive DVS*. It first sets a uniform speed for all concurrent tasks and lowers the speed when a task completes a job early. Specifically, it sets the speed to $\sum_{i=1}^{n} \frac{C_i^*}{P_i}$, where C_i^* is the number of allocated cycles when the ith task releases a job and is the number of consumed cycles when the task completes a job. This represents the reclamation DVS techniques [9, 83, 109].

- *Pro-ideal DVS* [42, 67, 107]. It is the proactive internal adaptation (i.e., adapting the CPU speed statistically during task execution based on the task's demand distribution) with an assumption of an ideal processor. Specifically, it optimizes the execution speed based on the demand distribution of each task. This optimization, however, assumes that the CPU supports a continuous range of speeds and the CPU power is proportional to the cube of the speed. At runtime, the speed calculated in this optimization is rounded to the upper bound of the available speeds.

- *Pro-nonideal DVS*. It is the proactive internal adaptation proposed in this thesis for non-ideal processors that has a discrete set of speed options. Specifically, it optimizes the execution speed based on the demand distribution of each task, the discrete set of speed options, and the total power of the device at different speeds.

Unless specified otherwise, each task specifies its statistical performance requirement as 0.95. The coordinator allocates cycles to each task based on the 95th percentile of demand cross all jobs of the task. That is, the allocation is sufficient for about 95% of jobs, and the desired deadline miss ratio should be below 5%. Under each of the DVS techniques, we repeat the above single and concurrent run experiments with sufficient initial battery energy. In each of the single and

concurrent runs, we measure the energy consumed by the laptop and the deadline miss ratio for each task. Figure 7.9 reports these two metrics. We now use these results to evaluate internal adaptation of GRACE-OS in terms of energy saving and QoS support.

7.4.1 Energy Saving

Compared to the baseline system without DVS, all DVS techniques save energy significantly. The reason is that the CPU does not need to always run at the highest speed. As a result, energy can be saved by adapting the CPU speed based on the application demands.

Compared to the uniform speed technique that performs adaptation only at coarse time granularity, the internal adaptation DVS techniques (i.e., reactive and proactive DVS) consume almost the same energy when running the single `H263Dec`. The reason is that `H263Dec` demands only 194 million cycles per second (MHz). To meet this low demand, the CPU can always run at the lowest speed 300 MHz (for proactive DVS, the speed schedule of `H263Dec` consists of only one speed changing point with speed 300 MHz). Consequently, the energy is already minimized. This indicates that the capability of internal adaptation (and other DVS algorithms) is limited by the lower bound of the supported speeds. In other word, we expect that the internal adaptation can save more energy if the CPU supports more speeds with a lower minimum speed.

In all other cases, the internal adaptation DVS techniques saves energy by up to 10% than the uniform DVS technique. This shows the benefits of internal adaptation at fine time granularity for saving more energy.

Among the internal adaptation approaches, proactive internal adaptation saves more energy than reactive internal adaptation. This clearly demonstrates the benefits of optimizing energy based on the demand distribution of each task. That is, proactive internal adaptation always tries to minimize the energy while not affecting application performance. Compared to *pro-ideal*, *pro-nonideal* reduces the total energy by 2% to 5%. The reason is that *pro-nonideal* explicitly considers the discrete speed options and the total power of the device when calculating the speed schedule, while *pro-ideal* makes a wrong assumption with continuous speeds.

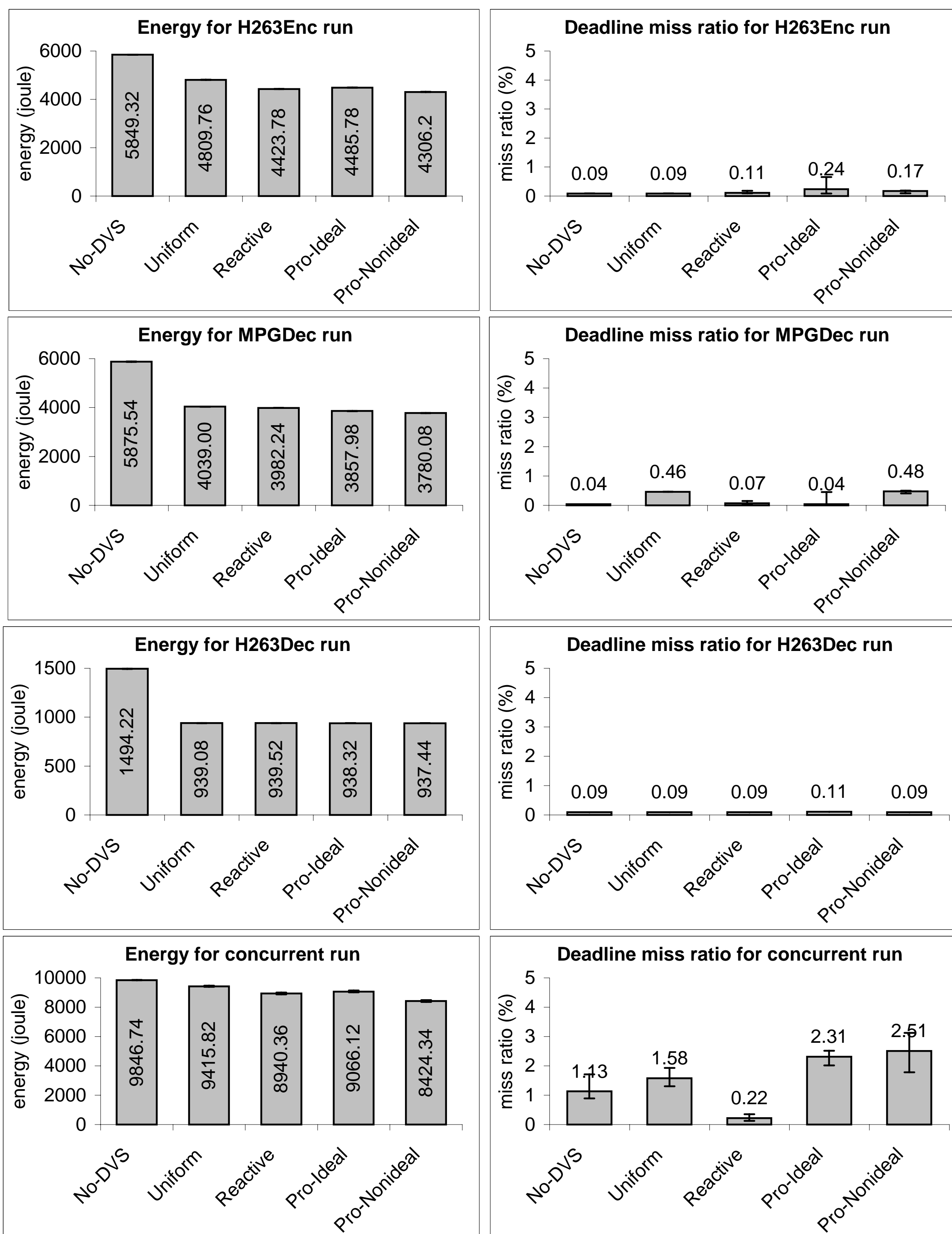


Figure 7.9: Comparing different internal adaptation approaches: the bars show the mean of five measurements and the error bars show the minimum and maximum of the five measurements.

