Energy-Efficient, Utility Accrual Real-Time Scheduling

Haisang Wu

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

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Peter M. Athanas
E. Douglas Jensen
Tom Martin
Amitabh Mishra
Subhash C. Sarin

July 29, 2005
Blacksburg, Virginia

Keywords: Real-Time Scheduling, Energy-Efficient Scheduling, Time/Utility Functions,
Utility Accrual Scheduling, Dynamic Voltage Scaling, Overload Scheduling

Energy-Efficient, Utility Accrual Real-Time Scheduling

Haisang Wu

(ABSTRACT)

In this dissertation, we consider timeliness and energy optimization in battery-powered, mobile embedded real-time systems. We focus on real-time systems that operate in environments with dynamically uncertain properties, including context-dependent activity execution times and arbitrary activity arrival patterns. We consider an application model where activities are subject to time/utility function (or TUF) time constraints, mutual exclusion constraints on concurrent sharing of non-CPU resources, timeliness requirements including assurances on individual activity timeliness behavior, and system-level energy consumption requirements including a non-exhaustable energy budget.

To account for uncertainties in activity properties in dynamic systems, we stochastically describe activity execution demands, and describe activity arrival behaviors using the unimodal arbitrary arrival model, which allows unbounded arrival frequencies. We consider the scheduling optimality criteria of: (1) probabilistically satisfying lower bounds on individual activities' maximal timeliness utilities, and (2) maximizing system-level energy efficiency, while ensuring that the system's energy consumption never exhausts the energy budget and resource mutual exclusion constraints are satisfied.

For this multi-criteria scheduling problem, we present a DVS (dynamic voltage scaling)-based, real-time scheduling algorithm called the *Energy-Bounded Utility Accrual Algorithm* (or EBUA). Since the scheduling problem is $\mathcal{NP}$-hard, EBUA heuristically (and dynamically) allocates CPU cycles to activities, computes activity schedules, and scales CPU voltage and frequency with a polynomial-time cost. If activities' cumulative execution demands exceed the available CPU time or may exhaust the system's energy budget, the algorithm defers and rejects jobs in a controlled fashion, minimizing system-level energy consumption and

maximizing total accrued utility.

We analytically establish several properties of EBUA. We prove that the algorithm never exhausts the specified energy budget. Further, we establish EBUA's timeliness optimality during under-loads, freedom from deadlocks, and correctness in mutually exclusive resource sharing. In particular, we prove that the algorithm's timeliness behavior subsumes the optimal timeliness behavior of deadline scheduling as a special case, and identify the conditions under which lower bounds on individual activity utilities are satisfied. In addition, we upper bound the time needed for mutually exclusively accessing shared resources under EBUA.

We conduct experimental studies by simulating the algorithm on the DVS-enabled AMD k6 processor model, and by implementing it on QNX Neutrino 6.2.1 RTOS. Our experimental results validate our analytical results. Further, they confirm EBUA's superiority over other energy-efficient real-time scheduling algorithms on timeliness and energy consumption behaviors.

This work was supported by the U.S. Office of Naval Research (ONR) under Grant N00014-00-1-0549 and by The MITRE Corporation under Grant 52917. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ONR and MITRE.

To my parents and Feihui

Acknowledgments

First, I want to thank my advisor, Dr. Binoy Ravindran, for his support in every aspect of my three-year stay at Virginia Tech. He is always there to discuss with me, to guide me, to help me, and to cheer me up, with his encouragement, enthusiasm, and patience, whenever I have doubts about myself. Without his advising and support, little can I achieve in the past three years. I have learnt an enormous amount from working with him, and have thoroughly enjoyed doing so throughout my stay at VT.

Many thanks to Dr. E. Douglas Jensen, for giving me the opportunity to collaborate with him, and for his tremendous help and support. My work has benefited tremendously from lively discussions with him, through e-mails, phone calls, and face-to-face meetings. I am honored to have the opportunity of working with Dr. Jensen, the original pioneer of time/utility functions and utility accrual scheduling. I am also grateful to him for arranging financial support for me through his research project at MITRE.

Thanks, too, to the rest of my committee: Dr. Peter M. Athanas, Dr. Tom Martin, Dr. Amitabh Mishra, and Dr. Subhash C. Sarin. I greatly appreciate their being supportive and encouraging to my work, and also their willingness to bend their schedules to accommodate the meetings and exams. I am also grateful to Dr. Raymond Clark at MITRE and Dr. Gérard Le Lann at INRIA for the helpful discussions on this problem and possible directions.

In addition, I would like to thank all my colleagues over the past few years who have helped

me become a better researcher, and contributed to components of my dissertation research. The list would definitely include the following people: Peng Li, Hyeonjoong Cho, Karthik Channakeshava, Jinggang Wang, Umut Balli, Michelle Gong, and Shahrooz Feizabadi. I am also grateful to the staff of the ECE department at Virginia Tech for helping me in innumerable ways and for being a pleasure to work with. Life would be much more difficult (and no fun) without the support and companionship of many good friends to make my stay at VT a very memorable one.

Last, but not least, I would like to thank my family members for all the love and support that they are providing me. I am grateful to my parents for encouraging my education from a young age, and allowing my inquisitive nature to blossom. Without their love and constant encouragement, none of this would have been possible. My grandparents brought me up and always wished me the best with their unconditional love. My sister and my brother in-law helped me a lot on almost every aspect of my journey. Finally, I am forever indebted to my wife Feihui Li: she has been incredibly supportive of me through the ups and downs over the past years. I cannot thank her enough for being such a wonderful wife and friend.

This dissertation is to all people who have helped and are helping me. I deeply appreciate all their help.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

With the proliferation of mobile and embedded devices that operate on limited battery power, power and energy management of embedded systems have become critically important. Most such devices have finite energy bounds, embodied by a battery that has a finite lifetime. For mobile and portable embedded systems, minimizing energy consumption results in longer battery life. But intelligent devices usually need powerful processors, which consume more energy than those in simpler devices, thus reducing battery life. This fundamental trade-off between performance and battery life is critically important and has been addressed in the past [1, 2].

Saving energy without substantially affecting application performance is crucial for embedded real-time systems that are mobile and battery-powered, because most real-time and embedded applications running on energy-limited systems inherently impose temporal constraints on activity sojourn times [3].

Dynamic voltage scaling (DVS) is a common mechanism studied in the past to save CPU energy [4, 5, 6, 7, 8, 9, 10, 3, 11, 12, 13, 14]. DVS addresses the trade-off between performance and battery life by taking into account two important characteristics of most current

computer systems: (1) For CMOS-based processors, the maximum clock frequency scales almost linearly with the power supply voltage, and the energy consumed per cycle is proportional to the square of the voltage ($\propto V^2$) [15]; and (2) the peak computing rate needed is much higher than the average throughput that must be sustained. A lower frequency (i.e., speed) hence enables a lower voltage and yields a quadratic energy reduction, at the expense of roughly linearly increased sojourn time [16].

Most of the past efforts on energy-efficient real-time scheduling focus on the deadline time constraint and deadline-based timeliness optimality criteria, such as meeting all or some percentage of deadlines. Further, they consider periodic, or frame-based (where all periods are equal), or sporadic task arrival models. Furthermore, most past efforts focus on resource-independent activities—i.e., activities that do not access shared resources, which are subject to mutual exclusion constraints. For the optimality criterion of meeting all deadlines, past DVS schemes focus on minimizing energy consumption *of the CPU*, while meeting the deadlines of all (independent) activities.

1.1 Soft Timeliness Optimality

In this dissertation, we consider dynamic, embedded real-time systems in domains including robotics, space, defense, and consumer electronics. A specific motivating example from the robotics/space domain is NASA Jet Propulsion Laboratory's robotic systems (e.g., Mars Rover), which are envisioned for long-lived, scientific exploration missions on the Mars planet [17].

Such systems are time-critical, as they must sense external objects in a timely manner and must produce timely control responses—e.g., to avoid obstacles in the physical world. Further, they are energy-critical, as they operate on batteries, and are often subject to a

finite energy budget for a mission's duration. This is typically due to the non-availability of battery recharging time and/or energy source. Hence, they must operate without violating their energy budgets. Doing so, and prolonging the battery life requires bounding and minimizing the *system's* energy consumption, and not just that of the CPU's consumption.

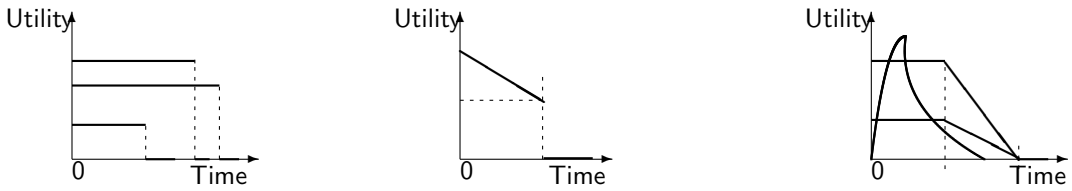
Moreover, these embedded systems are fundamentally distinguished by the fact that they operate in environments with dynamically uncertain properties. These uncertainties include transient and sustained resource overloads due to context-dependent activity execution times and arbitrary activity arrival patterns. Nevertheless, such systems' desire the strongest possible assurances on activity timeliness behavior. Consequently, their non-deterministic operating situations must be characterized with stochastic or extensional (rule-based) models. Another important distinguishing feature of these systems is their relatively long execution time magnitudes—e.g., in the order of milliseconds to seconds, or seconds to minutes.

When resource overloads occur, meeting deadlines of all application activities is impossible as the demand exceeds the supply. The urgency of an activity is typically orthogonal to the relative importance of the activity. For example, the most urgent activity can be the least important, and vice versa; the most urgent can be the most important, and vice versa. Hence when overloads occur, completing the most important activities irrespective of activity urgency is often desirable. Thus, a clear distinction has to be made between the urgency and the importance of activities, during overloads. During under-loads, such a distinction need not be made, because deadline-based scheduling algorithms such as Earliest Deadline First (EDF) are optimal for those situations [18]—they can satisfy all deadlines.

Deadlines by themselves cannot express both urgency and importance. Thus, we consider the abstraction of time/utility functions (or TUFs) [19] that express the utility of completing an application activity as a function of that activity's completion time. We specify the deadline as a binary-valued, downward "step" shaped TUF; Figure 1.1(a) shows examples. Note that a TUF decouples importance and urgency—i.e., urgency is measured as a deadline on the

x-axis, and importance is denoted by the utility on the y-axis.

Many embedded real-time systems also have activities that are subject to *non-deadline* and *soft* time constraints (besides hard), such as those where completing an activity at any time will result in some (positive or negative) utility to the system, and that utility depends on the activity's completion time. This is in contrast to deadlines, where a positive utility (also known as value) is accrued for completing the activity anytime before the deadline, after which zero or infinitively negative utility is accrued. These soft time constraints are subject to optimality criteria such as completing all time-constrained activities as close as possible to their optimal completion times—so as to yield maximal collective utility.



(a) Some Step TUFs (b) MITRE/TOG AWACS TUF [20] (c) GD/CMU BM/C2 TUFs [21]

Figure 1.1: Example TUF Time Constraints

The soft time constraints are generally at least as mission- and safety-critical as the hard time constraints. Figures 1.1(b)–1.1(c) show examples of such time constraints from two real applications in the defense domain: (1) an Airborne Warning And Control System (AWACS) built by The MITRE Corporation and The Open Group (TOG) [20] and (2) a battle management (BM)/command and control (C2) application built by General Dynamics (GD) and Carnegie Mellon University (CMU) [21]. These applications are discussed in greater detail in Chapter 2.