7.4.2 QoS Support

Although slowing down the CPU to save energy, the internal adaptation has no or little impact on multimedia performance. In the single run cases, the deadline miss ratio is very small (below 0.6%), thus meeting application performance requirements, which demands that the deadline miss ratio is below 5%. In particular, `H263Enc` and `H263Dec` meet almost all deadlines since they can utilize the unallocated cycles when they overrun (i.e., need more cycles than the allocated). `H263Enc` has a long period (150 ms) and hence has enough time to catch the deadline. `H263Dec` has a low CPU demand (190 MHz) and there is a lot of unallocated cycles even at the lowest speed 300 MHz.

In the concurrent case, the deadline miss ratio is much higher than the single run case. The reason is that multiple tasks may overrun simultaneously and hence run in best-effort mode to compete for the CPU (recall that the reactive internal adaptation puts an overrun task into best-effort mode if the scheduler cannot handle the overrun, and the proactive internal adaptation puts an overrun task into best-effort immediately). However, the miss ratio is still lower than the application requirement (5%). In particular, the reactive internal adaptation can reduce the deadline miss ratio significantly by allocating extra budget to the overrun tasks. The proactive internal adaptation, on the other hand, does not handle overruns, thus resulting in higher deadline misses.

7.4.3 Summary of Internal Adaptation Results

Overall, our experimental results show that the internal adaptation of GRACE-OS saves energy substantially with no or little impact on multimedia QoS. In particular, compared to the uniform DVS which adapts the CPU only at the global adaptation, the internal adaptation of GRACE-OS saves energy by 2% to 8.9%. Among the internal adaptation approaches, the proactive internal adaptation for non-ideal processor is more efficient for saving energy. It saves energy by 0.8% to 3% relative to reactive internal adaptation.

In the experiments, we also find that to better support multimedia QoS, we need to handle the

dependency among tasks. For example, when we previously tried to run the experiments while running the X server in the best-effort, the deadline miss ratios, especially for `MPGDec`, were larger than 5%. With the priority inheritance protocol, we increases the priority of the X server and decrease the missed deadlines significantly.

7.5 Summary

In this chapter, we have experimentally evaluated GRACE-OS with a real system with adaptive processor and video codecs. We have also shown that GRACE-OS incurs acceptable overhead with the global and internal adaptation hierarchy.

For benefit evaluation, we have divided the experiments into two parts evaluate the global and internal adaptation separately. For global adaptation, we compared GRACE-OS with other systems that adapt some of the system layers. Our results show that by coordinating the adaptation in all three layers, GRACE-OS can effectively trade off QoS against energy based on the user's preference. For the policy *maximum-utility*, GRACE-OS improves the accumulated utility by up to 69% or saves energy by up to 59% without affecting multimedia utility. For the policy *desired-lifetime*, GRACE-OS always the user-desirable lifetime while increasing the utility by up to 45.8% when the battery energy is low.

For internal adaptation, we compared GRACE-OS with systems that adapt all three layers only at coarse time granularity. In particular, we compare GRACE-OS with the uniform DVS case, which always runs the CPU at the speed coordinated in global adaptation. Our results show that the internal adaptation of GRACE-OS further saves energy by 2% to 8.9% than the uniform DVS. Among two different internal adaptation approaches, the proactive method saves more energy by 0.8% to 3% than the reactive method since the former tries to minimize energy during the internal adaptation.

Chapter 8

Related Work

This chapter reviews current research work related to GRACE-OS. We first compare GRACE-OS with other soft real-time scheduling techniques, which are commonly used to support multimedia QoS requirements. We then describe research results on power management which are leveraged by GRACE-OS. Finally, we compare GRACE-OS with other coordinators that coordinate adaptation in different system layers.

8.1 Soft Real-Time Scheduling

To provide a desirable Quality of Service (QoS), multimedia applications present CPU resource requirements that need to meet in soft real-time, e.g., decoding a video frame within some time. *Soft real-time scheduling* is a common mechanism to support the demanding resource requirements of multimedia applications on open computing environments, where multimedia applications share the CPU with other applications.

In general, soft real-time scheduling integrates predictable CPU allocation (such as proportional sharing [23, 29, 40, 76] and reservation [25, 71, 54, 88]) and real-time scheduling algorithms (such as earliest deadline first, or EDF, and rate monotonic [65, 66]). The proportional sharing mechanism (e.g., SFQ [40], SFS [23], BVT [29], and SMART [76]) associates a weight, e.g., 10%, to each application and allocates processing time in proportion to this weight. The major goal of proportional sharing is to achieve a fair CPU allocation regardless of variation in

the application workload. On the other hand, resource reservation (e.g., in RT-Mach [71], Rialto [54], Resource Kernel [77], DQRM [18], and DSRT [25]) allows each application to reserve some processing time periodically (e.g., 5 milliseconds every 30 milliseconds) based on their QoS requirements. The scheduler makes admission control on the reservation request and provides resource and hence QoS guarantees to the admitted applications.

GRACE-OS also uses soft real-time scheduling to support multimedia QoS, but distinguishes itself from the above soft real-time scheduling approaches for three reasons: First, GRACE-OS performs the scheduling on a dynamic processor where the CPU speed changes dynamically, while previous work implicitly assumes a constant CPU speed. The variable speed brings new challenges to soft real-time scheduling, e.g., how to enforce the allocation (share or reservation) when the CPU speed changes. Second, GRACE-OS derives the CPU demand for each application through an automatic profiling, while previous work typically assumes that the CPU demand is known in advance. Finally, GRACE-OS allocates CPU to each application statistically based on its performance requirement (i.e., the probability to meet the deadline). This statistical allocation improves the CPU utilization and also provides more opportunity for energy saving.

8.1.1 Statistical Scheduling

Multimedia applications are soft real-time applications; that is, unlike hard real-time applications that require the worst-case guarantees, multimedia applications require only statistical performance guarantees, e.g., meeting 96% of deadlines. Several groups have also studied soft real-time scheduling for such statistical guarantees.

Gardner [36] proposed a stochastic time demand analysis technique to compute the bound of deadline miss ratio for fixed-priority systems. Such computation is based on the runtime execution by analyzing the time demand of an application and other applications with higher priority. In contrast, GRACE-OS aims for dynamic-priority (EDF-based) systems, and delivers statistical guarantees by allocating cycle budget based on the probability distribution of cycle demand of each individual application.

Hamann et al. [46] and Wang et al. [101] proposed scheduling techniques to provide statistical guarantees for imprecise computations and differentiated services, respectively. Both approaches assume a predefined stochastic demand distribution for each application. In contrast, GRACE-OS estimates the demand distribution through an automatic profiling and estimation, and also dynamically adapts to the changes of the demand distribution.

More recently, Urgaonkar et al. [99] proposed automatic profiling and overbooking techniques to provide statistical guarantees for web services. This is similar to the stochastic allocation in GRACE-OS. However, there are two differences. First, their approach profiles resource busy intervals in an isolated environment, while GRACE-OS profiles the actual cycles each application uses at the actual runtime. Second, the overbooking technique aims to support more services in shared hosting platforms, while GRACE-OS aims to save energy on mobile devices.

8.1.2 Overrun and Underrun Handling

Applications often dynamically change their resource demand due to the variation in the input data. Consequently, they may overrun or underrun their CPU allocation (i.e., need more or less than the allocated). An overrun application may miss its deadline or cause other applications to miss their deadlines, while an underrun often result in wasting of the CPU and energy resources. Different approaches have been proposed in the literature to handle overrun and underrun.

Gardner and Liu [37] proposed two approaches for handling overruns. The first approach, called Overrun Server Method (OSM), uses a sporadic server to schedule overrun parts of all applications. This method guarantees that applications which do not overrun meet deadlines, but cannot ensure when to finish the overrun part. The second approach, called the Isolation Server Method (ISM), handles overrun by sharing the budget within the same application.

Similarly, Abeni and Buttazzo [2] proposed a mechanism, called Constant Bandwidth Server (CBS), in which a CBS is used to schedule each individual application. If an application overruns, its deadline is postponed, thus being isolated from other applications. The CBS algorithm has been extended by several groups later. Lipari and Baruah [63, 64] proposed algorithms which enable

overrun applications to use the residual budget from underrun applications, thus reducing the deadline miss ratio. Caccamo et al. [20] proposed a capacity sharing for overrun control, called CASH. The CASH algorithm uses a global queue to store all residual budgets from underrun applications. When an application is executed, it first uses budgets from the CASH queue and then uses its own budget, thus implicitly controlling overruns. Similarly, Bavier and Peterson [14] proposed a budget borrowing mechanism for overrun control for multimedia applications. In this mechanism, when a multimedia application overruns, it borrows the budget form best-effort applications; when a multimedia application underruns, it then returns the budget to best-effort applications.

The above related work on overrun and underrun handling is orthogonal and complementary to the soft real-time scheduling in GRACE-OS. For example, GARCE-OS could use the CASH algorithm during the internal adaptation in operating system layer. GRACE-OS differs from the above work for three reasons. First, its soft real-time scheduling is integrated with the cross-layer adaptation. In particular, the CPU allocation is determined by the coordinator and the scheduler also dynamically adapts CPU allocation during runtime. Second, GRACE-OS uses a novel algorithm which allocates an additional budget to an overrun application by speeding up the CPU. Finally, GRACE-OS provides an opportunity to save energy, in addition to overrun control, through budget sharing. For example, GRACE-OS can relax the time constraint by adding residual time from other applications when calculating the speed schedule (Section 5.4).

8.2 QoS-Aware Application Adaptation

In mobile computing environments, system resources such as CPU time and network bandwidth are often limited and further change dynamically. As a result, applications need to adapt their QoS in many occasions, typically with the help of the operating system or middleware.

Blair et al. [17] proposed a reflective approach that provides support for QoS monitoring and adaptation in middleware platforms. In [30], the authors proposed a dynamic QoS meta-management solution for distributed multimedia systems. In this solution, the functions for QoS

provisioning are dynamically configurable and reusable. More recently, Gu and Klara [44] proposed a dynamic QoS-aware configuration service, which adapts different components of a distributed application and ensures the consistency of the configuration of these components. These previous approaches provide different mechanisms to decide how to adapt multimedia applications, and hence can be leveraged by GRACE-OS.

The Odyssey operating system [78] adds system support for mobile application adaptation, focusing on data fidelity and adaptation agility. During the adaptation, the operating system and application cooperate with each other: The operating system monitors resource availability and notifies the application upon resource changes, while the application decides how to adapt when notified. Agilos [61] is a middleware control architecture, which enforces the best possible adaptation decisions for distributed multimedia applications through dynamic controls and reconfigurations of their internal parameters and functionalities. Unlike Odyssey, Agilos decides how to adapt an application within the middleware.

In terms of support for QoS adaptation, GRACE-OS is more like Agilos in that the operating system is simply a mediator, while the application itself decides how to adapt without exposing its internals. GRACE-OS differs from Odyssey for two reasons: First, GRACE-OS coordinates the adaptation of multiple applications to maximize their total utility. Second, in addition to adapting applications, GRACE-OS also adapts the hardware resources (currently the CPU speed) at finer granularity.

8.3 Energy-Aware CPU Adaptation

Energy is a critical resource for battery-powered mobile devices. Recently, there has been a lot of related work on reducing energy for various components such as CPU [39, 43, 80, 102, 107, 81, 86, 83], network [6, 55, 56], disk [45, 50], memory [59], and display [53]. In this section, we focus on the related work on adapting the CPU for energy saving.

8.3.1 CPU Adaptation Mechanisms

In general, there are three adaptation approaches for saving CPU energy:

- **Architecture adaptation**. The CPU can resize its instruction window and active different number of functional units while execution applications [49, 92]. Architecture adaptation often happens at a very fine granularity (e.g., every few instructions). Currently, architecture adaptation is often simulated since most processors do not enable the software to control the architecture at runtime.

- **Dynamic power management (DPM)**. The CPU can operate at different states: active, idle, and sleep, where the sleep state consumes much less power. The DPM approach puts the CPU into the sleep state when the CPU is idle [15, 95].

  The DPM approach is not suitable for our targeted multimedia applications. The reason is that multimedia applications need to use the CPU periodically (e.g., every 30 milliseconds) and consequently the idle interval within the period is often much shorter than the overhead to put the CPU into and from the sleep state (e.g., it takes about 160 milliseconds for StrongARM SA-1100 to wake up from the sleep state [15]).

- **Dynamic frequency/ voltage scaling (DVS)**. The modern mobile processors such as Intel Pentium-M [51], AMD Athlon [5] and Transmeta Crusoe [98] can run at multiple speeds (frequencies and voltages), trading off performance for power. The DVS approach lowers the operating speed of the active CPU [9, 33, 43, 83, 102, 105, 107].

  DVS exploits two important characteristics in mobile systems: First, the application workload is dynamic; consequently, the CPU does not need to always run at the highest speed (performance). Second, mobile processors are often built on CMOS logic and their power consumption is dependent on the operating frequency and voltage. At a lower frequency, the CPU can operate at a lower voltage, thus reducing power.

8.3.2 Operating System Directed DVS

The major goal of DVS is to slow down the CPU by as much as possible, thus minimizing energy, while not affecting application performance. As a result, it is often the software, typically the operating system, that makes decisions on DVS.

Recently, DVS has been investigated in two main areas, general-purpose systems (GP-DVS) and real-time systems (RT-DVS). GP-DVS algorithms heuristically predict the workload based on the average CPU utilization in previous intervals [39, 43, 80, 102]. Although GP-DVS can save energy without significantly degrading performance of best-effort applications, it cannot be directly applied to multimedia applications due to the timing constraint and demand variations of multimedia applications. Grunwald et al. [43], for example, concluded that no heuristic algorithm they examined saves energy without affecting multimedia application performance.