When activity time constraints are specified using TUFs, which subsume deadlines, the scheduling criteria are based on accrued utility, such as maximizing the sum, or the expected sum, of the activities' attained utilities, or assuring satisfaction of lower bounds on activities'

maximal utilities. Such criteria are called Utility Accrual (or UA) criteria, and sequencing (scheduling, dispatching) algorithms that consider UA criteria are called UA sequencing algorithms. In general, other factors may also be included in the optimality criteria, such as resource dependencies and precedence constraints. Several UA scheduling algorithms are presented in the literature [22, 23, 24, 25, 26, 27].

UA algorithms that maximize summed utility under downward step TUFs (or deadlines) [23, 24, 28] default to EDF during under-loads, since EDF can satisfy all deadlines during those situations. Consequently, they obtain the maximum total utility during under-loads. When overloads occur, they favor activities that are more important (since more utility can be attained from them), irrespective of their urgency. Thus, UA algorithms' timeliness behaviors subsume the optimal timeliness behavior of deadline scheduling.

1.2 System-Level Energy Consumption

Most of the past work on energy-efficient real-time scheduling using DVS only considers the energy consumed by the CPU[1]. However, the battery life of a system is determined by the *system's* energy consumption, not just the energy consumption of the CPU. Thus, energy consumption models used in past efforts are not accurate for prolonging the battery life.

Based on the experimental observations that some components in computer systems consume constant energy and some consume energy only scalable to frequency (i.e., voltage), Martin presents a *system-level* energy consumption model in [30, 31]. In this model, the system-level energy consumption per cycle does not scale quadratically to the CPU frequency. Instead, a polynomial is used to represent the relation. We further elaborate on this energy model in Section 3.6 (Page 22).

[1]The PA-BTA algorithm [29] considers system-level energy consumption, but it is restricted to independent activities and provides no assurances—individual or collective—on timeliness behavior.

1.3 Arbitrary Activity Arrival Behaviors

Most past efforts on energy-efficient, real-time scheduling consider activity arrival models that are either periodic, or frame-based (where all periods are equal), or sporadic. These include past works that consider deadline-based timeliness optimality criteria (e.g., meeting all or some percentage of deadlines) [11, 7, 3, 12, 13], and those that consider UA criteria (e.g., maximizing summed utility) [14, 29, 32, 33]. As far as we know, the only exception is [29], which allows aperiodic arrivals. However, [29] provides no timeliness assurances. Thus, prior efforts are concentrated on two extremes: (1) those that provide timeliness assurances, but under highly restrictive periodic, frame-based, or sporadic arrivals; or (2) those that allow aperiodic arrivals, but provide no timeliness assurances. Both extremes are inappropriate for the applications/domains of interest to us.

In this dissertation, we bridge these extremes by considering the unimodal arbitrary arrival model (or UAM) [34]. UAM allows unbounded arrival frequencies, embodies a "stronger" adversary than most traditional arrival models (e.g., periodic, frame-based, sporadic), and subsumes those models as special cases.

1.4 Non-Exhaustable Energy Bounds

As mentioned previously, we consider energy-critical applications, which are subject to a finite energy budget for the entire duration of their operational/mission life. Example motivations for such a constraint include the non-availability of battery recharging time and/or energy source. Consequently, they must operate without violating their energy budgets. Thus, the energy budget is a *hard* constraint.

If the system's energy budget does not (transiently or permanently) allow the execution of all jobs—e.g., due to (transient or permanent) overloads, some jobs should be deferred or

rejected in a controlled fashion so as to enable maximal utility to be accrued by the mission, without adversely affecting the system's functionality during the mission. Further, assurances on activity timeliness behavior must be provided, whenever possible. Furthermore, the system's energy consumption must be minimized and timeliness performance must be maximized, as much as possible, at all times.

Energy-bounded real-time scheduling algorithms are proposed for real-time systems with fixed energy budgets. But most of the past efforts are restricted to deadlines, step TUFs, and deadline-based timeliness optimality. Examples include [35] (which optimizes total "reward" for step reward functions, where reward is equivalent to our utility notion), [36], and [37]. On the other hand, energy-efficient real-time scheduling works, which do not allow energy budgets include those that consider deadlines and deadline-based optimality [16, 7, 3], and those that consider non-step TUFs and UA optimality [29, 32, 33, 38].

Therefore, from the energy budget standpoint, past works are concentrated on two extremes: (1) those that satisfy energy budgets, but for deadlines and deadline-based optimality; or (2) those that allow non step TUFs and UA optimality, but do not satisfy energy budgets. In this dissertation, we bridge these extremes by satisfying energy budgets under UA criteria.

1.5 Contributions

Most of the past efforts on energy-efficient real-time scheduling focus on the deadline time constraint and deadline-based timeliness optimality criteria (e.g., meeting all or some percentage of deadlines [16, 7, 3, 29]), resource-independent activities—i.e., activities that do not share non-CPU resources which are subject to mutual exclusion constraints, and minimizing only the CPU's energy consumption (as mentioned before). Exceptions include [14, 29, 39].

The work in [14] considers the optimality criterion of maximizing collective value, where

value is equivalent to our utility notion. However, [14] is restricted to step value functions or step TUFs (see Figure 1.1(a)). The work in [29] considers non-step TUFs, but it is restricted to resource-independent activities. The work in [39] considers voltage scheduling for periodic real-time tasks with non-preemptive blocking sections. However, [39] is restricted to deadlines and deadline-based timeliness optimality. The work in [29] considers system-level energy consumption, but it is restricted to resource-independent activities.

Resource dependencies are important for many embedded systems, as many such systems use shared resources and simultaneously access them for application progress [40]. UA scheduling under resource dependencies has been studied in the past [24, 22]. But energy-efficient UA scheduling has not been studied prior to this dissertation. Further, all past UA scheduling algorithms maximize the collective utility attained by all activities. They provide no assurance on individual timeliness behavior such as a lower bound on individual activity utility that is probabilistically satisfied.

In this dissertation, we consider timeliness and energy optimization in dynamic real-time embedded systems with the previously mentioned characteristics. In particular, we focus on the scheduling problem that intersects the following individual scheduling problems: (1) CPU scheduling under TUF time constraints to optimize UA optimality criteria including providing assurances on individual activity timeliness behavior, while satisfying mutual exclusion resource constraints; (2) DVS-based CPU scheduling for bounded and reduced system-level energy consumption; and (3) CPU scheduling under the UAM for activity arrival behaviors. This overlapping scheduling problem has not been studied in the past (though each of the individual scheduling problems has been studied) and is the focus of this dissertation.

We consider an application model where activities are subject to TUF time constraints, mutual exclusion constraints on concurrent sharing of non-CPU resources, timeliness requirements including assurances on individual activity timeliness behavior, and system-level energy consumption requirements including a non-exhaustable energy budget.

To account for uncertainties in activity properties in dynamic systems, we stochastically describe activity execution demands, and describe activity arrival behaviors using the UAM. Further, we consider Martin's system-level energy consumption model, where each system component's energy consumption is individually modeled and aggregated to account for system's energy consumption.

For such a model, we consider the scheduling optimality criteria of: (1) probabilistically satisfying lower bounds on individual activities' maximal timeliness utilities, and (2) maximizing system-level energy efficiency, while ensuring that the system's energy consumption never exhausts the energy budget and resource mutual exclusion constraints are satisfied.

For this multi-criteria scheduling problem, we present a DVS-based, real-time scheduling algorithm called the *Energy-Bounded Utility Accrual Algorithm* (or EBUA). Since the scheduling problem is $\mathcal{NP}$-hard [37], EBUA heuristically (and dynamically) allocates CPU cycles to activities, computes activity schedules, and scales CPU voltage and frequency with a polynomial-time cost. If activities' cumulative execution demands exceed the available CPU time or may exhaust the system's energy budget, the algorithm defers and rejects jobs in a controlled fashion, minimizing system-level energy consumption and maximizing total accrued utility.

We analytically establish several properties—timeliness and non-timeliness—of EBUA. We prove that the algorithm never exhausts the specified energy budget. Further, we establish EBUA's timeliness optimality during under-loads, freedom from deadlocks, and correctness in mutually exclusive resource sharing. In particular, we prove that the algorithm's timeliness behavior subsumes the optimal timeliness behavior of deadline scheduling as a special case, and identify the conditions under which lower bounds on individual activity utilities are satisfied. In addition, we upper bound the time needed for mutually exclusively accessing shared resources under EBUA.

We conduct experimental studies by simulating the algorithm on the DVS-enabled AMD k6 processor model, and by implementing it on QNX Neutrino 6.2.1 RTOS. Our experimental results validate our analytical results. Further, they confirm EBUA's superiority over other energy-efficient real-time scheduling algorithms on timeliness and energy consumption behaviors.

Thus, the central contribution of the dissertation is the EBUA algorithm. We are not aware of any other efforts that solve the *energy-bounded, TUF/UA scheduling problem under the UAM and system-level energy model* that is solved by EBUA.

1.6 Organization

The rest of the dissertation is organized as follows. We first overview two dynamic real-time applications to provide the motivating context for soft timing constraints and the TUF/UA model. We summarize two significant applications that were successfully implemented using that model in Chapter 2.

In Chapter 3, we outline the activity, resource, and timeliness models, and introduce the system-level energy consumption model. We define energy-bounded systems, state the scheduling objective, and present EBUA in Chapter 4. In Chapter 5, we establish the algorithm's timeliness and non-timeliness properties. Chapter 6 discusses the experimental studies by simulations and implementations.

Chapter 7 surveys the existing efforts on energy-efficient real-time scheduling, focusing on overload scheduling, DVS in real-time systems, and energy-efficient real-time scheduling algorithms. Furthermore, EBUA is compared and contrasted with existing algorithms in this chapter. Finally, the dissertation concludes by describing its contributions and identifying future work in Chapter 8.

Chapter 2

Motivating Applications for TUFs

As example real-time systems requiring the expressiveness and adaptability of TUF time constraints, we summarize TUFs of two applications. These include: (1) AWACS (Airborne WArning and Control System) surveillance mode tracker system [20] built by The MITRE Corporation and The Open Group (TOG); and (2) a coastal air defense system [21] built by General Dynamics (GD) and Carnegie Mellon University (CMU). We only summarize some of the application time constraints here; other details can be found in [20, 21], respectively.

2.1 TUF in AWACS

The AWACS is an airborne radar system with many missions, including air surveillance. Surveillance missions generate aircraft tracks for command and control (C2) and battle management (BM). The surveillance tracker consists of several different activities. Its most demanding computation, called *association*, associates sensor reports to aircraft tracks. The tracker employs two sensors that sweep 180 degrees out of phase with a ten second period. Thus, association has a "critical time" at the period length. If the computation can process

a sensor report for a track in under five seconds (half the sweep), that will provide better data for the corresponding report from the out-of-phase sensor. Thus, prior to critical time, utility of association decreases as critical time nears.

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

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

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

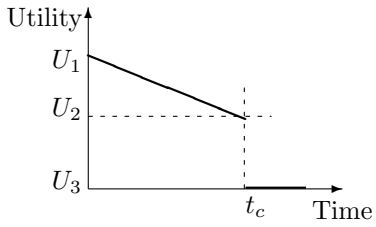
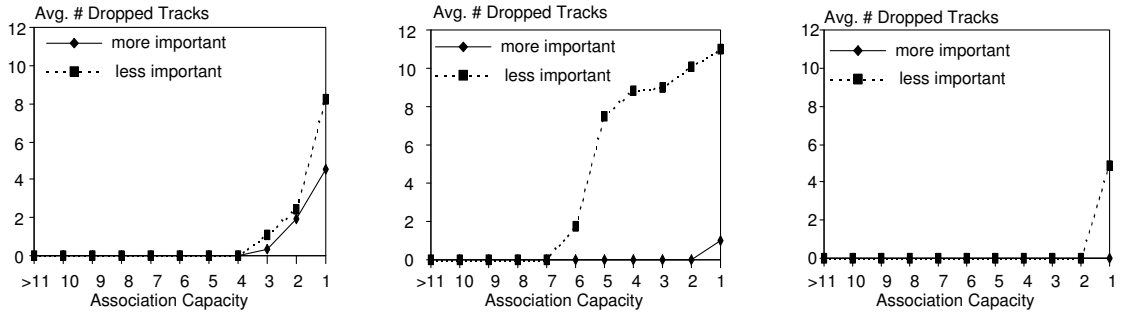


Figure 2.1: Track Association TUF in MITRE/TOG AWACS

The tracker creates an association thread for each airborne object that it receives a sensor report on each sweep period. The association threads execute concurrently with each other

and with the threads for the other tracker activities. The TUFs of all association threads have the same basic shape shown in Figure 2.1, but may have different values for U_1, U_2, and U_3. The tracker's UA scheduling algorithm resolves the resource contention among all the tracker threads and schedules system resources to maximize the total accrued (in this case, summed) utility.