RT-DVS algorithms, typically integrated with real-time CPU scheduling, derive workload from the worst-case CPU demand of real-time applications [9, 81, 83, 105]. Applications may, and often do, complete earlier before using up the worst-case allocation since they change CPU demand dynamically. To handle the runtime variations, some reclamation techniques have been proposed to reclaim the residual allocation to save more energy [9, 83]. These reclamation techniques first run the CPU fast by assuming the worst-case demand, and then slow down the CPU when an application completes earlier. Unlike GRACE-OS, the above RT-DVS algorithms do not consider the soft real-time nature and CPU usage patterns of multimedia applications, which provides more opportunities for energy saving.

Statistical DVS is an alternative approach to handling runtime variations of application CPU demand [42, 67, 95, 96]. Simunic et al. [95] and Sinha et al. [96] proposed algorithms that changes speed for each job of a task based on a stochastic model (e.g., Markov process) of the task's CPU demands. GRACE-OS differs from these two algorithms in that they changes speed only at the beginning of a job, while GRACE-OS uses intra-job DVS which dynamically changes speed within a job execution.

Some groups have also investigated on intra-job statistical DVS. Gruian [42] used statistical DVS for hard real-time systems. Lorch and Smith [67] used an algorithm, called PACE, to improve GP-DVS algorithms. The basic idea of these statistical DVS algorithms is similar to that in GRACE-OS: minimizing energy by adapting the execution speed based on the probability distribution of cycle demand of applications.

GRACE-OS differs from the above two stochastic DVS techniques for three reasons. First, GRACE-OS obtains the demand distribution via an automatic profiling and estimation, while the other two either assume a given distribution function. Second, GRACE-OS supports multiple applications by integrating soft real-time scheduling and DVS. In contrast, PACE supports only a single application and treats concurrent applications as a joint workload without isolation among them. Although Gruian's approach [42] claims to support concurrent applications for fixed-priority systems, it is not clear on how it decides the time allocation for multiple applications. Finally and more importantly, the other two present simulations only, while GRACE-OS implements the stochastic DVS. More recently, Lorch and Smith implemented the PACE algorithm in Windows 2000 [68]. Their implementation, however, does not support soft real-time scheduling.

8.3.3 Compiler Assisted DVS

Another related work is compiler-assisted adaptation for energy saving [3, 10]. Azevedo et al. [10] proposed an intra-task DVS technique under compiler control using program checkpoints. The compiler inserts checkpoints at the beginning of each branch, loop, function call, and normal segment. These checkpoints indicate places in the code where DVS should be invoked and further assist to estimate how many CPU cycles needed to for the remaining code.

More recently, AbouGhazaleh et al. [3] proposed an collaborative approach between the compiler and the operating system to save CPU energy. The compiler instruments application source code with path-dependent information, which captures the temporal behavior of the application at different paths. At runtime, this information is used by the operating system to dynamically change the CPU speed.

These compiler-assisted adaptation approaches are orthogonal and complementary to GRACE-OS. First, the profiler in GRACE-OS can use the annotation added by the compiler to estimate the cycle demand of an application more precisely (e.g., differentiating various frame types such as I, P, and B frames). Second, GRACE-OS can use the information provided by the compiler to perform adaptation at finer granularity (within a job).

8.3.4 DVS with Discrete Speeds

Previous DVS algorithms often assume an ideal processor that can change the speed continuously. In practice, however, mobile processors support a discrete set of speeds, rather than a continuous range. For example, the StrongARM SA-1110 CPU supports 11 different speeds, from 59 MHZ to 206 MHz in steps of 14.7 MHz.

Recently, much research effort has been made on handling the discrete speed options of the CPU. For example, Miyoshi et al. [73] empirically analyzed the runtime effect of DVS and found that different CPUs have different optimal speed levels. This work is orthogonal and complementary to GRACE-OS. Given the knowledge of the optimal CPU speeds, GRACE-OS can adapt the speed to minimize energy.

Other related work includes mapping the calculated speed to the speeds supported by the CPU. A simple approach is to round the calculated, optimal speed to the upper bound of the supported speeds [83, 107]. For example, if the CPU supports three speeds, 100, 200, and 300 MHz and the calculated speed is 210 MHz, the operating speed can be set to 300 MHz. This rounding-up typically will run the CPU at a speed higher than the demanded, thereby wasting energy.

An alternative approach is to emulate the calculated speed with two bounding supported speeds [42, 52, 67]. This approach distributes cycles that need to be executed at the calculated speed into two parts, one for the lower bound and the other for the upper bound. This emulation approach has been shown to be effective in simulations. It, however, may potentially result in large overhead when used in real implementations since it changes the speed more frequently.

Unlike the above mapping approaches, GRACE-OS explicitly considers the discrete speed

options when calculating the speed schedule for each individual application.

8.4 Energy-Aware Application Adaptation

Several projects advocate energy saving in the application layer. For example, the Milly Watt project [32] proposes that applications are the driving force for the higher-level power management and suggests a power-based API for the partnership between applications and the system in managing energy.

Recently, some groups have developed energy-aware adaptive applications [35, 27, 72, 91, 47]. Flinn et al. [35] developed a tool, called PowerScope, to profile energy usage by applications. Based on the profiling results, they further investigated how applications can dynamically adapt their behavior to save energy. Cornel et al. [27] developed a system, called Fugue, that consists of three separate controllers: transmission, video, and preference. This decomposition provides adaptation along different time scales: per-packet, per-frame, and per-video. Similarly, Mesarina et al. [72] and Sachs et al. [91] discussed how to reduce energy in MPEG decoding and H263 encoding, respectively. More recently, He et al. [47] proposed a metric, called Power-Rate-Distortion, to analyze wireless video encoding and transmission for energy saving.

All the above application-layer energy adaptation work is orthogonal and complementary to GRACE-OS. For example, when the battery runs out, GRACE-OS can notify these adaptive applications, so they can adapt their operation to reduce energy. Furthermore, GRACE-OS provides a mechanism to coordinate adaptations of various applications and the CPU hardware, potentially saving more energy.

8.5 Coordination of Adaptation

Given the adaptability of the hardware resources and multiple applications, it is necessary to coordinate their adaptation to achieve a system-wide optimization. Related work on coordination

of adaptation can be classified into two categories, *coordinated resource allocation* and *coordinated adaptation*. The former implicitly adapt multiple applications by controlling their resource allocation, while the latter explicitly controls the adaptation of multiple adaptive entities.

The work related to coordinated resource allocation includes follows. Q-RAM [89] allocates resources to multiple applications in a way that maximizes their total utility while guaranteeing minimum utility (and hence resources) to each application. Similar to GRACE-OS, Q-RAM proves that the constrained allocation problem is NP-hard and provides several heuristic algorithms. IRS [38] coordinates the allocation and scheduling of multiple resources to admit as many applications as possible. Unlike GRACE-OS, Q-RAM and IRS do not consider energy.