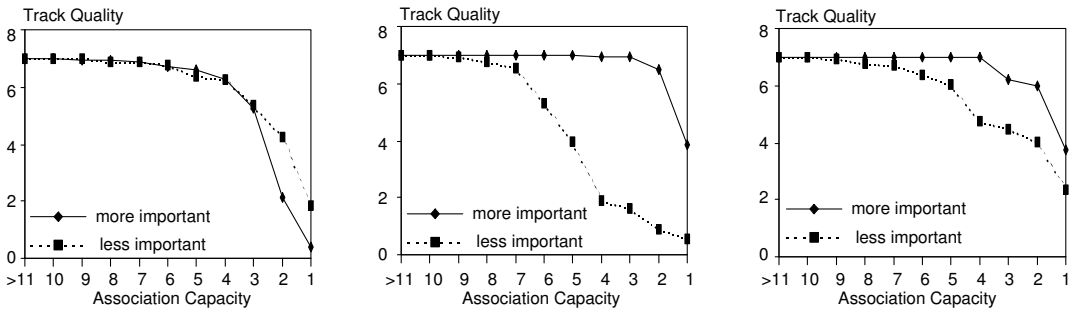


(a) FIFO Priority (b) Fixed Priority (c) UA Algorithm

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

The implementation of the AWACS tracker was done using The Open Group's MK7 operating system [41], which contains the UA scheduling algorithm described in [23]. To understand how well MK7's UA algorithm is able to provide adaptive timeliness, significant performance measurements and comparisons were made from the implementation. Different scheduling algorithms including FIFO (the discipline used by currently deployed AWACS surveillance trackers), and fixed priority (the discipline most commonly used in real-time systems) were used and compared with the UA algorithm.

Figures 2.2 and 2.3 show a snapshot of the measurements. For each of these three disciplines, the figures show the two most important application quality of service (AQoS) metrics for a surveillance tracker: the number of dropped tracks (because it is very expensive in both elapsed time and computation resources to re-acquire a dropped track); and track quality. The x-axes in the figures represent an increasingly constrained system in terms of association capacity. The figures illustrate that (among other things) the UA algorithm sacrifices a little



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

Figure 2.3: Track Quality vs. Association Capacity

quality of more important tracks to maintain higher quality for less important ones, thereby illustrating the adaptivity of the TUF/UA paradigm.

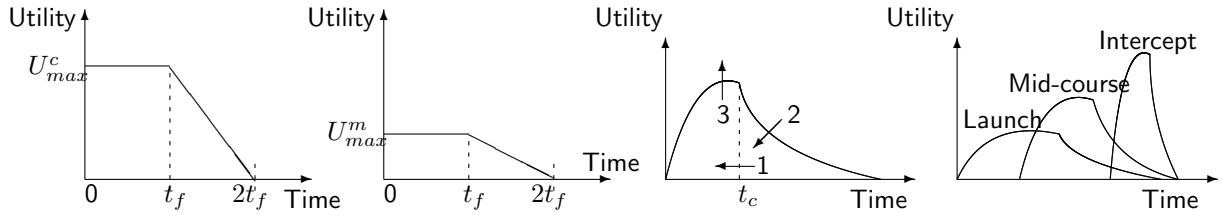
2.2 TUFs in Coastal Air Defense System

The coastal air defense system defends the coastline from incoming cruise missiles and bombers, using a variety of assets including guided interceptor missiles. Time constraints of three activities in the GD/CMU coastal air defense system, called *plot correlation*, *track maintenance*, and *missile control* are shown in Figures 2.4(a), 2.4(b), and 2.4(c), respectively.

The GD/CMU coastal air defense system included a number of activities. Time constraints of two application activities—called *radar plot correlation* and *track database maintenance*—have similar semantics. The correlation activity is responsible for correlating plot reports that arrive from sensor systems against a tracking database. The maintenance activity periodically scans the tracking database, purging old and uncorrelated reports so that stale information does not cause errors in tracking.

Both activities have "critical times" that correspond to the radar frame arrival rate: It is best if both are completed before the arrival of next data frame. However, it is acceptable for them

to be late by one additional time frame under overloads. Furthermore, the correlation activity has a greater utility to the system during overloads. TUF's in Figures 2.4(a) and 2.4(b) reflect these semantics.



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

Figure 2.4: TUFs of Three Activities in GD/CMU Coastal Air Defense

The *missile control activity* of the air defense system provides timely course updates to guide the intercepter such that the hostile targets can be destroyed. However, the frequency and importance of course updates at the desired time depend upon several factors.

As the distance between the target and the interceptor decreases, more frequent course corrections are needed (see arrow 1 in Figure 2.4(c)). In the meanwhile, it is best to abort a late update and restart course correction calculations with fresh information. Arrow 2 in Figure 2.4(c) illustrates how this requirement is reflected by a decrease in the utility obtained for completing the course correction activity after the critical time.

The utility of successfully intercepting a target depends upon the "threat potential" of the target. The threat potential depends upon changing parameters such as the distance of the target from the coastline. For example, intercepting a target that has deeply penetrated inside the coastline yields higher utility than a target that is farther away from the coastline. This is reflected by arrow 3 in Figure 2.4(c) that shows how the function is scaled upward as the threat increases. Figure 2.4(d) shows how the shape of the TUF dynamically changes.

AQoS metrics such as track quality and weapon spherical error probable are used to define

how each service's timeliness contributes to its utility to the current state of the mission. Note that the TUF for sending guidance updates to interceptor missiles have shapes that evolve during the course of each missile's engagement with its incoming target. This adaptive effect is extremely difficult to achieve with priorities. Performance evaluation of the system proves the effectiveness of TUF/UA. For brevity, here we skip the details of TUFs, application implementation, and adaptive timeliness measurements; these can be found in [21].

Chapter 3

Models and Assumptions

This chapter describes the task and resource models, the energy model, and the statistical performance requirements we considered. In describing the models, we outline the scope of the research.

3.1 Tasks and Jobs

We consider a preemptive real-time system which consists of a set of tasks, denoted as $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$. We assume that the target variable voltage processor can be operated at m frequencies $\{f_1, \cdots, f_m | f_1 < \cdots < f_m\}$. Each task T_i has a number of instances (*job*s). With the UAM model, we associate a tuple $\langle a_i, P_i \rangle$ with a task T_i, meaning the maximal number of its instance arrivals during any sliding time window of P_i is a_i. Instances may arrive simultaneously. Note that the periodic model is a special case of UAM model with $\langle \overline{1}, P_i \rangle$, 1 being both the upper and lower bound.

An instance of a task is called a *job*, and we refer to the j^{th} job of task T_i, which is also the j^{th} invocation of T_i, as $J_{i,j}$. The basic scheduling entity that we consider is the job abstraction.

Thus, we use J to denote a job without being task specific, as seen by the scheduler at any scheduling event; J_k can be used to represent a job in the job scheduling queue. Jobs can be preempted at arbitrary times.

3.2 Resource Model

Jobs can access non-CPU resources, which in general, are serially reusable. Examples include physical resources (e.g., disks) and logical resources (e.g., critical sections guarded by mutexes).

Similar to fixed-priority resource access protocols (e.g., priority inheritance, priority ceiling) [42] and that for UA algorithms [24, 22], we consider a single-unit resource model. Thus, only a single instance of a resource is present and a job must explicitly specify the resource that it wants to access.

Resources can be shared and can be subject to mutual exclusion constraints. Thus, only a single job can be accessing such resources at any given time. A job may request multiple shared resources during its lifetime. The requested time intervals for holding resources may be nested, overlapped or disjoint. We assume that a job explicitly releases all granted resources before the end of its execution. A job that is requesting a resource must specify the worst-case time units (task cycles in this dissertation) that it intends to hold the requested resource.

Jobs of different tasks can have precedence constraints. For example, a job J_k can become eligible for execution only after a job J_l has completed, because J_k may require J_l's results. As in [24, 22], we allow such precedences to be programmed as resource dependencies.

3.3 Timeliness Model

A job's time constraint is specified using a TUF. Following [43], a time constraint usually has a "scope"— a segment of the job control flow that is associated with a time constraint. We call such a scope a "scheduling segment." Scheduling segments can be nested or disjoint [43, 22] (see Figure 3.1 for an example). Thus, a job can execute inside multiple scheduling segments. When it does so, it is governed by the "tightest" of the nested time constraints, which is often application-specific (e.g., earliest deadline for step TUFs).

Jobs of a task have the same TUF. Thus, we use $U_i(\cdot)$ to denote task T_i's TUF. The TUF of task T_i's *jth* job is denoted as $U_{i,j}(\cdot)$, which has the same shape as $U_i(\cdot)$. Without being task specific, we use U_{J_k} to denote the TUF of a job J_k; thus completion of the job J_k at a time t will yield a utility $U_{J_k}(t)$. We assume a TUF can take arbitrary values, either positive or negative.

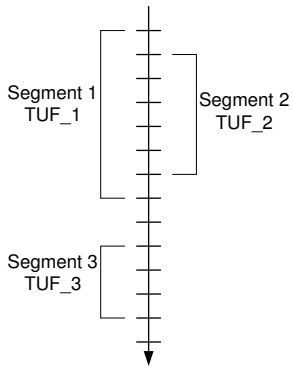


Figure 3.1: Example Scheduling Segments

TUFs can be classified into unimodal and multimodal functions. Unimodal TUFs are those for which any decrease in utility cannot be followed by an increase. Examples are shown in Figures 1.1(a), 1.1(b), and 1.1(c). Note that the traditional hard deadline time constraint can be expressed as a "downward step" function, such as the ones shown in Figure 1.1(a), where the completion of a task at anytime before a certain time will result in uniform utility;

completion of the task after that time will result in zero utility. TUFs which are not unimodal are multimodal. In this dissertation, we restrict our focus to *non-increasing*, unimodal TUFs i.e., those unimodal TUFs for which utility never increases as time advances. Figures 1.1(a) and 1.1(b) show examples. Later, we justify this restriction in Section 3.4.

Each TUF $U_{i,j}, i \in \{1, \cdots, n\}$ has an initial time $I_{i,j}$ and a termination time $X_{i,j}$. The initial time is the earliest time for which the TUF is defined. We assume that $I_{i,j}$ is equal to the arrival time of the job $J_{i,j}$. The termination time denotes the latest time for which the TUF is defined. In this dissertation, we assume that in terms of value, $X_{i,j}$ equals the next job release time, and thus $X_{i,j} - I_{i,j}$ is equal to the period or minimal inter-arrival time P_i of the task T_i.

If $J_{i,j}$'s $X_{i,j}$ is reached and execution of the corresponding job has not been completed, an exception is raised. Normally, this exception will cause $J_{i,j}$'s abortion and execution of exception handlers.

3.4 Statistical Timeliness Performance Requirement

Each task needs to accrue some percentage of its maximum possible utility. The *statistical performance requirement* of a task T_i is denoted as $\{\nu_i, \rho_i\}$, which implies that task T_i should accrue at least ν_i percentage of its maximum possible utility with the probability ρ_i. This is also the requirement for each job of the task T_i. Thus, for example, if $\{\nu_i, \rho_i\} = \{0.7, 0.93\}$, then the task T_i needs to accrue at least 70% of the maximum possible utility with a probability no less than 93%. For step TUFs, ν can only take the value 0 or 1.

This statistical performance requirement on the utility of a task implies a corresponding requirement on the range of task sojourn times. For non-increasing unimodal TUFs, this range is decided only by an upper bound, while for increasing unimodal TUFs, both a lower

bound and an upper bound are needed. In this article, we care about the upper bound. For this reason, we focus on non-increasing TUFs.

3.5 Task Cycle Demands

UA scheduling and DVS are both dependent on the prediction of task cycle demands. We estimate the statistical properties (e.g., distribution, mean, variance) of the demand rather than the worst-case demand for three reasons: (1) Many embedded real-time applications exhibit a large variation in their *actual* workload [20]. Thus, the statistical estimation of the demand is much more stable and hence more predictable than that of the actual workload; (2) worst-case workload information is usually a very conservative prediction of the actual workload [3]. Such conservatism usually results in resource over-supply, which exacerbates the power consumption problem; and (3) allocating cycles based on the statistical estimation of tasks's demands can provide statistical performance assurances. This is sufficient for the applications of interest to us. In fact, stronger assurances are generally infeasible for dynamic, embedded real-time systems.

Let Y_i be the random variable of task T_i's cycle demand. Estimating the task demand distribution involves two steps: (1) profiling its cycle usage and (2) deriving the probability distribution of the usage. Recently, a number of measurement-based profiling mechanisms have been proposed [44, 45, 46]. Profiling can be performed online or off-line. Off-line profiling provides more accurate estimation with the whole trace of CPU usage, but it is not applicable to "live" applications.

We assume that the mean and variance of task cycle demands are finite and determined through either online or off-line profiling. We denote the expected number of processor cycles required by a task T_i as $E(Y_i)$, and the variance on the workload as $Var(Y_i)$. Note

that, under a constant speed i.e., frequency f (given in cycles per second), the expected execution time of a task T_i is given by $e_i = \frac{E(Y_i)}{f}$.

3.6 Power and Energy Consumption Model

We consider Martin's *system level* energy consumption model that was derived from experimental observations that some components of a computer consume constant power, while others consume power that is scalable to either voltage or frequency [30, 31, 29]. We use this model to derive the energy consumption per cycle. This is summarized as follows:

The CPU is assumed to be capable of executing tasks at m clock frequencies. When the CPU operates at a frequency f, the CPU's dynamic power consumption, denoted as P_d, is given by $P_d = C_{ef} \times V_{dd}^2 \times f$, where C_{ef} is the effective switch capacitance and V_{dd} is the supply voltage. On the other hand, the clock frequency is almost linearly related to the supply voltage, since $f = k \times \frac{(V_{dd}-V_t)^2}{V_{dd}}$, where k is constant and V_t is the threshold voltage [47]. By approximation, $f = a \times V_{dd}$, where a is constant. Thus, $P_d = \frac{C_{ef}}{a^2} \times f^3$, which is equivalent to $P_d = S_3 \times f^3$, where S_3 is constant. In this case, both the supply voltage and the clock frequency can be scaled.

Besides the CPU, there are also other system components that consume energy. Given the dynamic power consumption equation $P_d = C_{ef} \times V_{dd}^2 \times f$, power consumption equations for all other system components can be derived. Some components in the system must operate at a fixed voltage and thus their power can only scale with frequency. Examples include main memory. In this case, $C_{ef} \times V_{dd}^2$ can be represented as another constant such as S_1, and the equation becomes $P_d = S_1 \times f$. Other components in the system consume constant power with respect to the CPU clock frequency. Examples include display devices. Thus, their power consumption can be represented as S_0, where S_0 is constant.

Finally, for completeness in fitting the measured power of a system to the cubic equation, another term is included to represent the quadratic term i.e., $P_d = S_2' \times V_{dd}^2$. Since f is almost linearly related to V_{dd} ($f \approx a \times V_{dd}$), P_d is represented as $P_d = S_2 \times f^2$. While this term does not represent the dynamic power consumption of CMOS, because it implies that V_{dd} is being lowered without also lowering f, in practice, it may appear because of variations in DC-DC regulator efficiency across the range of output power, CMOS leakage currents, and other second order effects [30].

Summing the power consumption of all system components together, a single equation for the system-level power consumption can be obtained as: $P = S_3 \times f^3 + S_2 \times f^2 + S_1 \times f + S_0$, where f is the CPU clock frequency and S_0, S_1, S_2, and S_3 are system parameters. The corresponding energy consumption of a task T_i is given by the equation $E_i = P \times e_i$, where e_i denotes T_i's expected execution time. Therefore, the expected energy consumption per cycle is given by:

$$E^e(f) = S_3 \times f^2 + S_2 \times f + S_1 + \frac{S_0}{f} \tag{3.1}$$

Note that we use $E^e(f)$ to denote the energy consumption per cycle under the frequency f, whereas we denote the mean value of the random variable Y_i as $E(Y_i)$.

We consider a single processor system that relies on battery power. Thus, the system has a limited energy budget bound E_{bnd}. Further, we assume that the system must remain operational in the mission time interval $[0, MT]$. Therefore, the system is subject to a *hard* energy constraint, and the total system-level energy consumption should not exceed E_{bnd} for MT time units. Re-charging the battery is not possible or feasible during the mission.

For a system with DVS capability and m possible frequencies, we assume that, during any time interval $[t_1, t_2]$, total cycles executed with CPU speed f_i are denoted by $cyc_i, i \in \{1, \cdots, m\}$. Thus, total system-level energy consumption during the time interval $[t_1, t_2]$ is given by $E^e(t_1, t_2) = \sum_{i=1}^{m} E^e(f_i) \times cyc_i$, where $E^e(f_i)$ is derived from Equation 3.1.

Chapter 4

The EBUA Algorithm

This chapter presents details of EBUA algorithm. We first define the energy-bounded UA scheduling and discuss the scheduling objective in Section 4.1. Then we describe our approach to determine task critical times, and statistical estimation of task cycle demands, in Section 4.2 and Section 4.3, respectively. Rationale of the algorithm is discussed in Section 4.4. This discussion is followed by procedural description of the EBUA algorithm in Section 4.5. Finally, we analyzes the complexity of EBUA in Section 4.6.

4.1 Definition and Scheduling Objective

For a real-time system with statistical performance requirements that must remain operational during the mission time $[0, MT]$, there is a minimum amount of energy needed to meet all the timeliness requirements when the system is not overloaded, while sustaining the system's operation until the end of the mission. We refer to this required threshold energy as E_{rqd} hereafter. As mentioned in Section 3.6, we denote the limited energy budget bound of a real-time system as E_{bnd}.

Definition 4.1.1. If $E_{bnd} < E_{rqd}$, a real-time system is **energy-bounded**; energy-efficient UA scheduling in such a system is called energy-bounded UA scheduling.

If $E_{bnd} \geq E_{rqd}$, we should try to satisfy the performance requirements. For an energy-bounded system with $E_{bnd} < E_{rqd}$, trying to execute all the jobs may result in a situation where the system runs out of energy in the middle of the mission. Thus, our aim then is to provide maximum energy efficiency while sustaining the system operation until the end of the mission.

With the problem definition, we consider a two-fold scheduling criterion: (1) assure that each task T_i accrues the specified percentage ν_i of its maximum possible utility with at least probability ρ_i; and (2) maximize the system-level "energy efficiency," under the condition of assuring the feasibility of the system with limited energy budget i.e., the consumed energy during an operation/mission never exceeds the energy bound E_{bnd}. When it is not possible to satisfy $\{\nu_i, \rho_i\}$ for each task, our objective goes to (2).

Equation 3.1 indicates that there is an optimal value (not necessarily the lowest one) for clock frequency that minimizes system-level energy consumption. This adds to the difficulty to decide whether a system is energy-bounded. Some simple cases can be derived. For example, the optimal CPU speed for periodic tasks that always execute their worst-case cycles is constant and equal to the worst-case aggregate CPU demand (see [3, 33]). Thus, with this periodic task model, if E_{bnd} is smaller than the energy needed to execute all tasks with the speed equal to the aggregate CPU demand, which is just E_{rqd}, then the system is energy-bounded. Otherwise, it is not.

Intuitively, for a system that is not energy-bounded, during overloads the scheduling objective becomes utility maximization under energy constraints, since DVS tends to select the highest frequency, making the system consume constant energy. During under-loads, the algorithm delivers the performance assurances. Thus, the scheduling objective becomes

the dual criterion problem of utility maximization, i.e., minimizing energy while achieving the given utility within the given time constraint. Such intuitions are slightly changed for energy-bounded systems. When $E_{bnd} < E_{rqd}$, our objective becomes utility maximization under fixed, limited energy consumption bound.

This problem is $\mathcal{NP}$-hard because it subsumes the problem of scheduling dependent tasks with step-shaped TUFs, which has been shown to be $\mathcal{NP}$-hard in [24].

4.2 Determining Task Critical Time

To assure that tasks accrue their desired utility percentage and maximize the energy efficiency, EBUA needs to provide predictable CPU scheduling and speed scaling.

Let $s_{i,j}$ be the sojourn time of the *jth* job of task T_i. Then, we have $Pr\left(U_i(s_{i,j}) \geq \nu_i \times U_i^{max}\right) \geq \rho_i$. By the assumption of non-increasing TUFs, it is sufficient to have $Pr(s_{i,j} \leq D_i) \geq \rho_i$, where D_i is the upper bound on the sojourn time of task T_i. D_i is calculated as $D_i = U_i^{-1}(\nu_i \times U_i^{max})$, where $U_i^{-1}(x)$ denotes the inverse function of TUF $U_i(\cdot)$. If there are more than one point on the time axis that correspond to $\nu_i \times U_i^{max}$, we choose the latest point. By doing so, we can potentially reduce the CPU bandwidth demand of a task. We call D_i "critical time" hereafter. Thus, T_i is probabilistically assured to accrue at least the utility percentage $\nu_i = U_i(D_i)/U_i^{max}$, with probability ρ_i.

Note that the period or minimum inter-arrival time P_i and critical time D_i of the task T_i have the following relations: (1) $P_i = D_i$ for a binary-valued, downward step TUF whose utility drops to a zero value at time P_i; and (2) $P_i \geq D_i$, for other non-increasing TUFs.

4.3 Statistical Estimation of Demand

EBUA's next step is to decide the number of cycles that must be allocated to each task. To provide the statistical timeliness assurance while maximizing energy efficiency, EBUA allocates cycles based on the statistical requirements and demand of each task. Knowing the mean and variance of task T_i's demand Y_i, by a one-tailed version of the Chebyshev's inequality, $Pr[Y_i \geq y] \leq \frac{Var(Y_i)}{Var(Y_i)+(y-E(Y_i))^2}$, when $y \geq E(Y_i)$, we have:

$$Pr[Y_i < y] \geq \frac{(y - E(Y_i))^2}{Var(Y_i) + (y - E(Y_i))^2} \tag{4.1}$$

From a probabilistic point of view, Equation 4.1 is the direct result of the cumulative distribution function of the task T_i's cycle demands i.e., $F_i(y) = Pr[Y_i \leq y]$. Now, let ρ_i be the statistical performance requirement of T_i i.e., each job $J_{i,j}$ of task T_i must accrue ν_i percentage of utility with a probability ρ_i. To satisfy this requirement, we let $\rho_i = \frac{(c_i-E(Y_i))^2}{Var(Y_i)+(c_i-E(Y_i))^2}$ and obtain the minimal required $c_i = E(Y_i) + \sqrt{\frac{\rho_i \times Var(Y_i)}{1-\rho_i}}$.