Recently, Park et al. [79] extended Q-RAM with the energy constraints. Similarly, ECOSystem [110] manages energy as a first class resource. It allocates energy to each individual application and seeks to achieve a desired battery lifetime. Rusu et al. [90] proposed two optimization algorithms that allocate CPU to multiple applications by considering the constraints of energy, deadline, and utility together.

All the above coordination approaches are similar to the global coordination in GRACE-OS in that all of them coordinate the resource allocation to multiple applications for a system-wide optimization. Unlike GRACE-OS, they do not perform internal adaptation in response to small changes at fine time granularity.

Recently, some groups have also been researching on the coordination of adaptation in different system layers. Efstratiou et al. [31] proposed a middleware platform that coordinates multiple adaptive applications for a system-wide objective. Q-fabric [85] supports the combination of application adaptation and distributed resource management via a set of kernel-level abstractions. HATS [58] adds control over bandwidth scheduling to the Puppeteer middleware [34] and coordinates adaptation of multiple applications to improve network performance. The above related work considers application adaptation only (with the support of resource management in the OS or middleware). In contrast, GRACE-OS considers cross-layer adaptation of the CPU frequency, operating system scheduling, and application QoS.

More recently, there is some work on cross-layer adaptation [74, 82, 87, 16]. Like GRACE-OS, TIMELY [16] also integrates and coordinates multiple-layer adaptation in the protocol stack, but focuses on the network bandwidth resource. PADS [87] is a framework for managing energy and QoS for distributed systems and focuses on the hardware and OS layers. Mohapatra et al. [74] proposed an approach that uses a middleware to coordinate the adaptation of hardware such as cache and application quality at coarse time granularity (e.g., at the time of admission control).

EQoS [82] is an energy-aware QoS adaptation framework. Like GRACE-OS, EQoS also formulates energy- and QoS-aware adaptation as a constrained optimization problem and uses heuristical algorithms to solve this problem. GRACE-OS differs from EQoS for two reasons: First, EQoS targets to hard real-time systems where the application set is typically static and requires worst-case guarantees. In contrast, GRACE-OS aims for multimedia-enabled mobile devices. The soft real-time nature of multimedia applications offer more opportunities for QoS and energy tradeoff; e.g., more energy can be saved via stochastic (as opposed to worst-case) QoS guarantees. Second, EQoS focuses on only global adaptation at coarse time granularity, while GRACE-OS uses both global and internal adaptation to handle changes at different time granularity. The global and internal adaptation hierarchy enables GRACE-OS to balance the benefits and cost of cross-layer adaptation, thus achieving the benefits with acceptable overhead.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this thesis, we have argued the needs for cross-layer adaptation to trade off multimedia quality against energy in multimedia-enabled mobile devices, such as camera phones. We have therefore presented the *GRACE-OS*, an energy-efficient mobile multimedia operating system to support the cross-layer adaptation in stand-alone mobile devices. The challenging problem, addressed in GRACE-OS, is as follows: given the adaptability (i.e., the ability to operate at multiple states) of multiple system layers, how to coordinate them for a system-wide optimization such as maximizing multimedia quality or achieving a desired lifetime.

GRACE-OS addresses the above problem by extending traditional scheduling with two additional dimensions, the quality level for multimedia tasks and the speed for the CPU. That is, the operating system decides (1) what quality level to assign for each tasks, (2) what CPU speed to execute tasks at, and (3) when to execute what tasks. GRACE-OS makes these decisions via three steps:

- First, when a task joins or leaves the system, GRACE-OS uses a global adaptation to coordinate all system layers to decide the *quality level* and CPU allocation for each of the concurrent tasks and the CPU speed and expected power consumption of the device. These global decisions seek to achieve a system-wide optimization based on the long-term prediction of the energy availability and CPU demand of individual tasks.

- Second, GRACE-OS uses an energy-aware real-time scheduling algorithm to enforce the globally coordinated decisions. The scheduler decides *when to execute what task* by assigning a cycle budget to individual tasks based on their coordinated allocation, dispatching the task with the earliest deadline and positive budget, and charing the budget of the executed task by the number of cycles it consumes.

- Third, GRACE-OS uses internal adaptation to handle small variations in the CPU usage of multimedia tasks due to the changes in their input data. The internal adaptation decides *what speed* to execute a task. The goal of the internal adaptation is to minimize the energy consumption while enabling each task to provide the coordinated quality.

The key contributions brought by GRACE-OS are as follows. First, with the global and internal adaptation hierarchy, we are now able to control and coordinate the adaptation in different system layers of mobile devices to trade off multimedia quality for energy. In particular, we balance the benefits and cost of the cross-layer adaptation, thus achieving a system-wide optimization with acceptable overhead. Second, previous real-time scheduling algorithms for QoS provisioning often assume a static processor. With our proposed speed-aware soft real-time scheduling algorithm, we are able to schedule applications predictably on a variable-speed processor. Furthermore, this scheduling algorithm also provides flexility to handle overruns and underruns by adapting the CPU speed while not affecting other tasks. Finally, with the kernel-based profiling technique in GRACE-OS, we are able to predict the CPU demand for individual applications. This prediction of CPU demand is necessary and important for both QoS provisioning and energy saving.

GRACE-OS has been implemented as a set of patches and modules in the Linux kernel 2.6.5 and evaluated with adaptive CPU and video codecs. Our experimental results show that although GRACE-OS employs heuristic algorithms in the global and internal adaptation, its cross-layer adaptation efficiently trades off multimedia quality for energy based on the user's preferences:

- Compared to previous systems that adapt only some of the three layers, GRACE-OS (1) improves the total utility by up to 69% or saves energy by 59% without affecting utility

when the user wants to maximize multimedia quality, and (2) achieves the desired lifetime while improving the utility by up to 45.8% when the user wants the battery to last for a desired lifetime.

- Compared to previous systems that adapt all three layers only at coarse time granularity, GRACE-OS saves energy by 2% to 8.9% without affecting multimedia quality.

9.2 Lessons Learned and Future Work

Although our current study on GRACE-OS yields strong results, lessons learned motivate the following future work:

1. Utility functions are a flexible tool to capture task adaptation behavior. The global adaptation in GRACE-OS is heavily dependent on how to define the utility function. However, utility definition is user-specific; e.g., different users may perceive different quality for the same task running at the same QoS level. Furthermore, it is often difficult to map the utility to system resource demands. In the future, we plan to investigate the utility definition with the help of objective or subjective assessment techniques [57, 69, 74] and map the user-level utility to the system-level demands with the support of Q-compiler [103].

2. The energy saving capability of GRACE-OS is limited by few CPU speed options. In particular, the processor often runs at a higher speed than the demanded, thus wasting energy. We expect that GRACE-OS will result in more benefits, if there are more speeds available and the speed adaptation incurs low overhead. In general, such expectation can be examined in three ways: (1) using a trace-based simulator to experiment with an ideal processor that supports continuous DVS, (2) applying GRACE-OS to processors that support continuous DVS (e.g., lpARM [81]), and (3) converting an optimal speed to two available speeds [42, 52, 67]. We plan to investigate the last approach, which would be another kind of internal adaptation in GRACE-OS.