Thus, the scheduler allocates c_i cycles to each job $J_{i,j}$, so that the probability that job $J_{i,j}$ requires no more than the allocated c_i cycles is at least ρ_i i.e., $Pr[Y_i < c_i] \geq \rho_i$.

4.4 Algorithm Rationale

For an energy-bounded system, the jobs to be executed should be carefully selected in order to make best use of available energy. Such selection process is guided by the performance metric *Utility and Energy Ratio* (or UER). UER is defined to facilitate optimization of timeliness objectives and energy consumption in a unified way.

A job's UER measures the amount of utility that can be accrued per unit energy consumption by executing the job and the job(s) that it depends upon (due to resource dependencies). A

job also has a Local UER (LoUER), which is defined as the UER that the job can potentially accrue by itself at the current time, if it were to continue its execution. The LoUER of T_i under frequency f at time t is calculated as $\frac{U_i(t+c_i/f)}{c_i \times E^e(f)}$, where $E^e(f)$ is derived using Equation 3.1. Equation 3.1 indicates that there is an optimal value for clock frequency to maximize T_i's LoUER.

4.5 Procedural Description

For the UAM model, we denote C_i as the total cycles of a_i jobs in the time window P_i, i.e., $C_i = a_i c_i$. With C_i, we define the *aggregate CPU demand* of the task set $\mathbf{T}$ as

$$CPU_{dmd} = \sum_{i=1}^{n} \frac{C_i}{P_i} \tag{4.2}$$

million cycles per second (MHz), and define the *system load* (*Load*) as

$$Load = \frac{1}{f_m} \sum_{i=1}^{n} \frac{C_i}{D_i} \tag{4.3}$$

Note that we do not denote CPU_{dmd} as $\sum_{i=1}^{n} C_i/D_i$, and only for step TUFs whose utility drop to a zero value at time P_is, $\sum_{i=1}^{n} C_i/P_i = \sum_{i=1}^{n} C_i/D_i$. The reason is elaborated in Section 4.5.7.

To describe EBUA, we define the following variables and auxiliary functions:

- $E_{rem} \leq E_{bnd}$; it is the system's remaining energy for execution.
- $\mathbf{T}$ is the task set. For task T_i, during a time window P_i, D_i^a and c_i^r are its earliest job's absolute critical time and remaining cycles, respectively. We define the task-level flag SEL_i to represent whether task T_i's instances are selected in the job selection phase. $SEL_i = skipped$ indicates that the task is skipped, and $SEL_i = selected$ indicates that it is, or can be selected for execution.

- $\mathcal{J}_r = \{J_1, J_2, \cdots, J_{n'}\}$ is the current unscheduled job set; σ is the ordered output schedule. $J_k \in \mathcal{J}_r$ is a job; $J_k.Dep$ is its dependency list.
- $J_k.D$ is job J_k's critical time; $J_k.X$ is its termination time, and $J_k.c$ is its remaining cycles.
- $T(J_k)$ returns the corresponding task of job J_k. Thus, if $T_i = T(J_k)$, then $J_k.c = c_i^r$, and $J_k.D = D_i^a$.
- Function `owner(`R`)` denotes the jobs that are currently holding resource R; `reqRes(`T`)` returns the resource requested by T.
- `headOf(`σ`)` returns the first job in σ; `sortByUER(`σ`)` sorts σ by each job's UER, in a non-increasing order. `selectFreq(`x`)` returns the lowest frequency $f_i \in \{f_1 < \cdots < f_m\}$, such that $x \leq f_i$.
- `insert(`T`,`σ`,`I`)` inserts T in the ordered list σ at the position indicated by index I; if there are already entries in σ at the index I, T is inserted before them. After insertion, the index of T in σ is I.
- `remove(`T`,`σ`,`I`)` removes T from ordered list σ at the position indicated by index I; if T is not present at the position in σ, the function takes no action.
- `lookup(`T`,`σ`)` returns the index value associated with the first occurrence of T in the ordered list σ.
- `feasible(`σ`)` returns a boolean value indicating schedule σ's feasibility. For a schedule σ to be feasible, the predicted completion time of each job in σ, calculated at the highest frequency f_m, must not exceed its termination time.
- `eBounded(`σ`)` checks whether the predicted energy consumption of σ is less than the allowed bound of energy consumption before σ's predicted completion time.
- `updateSel(`$\mathbf{T}$`)` updates the flag SEL_i for the task set $\mathbf{T}$.

4.5.1 Offline Computing

We consider the energy consumed by the *system* instead of that by just the processor and seek to maximize energy efficiency. We know that the processor can be operated at m frequencies $\{f_1, f_2, \cdots, f_m | f_1 < \cdots < f_m\}$. Equation 3.1 indicates that there is an optimal value (not necessarily the lowest one) for clock frequency that minimizes E_i for a task T_i.

EBUA first decides the optimal frequency for each task T_i that maximizes the task's LoUER. The procedure `offlineComputing()` is shown in Algorithm 4.1. It runs at $t = 0$ and calculates D_i and c_i for each task as described in Section 4.2 and Section 4.3. It then computes the optimal frequency $f^o_{T_i} \in \{f_1, \cdots, f_m\}$ for each task T_i, which maximizes T_i's LoUER (line 5). Each task initially runs at the speed $f^{ini}_{T_i}$, which is set in line 6.

Algorithm 4.1: `offlineComputing()`

1: **input** : Task set **T**;
2: **output** : D_i, c_i, $f^o_{T_i}$, $f^{ini}_{T_i}$;
3: $D_i = U_i^{-1}(\nu_i \times U_i^{max})$;
4: $c_i = E(Y_i) + \sqrt{\frac{\rho_i \times Var(Y_i)}{1-\rho_i}}$;
5: Decide $f^o_{T_i}$, such that $\frac{U_i(c_i/f^o_{T_i})}{c_i \times E^e(f^o_{T_i})} = \texttt{max}(\frac{U_i(c_i/f_j)}{c_i \times E^e(f_j)})$, $\forall j \in \{1, 2, \cdots, m\}$;
6: $f^{ini}_{T_i} = \texttt{max}(f^o_{T_i}, \texttt{selectFreq}(CPU_{dmd}))$

4.5.2 Energy-Bounded UA Scheduling

The scheduling events of EBUA include the arrival and completion of a job, a resource request, a resource release, and the expiration of a time constraint such as the arrival of the termination time of a TUF. A description of EBUA at a high level of abstraction is shown in Algorithm 4.2.

We include the procedure `offlineComputing()` in line 3, but this sub-routine is only exe-

cuted at $t = 0$. When EBUA is invoked at time t_{cur}, it first updates each task's remaining cycle (line 5–9). The algorithm then checks the feasibility of the jobs. If the earliest predicted completion time of a job is later than its termination time, then it can be safely aborted (line 11–13). Otherwise, EBUA builds the dependency list for the job (line 15).

Algorithm 4.2: EBUA: High Level Description

```
1:  input  : T = {T_1, ..., T_n}, J_r = {J_1, ..., J_n'}, E_rem
2:  output : selected job J_exe and frequency f_exe
3:  offlineComputing (T);
4:  Initialization: t := t_cur, σ := ∅, update E_rem;
5:  switch triggering event do
6:      case task_release(T_i)        c_i^r := c_i;
7:      case task_completion(T_i)     c_i^r := 0;
8:      otherwise                     Update c_i^r;
9:  endsw
10: for ∀J_k ∈ J_r do
11:     if feasible(J_k)=false then
12:         c^r_{T(J_k)} := 0;
13:         abort(J_k);
14:     else
15:         J_k.Dep := buildDep(J_k);
16:     endif
17: endfor
18: for ∀J_k ∈ J_r do
19:     J_k.UER:=calculateUER(J_k, t);
20: endfor
21: σ_tmp :=sortByUER(J_r);
22: for ∀J_k ∈ σ_tmp from head to tail do
23:     if J_k.UER > 0 then
24:         σ := insertByECF(σ, J_k);
25:     else
26:         break;
27:     endif
28: endfor
29: updateSel(T);
30: J_exe:=headOf(σ);
31: f_exe:=decideFreq(T, J_exe, t);
32: return J_exe and f_exe;
```

At each scheduling event, for all the n' jobs $\mathcal{J}_r = \{J_1, J_2, \cdots, J_{n'}\}$ currently in the scheduling queue, the UER of each job is computed by `calculateUER()` under the highest frequency

f_m. The jobs are then sorted by their UERs (line 18-21), in a non-increasing order.

In each step of the `for` loop from line 22 to 28, the job with the largest UER and its dependencies are inserted into σ, if it can produce a positive UER. The insertion is done based on the order of jobs' critical times (earliest critical time first), while respecting their resource dependencies. The schedule σ as the output of procedure `insertByECF()` is a feasible and energy-bounded schedule, sorted by the jobs' critical times, in an non-decreasing order.

If the system is overloaded, it is possible that the queue $\mathcal{J}_r = \{J_1, J_2, \cdots, J_{n'}\}$, whose *queue load* (Qload) defined as $\frac{1}{f_m}\sum_{k=1}^{n'}(C_{J_k}/(J_k.X - t_{cur}))$, is also overloaded. Note that $J_k.X$ refers to the termination time of J_k, and t_{cur} is the current time. Thus, upon inserting a job, in `insertByECF()` EBUA performs feasibility check as well as energy bound check, and ensures the feasibility (on both energy consumption and timing) of the tentative schedule by dropping some jobs.

At line 29, EBUA updates the flag SEL_i for each task. SEL_i is set to be *skipped* if and only if T_i has instances in the ready job queue $\mathcal{J}_r$ of line 21, but none of them are selected in σ. Note that even when T_i has no jobs in $\mathcal{J}_r$, SEL_i is set to be *selected*.

Finally, from line 30 to 32, with algorithm `decideFreq()`, EBUA analyzes the demands of the task set and applies DVS to decide the execution frequency f_{exe} for the selected job J_{exe}, i.e., the one at the head of the tentative schedule σ. DVS can reduce the energy consumption of the selected jobs whenever possible, which enables us to further increase excess energy that can be later used for job selection.

Intuitively, during overloads it is very possible for the DVS technique to select the highest frequency f_m for the execution of the processor, since the aggregate CPU demand defined in Equation 4.2 is higher than f_m. Therefore, during overloads, with the constant energy consumption at frequency f_m, to maximize the collective utility per unit energy as our objective, we need to maximize the collective utility. This is exactly why we sort the jobs

based on their UERs and perform the feasibility check. Such heuristics are explained in detail in the next sections.

4.5.3 Resource and Deadlock Handling

Before EBUA can compute job partial schedules, the dependency chain of each job must be determined, as shown in Algorithm 4.3.

Algorithm 4.3 follows the chain of resource request/ownership. For convenience, the input job J_k is also included in its own dependency list. Each job J_l other than J_k in the dependency list has a successor job that needs a resource which is currently held by J_l. Algorithm 4.3 stops either because a predecessor job does not need any resource or the requested resource is free. Note that "$\cdot$" denotes an append operation. Thus, the dependency list starts with J_k 's farthest predecessor and ends with J_k.

Algorithm 4.3: `buildDep()`: Build Dependency List

```
1: input  : Job J_k;
2: output : J_k.Dep;
3: Initialization : J_k.Dep := J_k; Prev := J_k;
4: while reqRes(Prev) ≠ ∅ ⋀ owner(reqRes(Prev)) ≠ ∅ do
       /* add new owner at the head of the list */
5:     J_k.Dep :=owner(reqRes(Prev)) ·J_k.Dep;
6:     Prev := owner(reqRes(Prev));
7: endw
```

To handle deadlocks, we consider a deadlock detection and resolution strategy, instead of a deadlock prevention or avoidance strategy. Our rationale for this is that deadlock prevention or avoidance strategies normally pose extra requirements; for example, resources must always be requested in ascending order of their identifiers.

Further, restricted resource access operations that can prevent or avoid deadlocks, as done in many resource access protocols, are not appropriate for the class of embedded real-time

systems that we focus on. For example, the Priority Ceiling protocol [42] assumes that the highest priority of jobs accessing a resource is known. Likewise, the Stack Resource policy [48] assumes preemptive "levels" of threads *a priori*. Such assumptions are too restrictive for the class of systems that we focus on (due to their dynamic nature).

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

Algorithm 4.4: Deadlock Detection and Resolution

```
1:  input  : Requesting job J_k, t_cur;
    /* deadlock detection */
2:  Deadlock := false;
3:  J_l := owner(reqRes(J_k));
4:  while J_l ≠ ∅ do
5:      J_l.LoUER := U_{J_l}(t_cur + J_l.c/f_m)/(J_l.c × E^e(f_m));
6:      if J_l = J_k then
7:          Deadlock := true;
8:          break;
9:      else
10:         J_l := owner(reqRes(J_l));
11:     endif
12: endw
    /* deadlock resolution if any */
13: if Deadlock = true then
14:     abort(The job J_m with the minimal LoUER in the cycle);
15: endif
```

The deadlock detection and resolution algorithm (Algorithm 4.4) is invoked by the scheduler whenever a job requests a resource. Initially, there is no deadlock in the system. By induction, it can be shown that a deadlock can occur if and only if the edge that arises in the resource graph due to the new resource request lies on a cycle. Thus, it is sufficient to check if the new edge resulting from the job's resource request produces a cycle in the resource graph.

To resolve the deadlock, some job needs to be aborted. If a job J_l were to be aborted, then

its timeliness utility is lost, but energy is still consumed. To minimize such loss, we compute the LoUER of each job at t_{cur} at the frequency f_m. EBUA aborts the job with the minimal LoUER in the cycle to resolve a deadlock.

4.5.4 Manipulating Partial Schedules

The `calculateUER()` algorithm (Algorithm 4.5) accepts a job J_k (with its dependency list) and the current time t_{cur}. On completion, the algorithm determines UER for J_k, by assuming that jobs in $J_k.Dep$ are executed from the current position (at time t_{cur}) in the schedule, while following the dependencies.

Algorithm 4.5: `calculateUER()`

```
1:  input  : J_k, t_cur;
2:  output : J_k.UER;
3:  Initialization : C_c := 0, E := 0, U := 0;
4:  for ∀J_l ∈ J_k.Dep, from head to tail do
5:  |   C_c := C_c + J_l.c;
6:  |   U := U + U_{J_l}(t_cur + C_c / f_m);
7:  endfor
8:  E := E^e(f_m) × C_c;
9:  J_k.UER := U / E;
10: return J_k.UER;
```

To compute J_k's UER at time t_{cur}, EBUA considers each job J_l that is in J_k's dependency chain, which needs to be completed before executing J_k. The total computation cycles that will be executed upon completing J_k is counted using the variable C_c of line 5. With the known expected computation cycles of each task, we can derive the expected completion time and expected energy consumption under f_m for each task, and thus get their accrued utility to calculate UER for J_k.

Thus, the total execution time (under f_m) of the job J_k and its dependents consists of two parts: (1) the time needed to execute the jobs holding the resources that are needed to

execute J_k; and (2) the remaining execution time of J_k itself. According to the process of `buildDep()`, all the relative jobs are included in $J_k.Dep$.

Note that we are calculating each job's UER assuming that the jobs are executed at the current position in the schedule. This would not be true in the output schedule σ, and thus affects the accuracy of UERs calculated. But with the non-increasing shape of each job's TUF, we are calculating the highest possible UER of each job by assuming that it is executed at the current position. Intuitively, this would benefit the final UER, since `insertByECF()` always takes the job with the highest UER at each insertion on σ. Also, the UER calculated for the scheduled job, which is at the head of the feasible schedule, is always accurate.

The details of `insertByECF()` in line 24 of Algorithm 4.2 are shown in Algorithm 4.6. `insertByECF()` updates the tentative schedule σ by attempting to insert each job along with all of its dependencies to σ. The updated σ is an ordered list of jobs, where each job is placed according to the critical time it should meet.

Note that the time constraint that a job should meet is not necessarily the job critical time. In fact, the index value of each job in σ is the actual time constraint that the job must meet.

A job may need to meet an earlier critical time in order to enable another job to meet its time constraint. Whenever a job is considered for insertion in σ, it is scheduled to meet its own critical time. However, all of the jobs in its dependency list must execute before it can execute, and therefore, must precede it in the schedule. The index values of the dependencies can be changed with `insert()` in line 14 of Algorithm 4.6.

The variable $CuCT$ is used to keep track of this information. Initially, it is set to be the critical time of job J_k, which is tentatively added to the schedule (line 6, Algorithm 4.6). Thereafter, any job in $J_k.Dep$ with a later time constraint than $CuCT$ is required to meet $CuCT$. If, however, a job has a tighter critical time than $CuCT$, then it is scheduled to meet the tighter critical time, and $CuCT$ is advanced to that time since all jobs left in $J_k.Dep$

Algorithm 4.6: `insertByECF()`

```
1:  input  : J_k and an ordered job list σ;
2:  output : the updated list σ;
3:  if J_k ∉ σ then
4:      copy σ into σ_tent: σ_tent :=σ;
5:      insert(J_k, σ_tent, J_k.D);
6:      CuCT = J_k.D;
7:      for ∀J_l ∈ {J_k.Dep − J_k} from tail to head do
8:          if J_l ∈ σ_tent then
9:              CT=lookup(J_l, σ_tent);
10:             if CT < CuCT then continue;
11:             else remove(J_l, σ_tent, CT);
12:         endif
13:         CuCT:=min(CuCT, J_l.D);
14:         insert(J_l, σ_tent, CuCT);
15:     endfor
16:     if feasible(σ_tent) and eBounded(σ_tent) then
17:         σ := σ_tent;
18:     endif
19: endif
20: return σ;
```

must complete by then (lines 13–14, Algorithm 4.6). Finally, if this insertion produces a feasible schedule and does not exceed the allowed energy budget, the jobs are included in the schedule; otherwise, not (lines 16–17). We will explain how to monitor the dynamic energy consumption in Section 4.5.5.

It is worth noting that the procedure `insertByECF()` sorts jobs in the non-decreasing critical time order if possible, but its sub-procedure `feasible()` checks the feasibility of σ_{tent} based on each job's termination time. This is because a job's critical time is smaller or equal to its termination time according to our calculation in Section 4.2. So even if a job cannot complete before its critical time, it may still accrue some utility, as long as it finishes before its termination time. Thus, we need to prevent "over-killing" in `feasible()`. The effectiveness of such prevention is further illustrated in Section 6.2.1.

4.5.5 Monitoring Energy Consumption Online

Since our energy bound E_{bnd} is associated with the mission time $[0, MT]$, we need to dynamically monitor the system-level energy consumption and adjust the selected jobs to execute. The online monitoring is conducted with the function `eBounded(`σ_{tent}`)` in line 16 of Algorithm 4.6.

`offlineComputing(T)` sets the initial speed $f_{T_i}^{ini}$ for each task. We assume the last job in σ_{tent} has the predicted completion time D_{tent}. Thus, for σ_{tent} where each job is executed at its initial speed, its expected energy consumption $E^e(t_{cur}, D_{tent})$ can be calculated by the mechanism described in the last paragraph of Section 3.6. When $E_{bnd} - E_{rem} + E^e(t_{cur}, D_{tent}) \leq \frac{D_{tent}}{MT} \times E_{bnd}$, `eBounded(`$\sigma_{tent}$`)` returns $true$; otherwise, it returns $false$.

4.5.6 Deciding the Processor Frequency

The parameter c_i determines *how long* (in number of cycles) to execute each task. We now discuss the other scheduling dimensions—*how fast* (i.e., CPU speed scaling) and *when* to execute each task.

We first consider a simple system which is not energy-bounded, and contains a set of n periodic tasks with parameters $\langle \overline{1}, P_i \rangle$ and step TUFs. The intuitive idea is to assign a uniform speed to execute all tasks until the task set changes. Assume that each task is allocated c_i cycles within its P_i (thus D_i). The aggregate CPU demand of the concurrent tasks is: $\sum_{i=1}^{n} \frac{C_i}{P_i} = \sum_{i=1}^{n} \frac{C_i}{D_i}$ MHz. To meet this aggregate demand, the CPU only needs to run at speed $\sum_{i=1}^{n} \frac{C_i}{P_i}$. Equation 4.2 thus gives the static, optimal CPU speed to minimize the total energy while meeting all the D_i under the traditional energy consumption model, *assuming that each task presents its worst-case workload to the processor at every instance* [3].

However, the cycle demands of tasks often vary greatly. In particular, a task may, and often

does, complete a job before using up its allocated cycles. Such early completion often results in CPU idle time, thereby wasting energy. To save this energy, we need to dynamically adjust the CPU speed.

In general, there are two dynamic speed scaling approaches, namely the conservative approach and the aggressive approach. The conservative approach assumes that a job will use its allocated cycles, and starts a job with at above static optimal speed and then decelerates when the job completes early. On the other hand, the aggressive approach assumes that a job will use fewer cycles than allocated, and starts a job at a lower speed and then accelerates as the job progresses. The aggressive approach is adopted in [11, 3, 7], because it saves more energy for jobs that complete early, and most jobs in its studied application use fewer cycles than allocated.

We adopt the aggressive approach. EBUA applies a stochastic DVS technique based on the Look-Ahead EDF (LaEDF) technique discussed in [7]. The detailed DVS rationale and approach are described in the next section.

4.5.7 DVS with the Energy Bound

We have CPU-time reclamation through DVS. When tasks complete early, we have some slack time that can be used to further increase the excess energy by reducing the execution speeds of some subsequent jobs (as long as this does not compromise the sojourn times of already selected jobs).

We consider the "processor demand approach" [49] to analyze the feasibility of tasks with stochastic parameters and UAM arrival model.

Theorem 4.1. *For a task T_i with a UAM pattern $\langle a_i, P_i \rangle$ and critical time D_i, all its jobs can meet their D_i, if T_i is executed at a frequency no lower than $\frac{C_i}{D_i}$, where C_i is the total*

cycles of a_i jobs in the time window P_i, i,e., $C_i = a_i c_i$.

Proof The necessary and sufficient condition for satisfying job critical times is $fL \geq C_i(0, L), \forall L > 0$, where f is the processor frequency allocated to T_i, and $C_i(0, L)$ is the cycle demand of task T_i on the time interval $[0, L]$, i.e., $C_i(0, L) = \left(\left\lfloor \frac{L-D_i}{P_i} \right\rfloor + 1\right) C_i$. Thus, we need $f \geq \frac{1}{L}\left(\left\lfloor \frac{L-D_i}{P_i} \right\rfloor + 1\right) C_i$, $\forall L > 0$. Since $\left(\left\lfloor \frac{L-D_i}{P_i} \right\rfloor + 1\right) \leq \left(\frac{L-D_i}{P_i} + 1\right)$, it is sufficient to have $f \geq \frac{1}{L}\left(\frac{L-D_i}{P_i} + 1\right) C_i = \frac{C_i}{P_i}\left(1 + \frac{P_i - D_i}{L}\right)$, $\forall L > 0$. As $P_i \geq D_i$, we have:

Case 1: $P_i > D_i$

It is easy to see that $\frac{C_i}{P_i}\left(1 + \frac{P_i - D_i}{L}\right)$ monotonically decreases with the increase of L. Furthermore, notice that if $L \leq D_i$, $C_i(0, L) = 0$ because in such an interval $[0, L]$, no job has a critical time earlier than D_i. Thus, it is sufficient to consider the case where L has the smallest possible value—when $L = D_i$, $f \geq C_i/D_i$.