In addition to the work motivated by the lessons, there exist open problems in the cross-layer adaptation and enhancement opportunities to GRACE-OS. We describe some of them as follows.

- GRACE-OS targets multimedia applications that process multimedia streams periodically and whose demand distribution is stable or changes slowly. Beyond periodic multimedia applications, we expect that GRACE-OS can also benefit best-effort applications such as web browsers. Furthermore, although developed for thin mobile devices, GRACE-OS may also apply to other platforms such as hosting servers, which need to save energy due to the environmental concerns (e.g., cooling overhead and noise). A careful examination of these open problems involves, e.g., how to model the constraints in the hardware platforms, how to model the adaptation behavior (such as the perceptual quality) of applications, how to predict the variations of the resource demand and availability in these systems.

- GRACE-OS currently changes the CPU speed during the execution of each individual task. We expect that sharing budget among different tasks would result in more energy saving by smoothing the CPU speed. For example, by sharing time among different tasks, we may find a better speed schedule to save more energy. An interesting future work is to calculate the speed schedule for concurrent tasks based on their aggregate demand distribution.

- GRACE-OS needs to be integrated with other components in the GRACE system, which are currently in development. First, we need to extend our current resource model to consider other system resources such as network bandwidth. This in turn will modify the algorithms for global adaptation, include the schedulers for other resources, and extend the adaptation hierarchy with other levels of adaptation such as per-application adaptation [91].

- GRACE-OS is currently developed for stand-alone mobile devices, but needs to be extended for distributed computing environments for two reasons. First, multimedia applications are often distributed, e.g., video streaming from a remote server. Second, multiple devices such as networked sensors often cooperate with each other. Mobile distributed systems introduce

two new problems: (1) how to save energy for mobile nodes by taking advantage of static nodes, and (2) how to save the overall energy for the whole system, such as a mobile ad hoc network. To address these problems, the cross-layer adaptation needs to be extended to *multi-node adaptation*. For example, a video server or proxy can adapt the quality of a video streamed to mobile devices based on their available resources. Consequently, GRACE-OS needs to be designed as an energy-aware distributed operating system, possibly combined with compilers and middleware, to support the multi-node, cross-layer adaptation.

In summary, the research areas of adaptation for QoS and energy continue to present new challenges and also offer new opportunities. Our achievements with GRACE-OS have led to a real cross-layer adaptive system, which can serve as the base for tackling the above problems.

References

[1] R. Aalmoes. Roalt's h.263 page. http://www.xs4all.nl/ roalt/h263.html, 2003.

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of 19th IEEE Real-Time Systems Symposium*, pages 4–13, Phoenix, AZ, Dec. 1998.

[3] N. AbouGhazaleh, D. Mosse, B. Childers, R. Melhem, and M. Craven. Collaborative operating system and compiler power management for real-time applications. In *Proceedings of 9th IEEE Real-Time Technology and Applications Symposium*, Washington, DC, May 2003.

[4] S. Adve et al. The Illinois GRACE Project: Global Resource Adaptation through CoopEration. In *Proceedings of Workshop on Self-Healing, Adaptive and self-MANaged Systems*, New York City, NY, June 2002.

[5] AMD. Mobile AMD Athlon 4 processor model 6 CPGA data sheet. http://www.amd.com, Nov. 2001.

[6] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning wireless network power management. In *Proceedings of International Conference on Mobile Computing and Networking*, San Diego, CA, Sept. 2003.

[7] J. M. Anderson et al. Continuous profiling: Where have all the cycles gone? In *Proceedings of 16th Symposium on Operating Systems Principles*, St-Malo, France, Oct. 1997.

[8] G. Anzinger et al. High resolution POSIX timers. http://high-res-timers.sourceforge.net/, 2004.

[9] H. Aydin, R. Melhem, D. Mosse, and P. Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of 22nd IEEE Real-Time Systems Symposium*, London, UK, Dec. 2001.

[10] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Proceedings of Design, Automation and Test in Europe Conference*, 2002 Mar.

[11] S. Banachowski and S. Brandt. The BEST scheduler for integrated processing of best-effort and soft real-time processes. In *Proceedings of SPIE Multimedia Computing and Networking Conference*, San Jose, CA, Jan. 2002.

[12] S. Banachowski, J. Wu, and S. Brandt. Missed deadline notification in best-effort schedulers. In *Proceedings of Multimedia Computing and Networking Conference*, San Jose, CA, Jan. 2004.

[13] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of 3rd Symposium on Operating System Design and Implementation*, New Orleans, LA, Feb. 1999.

[14] A. Bavier and L. Peterson. The power of virtual time for multimedia scheduling. In *Proceedings of 10th International Workshop for Network and Operating System Support for Digital Audio and Video*, Chapel Hill, NC, June 2000.

[15] L. Benini, A. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3), June 2000.

[16] V. Bharghavan, K. Lee, S. Lu, S. Ha, J. Li, and D. Dwyer. The TIMELY adaptive resource management architecture. *IEEE Personal Communications Magazine*, 5(4), Aug. 1998.

[17] G. Blair, A. Andersen, L. Blair, and G. Coulson. The role of reflection in supporting dynamic qos management functions. In *Proceedings of 7th IEEE International Workshop on Quality of Service*, London, UK, June 1999.

[18] S. Brandt, G. Nutt, T. Berk, and J. Mankovich. A dynamic quality of service middleware agent for mediating application resource usage. In *Proceedings of 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 1998.

[19] S. Brandt and G. J. Nutt. Flexible soft real-time processing in middleware. *Real-Time Systems*, 22(1-2), 2002.

[20] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of 21th IEEE Real-Time Systems Symposium*, Orlando, FL, Dec. 2000.

[21] M. Caccamo, G. Buttazzo, and L. Sha. Handling execution overruns in hard real-time control systems. *IEEE Transactions on Computers*, 54(7), July 2002.

[22] B. M. R. Center. Berkeley mpeg tools. http://bmrc.berkeley.edu/frame/research/mpeg/, 2001.

[23] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Proceedings of 4th Symposium on Operating System Design and Implementation*, San Diego, CA, Oct. 2000.

[24] A. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27:473–484, Apr. 1992.

[25] H. H. Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pages 296–301, Florence, Italy, June 1999.

[26] Compaq, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification. http://www.teleport.com/acpi/spec.htm, July 2000.

[27] M. Corner, B. Noble, and K. Wasserman. Fugue: time scales of adaptation in mobile video. In *Proceedings of SPIE Multimedia Computing and Networking Conference*, San Jose, CA, Jan. 2001.

[28] Z. Deng and J. S. Liu. Scheduling of real-time applications in an open environment. In *Proceedings of 18th IEEE Real-time Systems Symposium*, San Francisco, CA, Dec. 1997.

[29] K. Duda and D. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general purpose scheduler. In *Proceedings of 17th Symposium on Operating Systems Principles*, Charleston, SC, Dec. 1999.