Case 2: $P_i = D_i$

When $P_i = D_i$, $\frac{C_i}{P_i}\left(1 + \frac{P_i - D_i}{L}\right)$ is independent of L. It can be seen that $f \geq \frac{C_i}{P_i} = \frac{C_i}{D_i}$.

Combining the above two cases, we have a sufficient condition $f \geq C_i/D_i$. □

From the above proof, for a task T_i, when $P_i > D_i$, T_i's CPU demand can be denoted as $\frac{C_i}{D_i}$ [38]. But the task set's aggregate CPU demand will be overestimated if we denote it as $\sum_{i=1}^{n} \frac{C_i}{D_i}$. This is because, when $P_i = D_i$, our calculation will safely assume that beyond Di, the next invocation of task T_i starts immediately and consumes $\frac{Ci}{Pi}$ processing resources. But when $P_i > D_i$, substituting the critical time D_i for P_i will underestimate the future processing capacity that is available.

We elaborate such estimation also through the processor demand approach [49, 50]. The necessary and sufficient condition for satisfying a task set's critical times is $fL \geq \sum_{i=1}^{n} \left\lfloor \frac{L+P_i-D_i}{P_i} \right\rfloor C_i$, $\forall L > 0$, where f is the processor frequency allocated to the task set, and $\left\lfloor \frac{L+P_i-D_i}{P_i} \right\rfloor C_i$ is the cycle demand of task T_i during the time interval $[0, L]$. Since $\left\lfloor \frac{L+P_i-D_i}{P_i} \right\rfloor \leq \left(\frac{L}{P_i} + \frac{P_i-D_i}{P_i}\right)$, it

is sufficient to have:

$$f \geq \sum_{i=1}^{n} \frac{C_i}{L}\left(\frac{L}{P_i} + \frac{P_i - D_i}{P_i}\right) = \sum_{i=1}^{n} \frac{C_i}{P_i} + \sum_{i=1}^{n}\left(\frac{P_i - D_i}{L} \times \frac{C_i}{P_i}\right), \forall L > 0. \tag{4.4}$$

We study the part $\sum_{i=1}^{n}\left(\frac{P_i-D_i}{L} \times \frac{C_i}{P_i}\right)$ in Equation 4.4. In addition to $\sum_{i=1}^{n} \frac{C_i}{P_i}$, this part represents the CPU demand resulting from the fact that $D_i < P_i$. Note that for a task T_i, if $L < D_i$, $\left\lfloor \frac{L+P_i-D_i}{P_i} \right\rfloor C_i = 0$. This is because, in the interval $[0, L]$, no job of T_i has a critical time earlier than D_i, and thus there are no cycle demand from task T_i. If we assume from T_1 to T_n, $D_1 < D_2 < \cdots < D_n$, then only when $L \geq D_n$, each task can contribute to $\sum_{i=1}^{n}\left(\frac{P_i-D_i}{L} \times \frac{C_i}{P_i}\right)$. Further, it is easy to see that $\left(\frac{P_i-D_i}{L} \times \frac{C_i}{P_i}\right)$ monotonically decreases with the increase of L. Therefore, in Equation 4.4, $\sum_{i=1}^{n}\left(\frac{P_i-D_i}{L} \times \frac{C_i}{P_i}\right)$ has very limited contribution to the required frequency f, and the majority of CPU demand is consumed by $\sum_{i=1}^{n} \frac{C_i}{P_i}$. So we denote the aggregate CPU demand of $\mathbf{T}$ as $CPU_{dmd} = \sum_{i=1}^{n} \frac{C_i}{P_i}$.

We validate the efficiency of the definition of CPU_{dmd} through experimental comparison in Section 6.2.1. We observe in our experiments that estimating CPU_{dmd} as $\sum_{i=1}^{n} \frac{C_i}{D_i}$ is safe but very conservative, resulting in much less energy savings than the more aggressive estimation. Thus, we adopt the measurement $CPU_{dmd} = \sum_{i=1}^{n} \frac{C_i}{P_i}$, and propose the aggressive energy-conserving DVS approach for tasks with UAM arrivals in energy-bounded realtime systems. The procedure `decideFreq()` is shown in Algorithm 4.7.

In line 4–8 of Algorithm 4.7, we calculate CPU_{dmd} only with tasks whose SEL_i is *selected*. If the task-level flag SEL_i is *skipped*, the skipped task can be considered with zero actual execution cycles, enabling us to further reduce the speed. EBUA keeps track of the remaining computation cycles C_i^r. For the current time window P_i with a_i' instances, C_i^r is calculated as $C_i^r = \texttt{min}((a_i' - 1)c_i + c_i^r, (a_i - 1)c_i + c_i^r)$. Note that the actual number of jobs a_i' can be larger than the maximum job arrivals a_i, because there may be unfinished jobs from the previous time window P_i. But we only need to consider at most a_i instances of them. c_i^r is updated from line 5–9 of Algorithm 4.2.

From line 9 to line 19, EBUA considers the interval until the next task critical time and tries to "push" as much work as possible beyond the critical time. Similar to LaEDF [7], the algorithm considers the tasks in the latest-critical-time-first order in line 10. But since there may be more than one job of T_i in P_i, D_i^a is set to be the earliest invocation's absolute critical time. LaEDF [7] updates each task's D_i^a immediately when a task instance completes. In our DVS approach, we delay such an update until the next task instance is released, which results in additional energy savings.

Algorithm 4.7: `decideFreq()`

```
1:  input  : T, J_exe, t_cur;
2:  output : f_exe;
3:  CPU_dmd := 0;
4:  for T_i ∈ T do
5:      if SEL_i = selected then
6:          CPU_dmd := CPU_dmd + C_i/P_i;
7:      endif
8:  endfor
9:  s := 0;
10: for i = 1 to n, T_i ∈ {T_1, ··· , T_n | D_1^a ≥ ··· ≥ D_n^a} do
11:     if SEL_i = skipped then
12:         continue;
13:     endif
        /* reverse EDF order of tasks */
14:     CPU_dmd := CPU_dmd − C_i/D_i;
15:     x := max(0, C_i^r − (f_m − CPU_dmd) × (D_i^a − D_n^a));
16:     CPU_dmd := { f_m,                          if D_i^a − D_n^a = 0
                   { CPU_dmd + (C_i^r − x)/(D_i^a − D_n^a),   otherwise ;
17:     s := s + x;
18: endfor
19: f := min(f_m, s/(D_n^a − t_cur));
20: f_exe := selectFreq (f);
21: f_exe := max(f_exe, f^o_{T(J_exe)});
```

x counts the minimum number of cycles that a task must execute before the closest critical time, D_n^a, in order for it to complete by its own critical time (line 15), assuming worst-case aggregate CPU demand CPU_{dmd} by tasks with earlier critical times. In line 16, CPU_{dmd} is adjusted to reflect the actual demand of the task for the time after D_n^a, with the consideration

of the case that jobs of different tasks have the same termination times, which can occur, especially during overloads. s is simply the sum of the x values calculated for all of the tasks, and therefore reflects the minimum number of cycles that must be executed by D_n^a in order for the selected tasks to meet their critical times (line 17). In line 19, the CPU frequency is set just fast enough to execute s cycles over this interval.

Thus, `decideFreq()` capitalizes on early task completion by deferring work for future tasks in favor of scaling the current task. In line 16, we consider the case that jobs of different tasks have identical absolute critical times, which sometimes occurs, especially during overloads. In addition, during overloads, the required frequency may be higher than f_m and `selectFreq()` would fail to return a value. In line 19, we solve this by setting the upper limit of the required frequency to be f_m.

Finally, the result of `selectFreq()` is compared with $f^o_{T(J_{exe})}$ (line 21). The higher frequency is selected to provide the timeliness assurance—we cannot decrease f_{exe}, but may increase it to maximize the system-level energy efficiency.

4.6 Computational Complexity

To analyze the complexity of EBUA (Algorithm 4.2), we assume that the available number of CPU frequencies m is a constant with respect to the problem size (i.e., number of jobs, resources, etc.).

We consider n jobs in the ready queue and a maximum of r resources. In the worst case, `buildDep()` may build a dependency list with a length n; so the `for`-loop from line 10 to 17 requires $O(n^2)$ time. Also, the `for`-loop containing `calculateUER()` (line 18–20) can be repeated $O(n^2)$ times in the worst case. The complexity of procedure `sortByUER()` is $O(n \log n)$.

Complexity of the `for`-loop body starting from line 22 is dominated by `insertByECF()` (Algorithm 4.6). Its complexity is dominated by the `for`-loop (line 7–15, Algorithm 4.6), which requires $O(n \log n)$ time since the loop will be executed no more than n times and each execution requires $O(\log n)$ time to perform `insert()`, `remove()` and `lookup()` operations on the tentative schedule. Therefore, the worst-case complexity of the EBUA algorithm is $2 \times O(n^2) + O(n \log n) + n \times O(n \log n) = O(n^2 \log n)$.

As we mentioned before, the systems considered in this dissertation have a distinguishing feature, which is their relatively long execution time magnitudes—e.g., in the order of milliseconds to seconds, or seconds to minutes. Therefore, although EBUA has a higher algorithm overhead than EDF, which has a complexity of $O(n)$, we emphasize that the higher asymptotic cost of the algorithm and the consequent higher overhead are justified due to the long execution time magnitudes of the problem.

Chapter 5

Algorithm Properties

This chapter presents two classes of properties for EBUA. We first discuss several non-timeliness properties of EBUA, such as deadlock-freedom, correctness, and mutual exclusion. We then consider the timeliness properties of EBUA, as a real-time system fundamentally differs a non-real-time system in that a real-time system requires timeliness properties.

5.1 Non-Timeliness Properties

We now discuss EBUA's non-timeliness properties including deadlock-freedom, correctness, and mutual exclusion.

EBUA respects resource dependencies by ensuring that the job selected for execution can execute immediately. Thus, no job is ever selected for normal execution if it is resource-dependent on some other job.

Theorem 5.1. *EBUA ensures deadlock-freedom.*

Proof A cycle in the resource graph is the sufficient *and* necessary condition for a deadlock

in the single-unit resource request model. EBUA does not allow such a cycle by deadlock detection and resolution; so it is deadlock free. □

Lemma 5.2. *In `insertByECF()`'s output, all the dependents of a job must execute before it can execute, and therefore, must precede it in the schedule.*

Proof `insertByECF()` seeks to maintain an output queue ordered by jobs' critical times, while respecting resource dependencies. Consider job J_k and its dependent J_l. If $J_l.D$ is earlier than $J_k.D$, then J_l will be inserted before J_k in the schedule. If $J_l.D$ is later than $J_k.D$, $J_l.D$ is advanced to be $J_k.D$ with the operation with $CuCT$. According to the definition of `insert()`, after advancing the critical time, J_l will be inserted before J_k. □

Theorem 5.3. *When a job J_k that requests a resource R is selected for execution by EBUA, J_k's requested resource R will be free. We call this EBUA's correctness property.*

Proof From Lemma 5.2, the output schedule σ is correct. Thus, EBUA is correct. □

Thus, if a resource is not available for a job J_k's request, jobs holding the resource will become J_k's predecessors. We present EBUA's mutual exclusion property by a corollary.

Corollary 5.4. *EBUA satisfies mutual exclusion constraints in resource operations.*

5.2 Timeliness Properties

We first establish EBUA's assurance on energy-bounded systems.

Theorem 5.5. *EBUA assures that the consumed energy during a mission interval $[0, MT]$ never exceeds the energy bound E_{bnd}.*

Proof EBUA's dynamic monitoring of energy consumption described in Section 4.5.5 assures that, at any scheduling event between $[0, MT]$, the energy consumption of selected jobs never exceeds the allowed portion of E_{bnd}. Thus, the theorem holds. □

When $E_{bnd} \geq E_{rqd}$, with non-energy-bounded systems, EBUA has timeliness properties such as upper bounded time on accessing shared resources, and optimality during under-loads.