[30] D. Ecklund, V. Goebel, T. Plagemann, E. Ecklund, C. Griwodz, J. Aagedal, K. Lund, and A.-J. Berre. Qos management middleware - a seperable, reuable solutio. In *Proceedings of 8th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, Lancaster, UK, Sept. 2001.

[31] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst. A platform supporting coordinated adaptation in mobile systems. In *Proceedings of 4th IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, NY, June 2003.

[32] C. Ellis. The case for higher-level power management. In *Proceedings of 7th IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, Mar. 1999.

[33] K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for linux. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[34] J. Flinn, E. de Lara, M. Satyanarayanan, D. Wallach, and W. Zwaenepoel. Reducing the energy usage of office applications. In *Proceedings of Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[35] J. Flinn and M. Satyanarayanan. PowerScope: A tool for proling the energy usage of mobile applications. In *Proceedings of 2nd IEEE Workshop on Mobile Computing Systems and Applications*, New Orleans, LA, Feb. 1999.

[36] K. Gardner. Probabilistic analysis and scheduling of critical soft real-time systems. PhD thesis, Dept of Computer Science, Univ of Illinois at Urbana-Champaign, 1999.

[37] M. Gardner and J. S. Liu. Performance of algorithms for scheduling real-time systems with overrun and overload. In *Proceedings of 11th Euromicro Conference on Real-Time Systems*, pages 9–11, York, UK, June 1999.

[38] K. Gopalan and T. Chiueh. Multi-resource allocation and scheduling for periodic soft real-time applications. In *Proceedings of SPIE Multimedia Computing and Networking Conference*, San Jose, CA, Jan. 2002.

[39] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power CPU. In *Proceedings of Annual international Conference on Mobile Computing and Networking*, Berkeley, CA, Nov. 1995.

[40] P. Goyal, X. Guo, and H. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of Symposium on Operating System Design and Implementation*, Seattle, WA, Oct. 1996.

[41] V. Grassi and R. Mirandola. Derivation of markov models for effectiveness analysis of adaptable software architectures for mobile computing. *IEEE Transactions on Mobile Computing*, 2(2), June 2003.

[42] F. Gruian. Hard real-time scheduling for low energy using stochastic data and DVS processors. In *Proceedings of International Symposium on Low-Power Electronics and Design*, Huntington Beach, CA, Aug. 2001.

[43] D. Grunwald, P. Levis, K. Farkas, C. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of Symposium on Operating System Design and Implementation*, San Diego, CA, Oct. 2000.

[44] X. Gu and K. Nahrstedt. Dynamic QoS-aware multimedia service configuration in ubiquitous computing environments. In *Proceedings of IEEE 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[45] S. Gurumurthi, A. Sivasubramaniam, and M. Kandemir. DRPM: Dynamic speed control for power management in server class disks. In *Proceedings of 30th Annual International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[46] C. Hamann et al. Quality-assuring scheduling-using stochastic behavior to improve resource utilization. In *Proceedings of 22nd IEEE Real-Time Systems Symposium*, London, UK, Dec. 2001.

[47] Z. He, Y. Liang, L. Chen, I. Ahmad, and D. Wu. Power-rate-distortion analysis for wireless video communication under energy constraints. *IEEE Transactions on Circuits and Systems for Video Technology, Special Issue on Integrated Multimedia Platforms*, May 2004.

[48] C. Hughes, P. Kaul, S. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *Proceedings of International Symposium on Computer Architecture*, Goteborg, Sweden, 2001.

[49] C. Hughes, J. Srinivasan, and S. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of 34th International Symposium on Microarchitecture*, Austin, TX, Dec. 2001.

[50] C.-H. Hwang and A. Wu. A predictive system shutdown method for energy saving of event-driven computation. In *Proceedings of International Conference on Computer Aided Design*, Nov. 1997.

[51] Intel. Intel Pentium M Processor. http://developer.intel.com/design/mobile/datashts/25261203.pdf, Apr. 2004.

[52] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of International Symposium on Low-Power Electronics and Design*, Monterey, CA, 1998.

[53] S. Iyer, L. Luo, R. Mayo, and P. Ranganathan. Energy-adaptive display system designs for future mobile environments. In *Proceedings of International Conference on Mobile Systems, Applications, and Services*, San Francisco, CA, May 2003.

[54] M. Jones, D. Rosu, and M. Rosu. CPU reservations & time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of 16th Symposium on Operating Systems Principles*, St-Malo, France, Oct. 1997.

[55] R. Krashinsky and H. Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *Proceedings of 8th ACM International Conference on Mobile Computing and Networking*, Atlanta, GA, Sept. 2002.

[56] R. Kravets and P. Krishnan. Power management techniques for mobile communication. In *Proceedings of 4th ACM International Conference on Mobile Computing and Networking*, Dallas, TX, 1998.

[57] T. Kunz, M. Shentenawy, A. Gaddah, and R. Hafez. Image transcoding for wireless WWW access: The user perspective. In *Proceedings of SPIE Multimedia Computing and Networking*, San Jose, CA, Jan. 2002.

[58] E. Lara, D. Wallach, and W. Zwaenepoel. HATS: hierarchical adaptive transmission scheduling for multi-application adaptation. In *Proceedings of SPIE Multimedia Computing and Networking Conference*, San Jose, CA, Jan. 2002.

[59] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Nov. 2000.

[60] P. Levis et al. The emergence of networking abstractions and techniques in TinyOS. In *Proceedings of First Symposium on Networked System Designe and Implementation*, San Francisco, CA, Mar. 2004.

[61] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE J. Select. Areas Commun.*, 17(9):1632–1650, Sept. 1999.

[62] R. Liao and A. Campbell. A utility-based approach for quantitative adaptation in wireless packet networks. *Wireless Networks*, 7(5), Sept. 2001.

[63] G. Lipari and S. Baruah. Greedy reclaimation of unused bandwidth in constant bandwidth servers. In *Proceedings of 12th Euromicro Conference on Real-Time Systems*, Stokholm, Sweden, June 2000.

[64] G. Lipari and S. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings of 7th IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, May 2001.

[65] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, Jan. 1973.

[66] J. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[67] J. Lorch and A. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of ACM SIGMETRICS 2001 Conference*, Cambridge, MA, June 2001.

[68] J. Lorch and A. Smith. Operating system modifications for task-based speed and voltage scheduling. In *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services*, San Francisco, CA, May 2003.

[69] A. Mayache, T. Eude, and H. Cherifi. A comparison of image quality models and metrics based on human visual sensivity. In *Proceedings of IEEE International Conference on Image Processing*, Bacerlona, Spain, Oct. 1998.

[70] R. Melhem, N. AbouGhazaleh, H. Aydin, and D. Mosse. Power management points in power-aware real-time systems. In R. Graybill and R. Melhem, editors, *Power Aware Computing*. Plenum/Kluwer Publisher, 2002.

[71] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems (ICMCS'94)*, May 1994.

[72] M. Mesarina and Y. Turner. Reduced energy decoding of MPEG streams. In *Proceedings of SPIE Multimedia Computing and Networking Conference*, San Jose, CA, Jan. 2002.

[73] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of 16th Annual ACM International Conference on Supercomputing*, New York City, NY, June 2002.

[74] S. Mohapatra, R. Cornea, N. Dutt, A. Nicolau, and N. Venkatasubramanian. Integrated power management for video streaming to mobile devices. In *Proceedings of ACM Multimedia*, Berkeley, CA, Nov. 2003.

[75] S. Mohapatra and N. Venkatasubtramanian. Power-aware reconfigure middleware. In *Proceedings of IEEE 23nd International Conference on Distributed Computing Systems*, Providence, RI, May 2003.

[76] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of 16th Symposium on Operating Systems Principles*, St-Malo, France, Oct. 1997.

[77] D. Niz, L. Abeni, S. Saewong, and R. Rajkumar. Resource sharing in reservation-based systems. In *Proceedings of 22nd IEEE Real-Time Systems Symposium*, London, UK, Dec. 2001.

[78] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *Proceedings of 16th Symposium on Operating Systems Principles*, Saint Malo, France, Dec. 1997.

[79] S. Park, V. Raghunathan, and M. Srivastava. Energy efficiency and fairness tradeoffs in multi-resource, multi-tasking embedded systems. In *Proceedings of International Symposium on Low Power Electronics and Design*, Seoul, Korea, Aug. 2003.

[80] T. Pering, T.Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of International Symposium on Low Power Electronics and Design*, Monterey, CA, June 1998.

[81] T. Pering, T.Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of International Symposium on Low Power Electronics and Design*, Rapallo, Italy, July 2000.

[82] P. Pillai, H. Huang, and K. G. Shin. Energy-aware quality of service adaptation. Technical report CSE-TR-479-03, University of Michigan, 2003.

[83] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of 18th Symposium on Operating Systems Principles*, Banff, Canada, Oct. 2001.

[84] D. Pisinger. A minimal algorithm for the multiple-choice Knapsack problem. *European Journal of Operational Research, 83*, pages 394–410, 1995.

[85] C. Poellabauer, H. Abbasi, and K. Schwan. Cooperative run-time management of adaptive

applications and distributed resources. In *Proceedings of 10th ACM Multimedia Conference*, Juan Les Pins, France, Dec. 2002.

[86] J. Pouwelse, K. Langendoen, and H. Sips. Energy priority scheduling for variable voltage processors. In *Proceedings of International Symposium on Low Power Electronics and Design*, Huntington beach, CA, Aug. 2001.

[87] V. Raghunathan, P. Spanos, and M. Srivastava. Adaptive power-fidelity in energy aware wireless embedded systems. In *Proceedings of IEEE Real Time Systems Symposium*, London, UK, Dec. 2001.

[88] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proceedings of SPIE Multimedia Computing and Networking Conference*, Jan. 1998.

[89] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. In *Proceedings of 18th IEEE Real-Time Systems Symposium*, San Francisco, CA, Dec. 1997.

[90] C. Rusu, R. Melhem, and D. Mosse. Maximizing the system value while satisfying time and energy constraints. In *Proceedings of 23rd Real-Time Systems Symposium*, Austin, TX, Dec. 2002.

[91] D. Sachs, S. Adve, and D. Jones. Cross-layer adaptive video coding to reduce energy on general-purpose processors. In *Proceedings of IEEE International Conference on Image Processing*, Barcelona, Spain, Sept. 2003.

[92] R. Sasanka, C. J. Hughes, and S. V. Adve. Joint local and global hardware adaptations for energy. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems*, San Jose, CA, Oct. 2002.

[93] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. FAST: Frequency-aware static timing analysis. In *Proceedings of 24th IEEE Real Time Systems Symposium*, Cancun, Mexico, Dec. 2003.

[94] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchroniztion. *IEEE trans. on computers*, 39(9), Sept. 1990.

[95] T. Simunic et al. Dynamic voltage scaling and power management for portable systems. In *Proceedings of Design Automation Conference*, Las Vegas, CA, June 2001.

[96] A. Sinha and A. Chandrakasan. Dynamic voltage scheduling using adaptive filtering of workload traces. In *Proceedings of 4th International Conference on VLSI Design*, Bangalore, India, Jan. 2001.

[97] J. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, and S. Son. Feedback control scheduling in distributed systems. In *Proceedings of 22nd IEEE Real-Time Systems Symposium*, London, UK, Dec. 2001.

[98] Transmeta. Crusoe processor model TM5600 features. http://www.transmeta.com/crusoe/download/pdf/ TM5600_ProductBrief_8-2-00.pdf, 2000.

[99] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[100] V. Vardhan. Cost of high resolution timer in the linux kernel. Personal communication, 2004.

[101] S. Wang, D. Xuan, R. Bettati, and W. Zhao. Differentiated services with statistical real-time guarantees in static-priority scheduling networks. In *Proceedings of 22nd IEEE Real-Time Systems Symposium*, London, UK, Dec. 2001.

[102] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of Symposium on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[103] D. Wichadakul. Q-compiler: Meta-data qos-aware programming and compilation framework. PhD thesis, Dept of Computer Science, University of Illinois at Urbana-Champaign, 2003.

[104] L. Yan, J. Luo, and N. Jha. Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. In *International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2003.

[105] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of 36th Annual Symposium on Foundations of Computer Science*, Milwaukee, WI, Oct. 1995.

[106] W. Yuan and K. Nahrstedt. Integration of dynamic voltage scaling and soft real-time scheduling for open mobile systems. In *Proceedings of 12th International Workshop on Network and OS Support for Digital Audio and Video*, Miami Beach, FL, May 2002.

[107] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of 19th Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.

[108] W. Yuan and K. Nahrstedt. Practical voltage scaling for mobile multimedia devices. In *Proceedings of ACM Multimedia*, New York, NY, Oct. 2004.

[109] W. Yuan, K. Nahrstedt, S. Adve, D. Jones, and R. Kravets. Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems. In *Proceedings of Multimedia Computing and Networking Conference*, San Jose, CA, Jan. 2003.

[110] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Proceedings of 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2002.

[111] X. Zhang, Z. Wang, N. Gloy, J. Chen, and M. Smith. System support for automated profiling and optimization. In *Proceedings of Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.

Vita

Wanghong Yuan was born in a small village in Jiangxi Province, China. He received his Bachelor of Science and Master of Science degrees in Computer Science from Beijing University in 1996 and 1999, respectively. Since August 1999, he has been a Ph.D. student in Computer Science at the University of Illinois at Urbana-Champaign. He has worked on the DSRT (Dynamic Soft Real-Time) and GRACE (Global Resource Adaptation through CoopEration) projects during his PhD study. He worked as a research Intern in Microsoft Research in 2000 and in Microsoft Research Asia in 2001. After graduation, he will join DoCoMo Communications Laboratories USA as a research engineer. His research interests are operating systems, networking, multimedia, and real-time systems, with an emphasis on energy-efficient and quality-aware operating system and network protocols for mobile computing.