With Corollary 5.4, when a job needs to hold a resource, it must wait until no other job is holding the resource. A job waiting for an exclusive resource is said to be *blocked* on that resource. Otherwise, it can hold the resource and enter the the piece of code executed under mutual exclusion constraints, which is called a *critical section*. We first derive the maximum blocking time that each job may experience under EBUA.

Theorem 5.6. *Under EBUA for a non-energy-bounded system, a job J_k can be blocked for at most the duration of* `min`(n, m) *critical sections, where n is the number of jobs that could block J_k and have longer critical times than J_k has, and m is the number of resources that can be used to block J_k.*

Proof The operation of the procedure `insertByECF()` conforms to the Priority Inheritance Protocol (or PIP) [42]. In Algorithm 4.6, any job in $J_k.Dep$ with a later time constraint than $CuCT$ could block J_k, and it is required to meet $CuCT$, which is initially set to be $J_k.D$ (line 6). If, however, a dependent job has a tighter cricital time than $CuCT$, then it is scheduled to meet the tighter critical time, and $CuCT$ is advanced to that time since all jobs left in $J_k.Dep$ must complete by then. Note that in line 14, after insertion, the index of J_l is changed to $CuCT$. This is exactly the priority inheritance operation. Thus, the theorem immediately follows from properties of the PIP [42]. □

We also consider timeliness properties under no resource dependencies, where EBUA can be compared with a number of well-known algorithms. Specifically, the periodic model is a special case of UAM model. If $E_{bnd} \geq E_{rqd}$, with the conditions of (1) a non-energy-bounded system; (2) a set of periodic tasks with $\langle \overline{1}, P_i \rangle$ and step TUFs; and (3) there are sufficient processor cycles for meeting all task termination times—absence of CPU overloads (under Liu and Layland's condition [51]), we can establish EBUA's timeliness properties.

Theorem 5.7. *Under conditions (1), (2), and (3), a schedule produced by EDF [18] is also produced by EBUA, yielding equal total utilities. This is a critical time-ordered schedule.*

Proof We prove this by examining Algorithms 4.2 and 4.6. For a job J without dependencies, $J.Dep$ only contains J itself. For periodic tasks with step TUFs, a task's critical time is the same as its termination time. During non-overload situations, σ from line 24 of Algorithm 4.2 is termination time ordered.

The step TUF termination time that we consider is analogous to a deadline in [18]. As proved in [18, 51], a deadline-ordered schedule is optimal (with respect to meeting all deadlines) when there are no overloads. Thus, σ yields the same total utility as EDF. □

Some important corollaries about EBUA's timeliness behavior during under-loads can be deduced from EDF's optimality [18].

Corollary 5.8. *Under conditions (1), (2), and (3), EBUA always completes the allocated cycles of all tasks before their critical times, i.e., termination times.*

Corollary 5.9. *Under conditions (1), (2), and (3), EBUA minimizes the maximum lateness.*

EBUA also provides statistical performance assurances under possible conditions. With condition (2), the utility requirement of a task can only take $\nu = 0$ or $\nu = 1$. From Corollary 5.8, we can derive the properties of EBUA on performance assurances. With known height for each task's TUF, we can also derive the system-level utility assurance.

Theorem 5.10. *Under conditions (1), (2), and (3), EBUA meets the statistical timeliness performance requirements.*

Proof Under conditions (1), (2), and (3), the case of $\nu_i = 0$ is trivial, so we consider the meaningful case where $\nu_i = 1$ for each task. From Corollary 5.8, all the allocated cycles of tasks can be completed before their termination times. Furthermore, based on the results

of Equation 4.1, among the *actual* demanded processor cycles of task T_i's instances, at least ρ_i of them are less than the allocated cycles. Thus, for task T_i, EBUA can meet at least ρ_i termination times; i.e., EBUA accrues ν_i utility with a probability at least ρ_i. □

We also establish the relationship between task-level assurances and the system-level utility ratio in Theorem 5.11.

Theorem 5.11. *Under conditions (1), (2), and (3), if EBUA meets all statistical performance requirements, and a task T_i's TUF has the highest height U_i^{max}, then the system-level utility ratio, defined as the utility accrued by EBUA with respect to the system's maximum possible utility, is at least $\frac{\sum_{i=1}^{n} \rho_i \nu_i U_i^{max}}{\sum_{i=1}^{n} U_i^{max}}$.*

Proof We denote the number of jobs released by task T_i as m_i. Task T_i can accrue at least ν_i percentage of its maximum possible utility with the probability ρ_i. Thus, the system-level accrued utility to the system's maximum possible utility is $\frac{\rho_1 \nu_1 U_1^{max} m_1 + \cdots + \rho_n \nu_n U_n^{max} m_n}{U_1^{max} m_1 + \cdots + U_n^{max} m_n}$. Therefore, when m_i $(i = 1, \cdots, n)$ approaches $+\infty$, this formula becomes $\frac{\sum_{i=1}^{n} \rho_i \nu_i U_i^{max}}{\sum_{i=1}^{n} U_i^{max}}$. □

We also derive Theorem 5.10's counterpart for non-step and non-increasing TUFs in Theorem 5.12, where critical times are less than termination times. The proof for it can be found in [49].

Theorem 5.12. *In a non-energy-bounded system, for a set of independent periodic tasks, where each task is subject to a non-increasing TUF, the task set is schedulable and can meet all statistical timeliness requirements under the condition of Baruah, Rosier, and Howell [49].*

Note that for a set of independent periodic tasks, where each task is subject to a non-increasing TUF, $Load = \frac{1}{f_m} \sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1$ is also the sufficient condition for EBUA to meet all statistical timeliness requirements. This is because, with ν_i and ρ_i of task T_i, EBUA converts the performance assurance problem to the problem of meeting critical times. If $Load \leq 1$, according to the result of Theorem 5.10, the assertion holds.

But the condition of $Load \leq 1$ is only sufficient; actually, it is not the necessary condition. We illustrate this with an example in Chapter 6. Further, the condition of $Load \leq 1$ is more pessimistic than Theorem 5.12 in the sense of sufficiency [50]. Thus, we adopt Theorem 5.12 to describe the timeliness assurance property of EBUA with non-step and non-increasing TUFs.

Chapter 6

Experimental Results

In order to experimentally evaluate the performance of EBUA, we developed a simulator for the operation of hardware capable of DVS, and performed extensive simulations. We also study the overhead of EBUA without DVS, through an implementation on QNX Neutrino 6.2.1 RTOS [52]. In this chapter, we first present the simulation methodology in Section 6.1, and then discuss the simulation results in Section 6.2 and 6.3. Finally, we describe the results on overhead study in Section 6.4.

6.1 Simulation Methodology

Our simulator is written with the simulation tool OMNET++ [53], which provides a discrete event simulation environment. The simulator takes as input a task set, in which each task is specified with the time window P_i and its real-time requirements. The tasks' time constraints i.e., TUFs and the means/variances of the cycle demands are also specified as the input.

The energy consumption per cycle at a particular frequency is calculated using Equation 3.1. In practice, the S_3, S_2, S_1, and S_0 terms depend on the power management state of the system

and its subsystems. For example, if a laptop has its display on, the S_0 term will be large relative to the others. But if the display has been turned off, the S_0 term will be much smaller. Different types of systems will also have different relative values for the S terms. The S_3 term is probably a much larger fraction of the total power in a PDA than it is in a laptop [30, 31, 29].

We use experimental settings that are similar to those in Martin's PhD dissertation [30], but de-normalize the terms. For comparison, the experiments are carried out under three energy model settings, as shown in Table 6.1. From the table, in energy settings E_1, E_2, and E_3, the static term is 0%, 25%, and 50% of the total power at full speed f_m, respectively. Note that E_1 is the same as the traditional energy model, which only considers the energy consumed by the processor.

Table 6.1: Energy Model Settings

Energy Model	S_3	S_2	S_1	S_0
E_1	1.0	0	0	0
E_2	0.75	0	0	$0.25f_m^3$
E_3	0.5	0	0	$0.5f_m^3$

Other parameters that are supplied to the simulator include the processor specification. We consider a processor that supports seven different frequencies, {360, 550, 640, 730, 820, 910, 1000 MHz}. These frequencies reflect the setting that is available on a platform incorporating an AMD k6 processor with AMD's `PowerNow!` mechanism [54].

In our experiments, with a selected energy model, when a task is in execution, the energy consumption per cycle for each tested algorithms is calculated based on these energy settings. The total energy consumption is decided by the executed cycles. With this simplification, the task execution modeling can be reduced to counting cycles of execution, and execution traces are not needed.

The software-controlled halt feature, available on some processors and used for reducing energy expenditure during idle, is simulated by specifying an idle level parameter. We assume when a system is idling, it runs at the lowest available frequency. For simplicity, only task execution and idle (halt) cycles are considered. In particular, this does not consider preemption and task switch overheads or the time required to switch operating frequency or voltages. We assume such overheads can be incorporated into the task execution demands. There is no loss of generality from these simplifications. The preemption and task switch overheads are the same with or without DVS, so they have no effect on *relative* power dissipation. The voltage switching overheads incur a time penalty, which may affect the schedulability of some task sets, but only incur energy costs that are almost negligible, as the processor does not operate during the switching interval.

The real-time task sets are specified using parameters for each task, indicating its UAM settings and computation demands. We randomly generate two task sets. The first task set described in Section 6.1.1 contains only periodic and sporadic tasks, whose periods or minimum inter-arrival times are specified. The second task set described in Section 6.1.2 contains hybrid UAM tasks.

6.1.1 Task Sets for Task-Level Experiments

The tasks described in this section are periodic tasks and sporadic tasks. A sporadic task is an aperiodic task where consecutive jobs are separated by a minimum inter-arrival time [50].

For study of task-level assurances and other task-level performance metrics, we create simple task sets, which only contain several tasks selected from the ones defined in this section. We denote an experimental task set as G; the tasks contained in G are selected from Table 6.2.

In Table 6.2, the second column (♯ Jobs) shows the number of jobs generated for each task. The period or minimum inter-arrival time of each task is abbreviated as $P/I.A.$, and specified

in the third column. The table also summarizes these tasks' input TUFs. Tasks T_1 to T_4 have step TUFs, and the others' time constraints are specified by non-increasing and non-step TUFs, such as linear functions (those of T_6 and T_8) or right halves of parabolic functions (those of T_5 and T_7). Figure 6.1 shows the TUF shapes of task T_5, T_6, T_7, and T_8.

Table 6.2: Periodic and Sporadic Tasks

Task	♯ Jobs	$P/I.A.$	TUF
T_1	130	21	step, $height = 10$
T_2	124	22	step, $height = 80$
T_3	137	20	step, $height = 10$
T_4	109	25	step, $height = 80$
T_5	130	21	$\begin{cases} -0.025t^2 + 10, & 0 \leq t \leq 20 \\ 0, & \text{otherwise} \end{cases}$
T_6	124	22	$\begin{cases} -4t + 80, & 0 \leq t \leq 20 \\ 0, & \text{otherwise} \end{cases}$
T_7	137	25	$\begin{cases} -0.01t^2 - 0.15t + 10, & 0 \leq t \leq 25 \\ 0, & \text{otherwise} \end{cases}$
T_8	124	21	$\begin{cases} -0.5t + 10, & 0 \leq t \leq 20 \\ 0, & \text{otherwise} \end{cases}$
T_9	124	20	the same as T_8's
T_{10}	124	25	the same as T_8's

We change the tasks' cycle demands to change the system load (*Load*) as defined in Equation 4.3. For each task cycle demand Y_i, we generate normally-distributed values. The initial $E(Y_i)$ is normally distributed within the range $[f_m \times \frac{P_i}{10}, f_m \times P_i]$, and the initial $Var(Y_i)$ is set as $Var(Y_i) \approx 10 \times E(Y_i)$. Finally, according to the calculation of c_i in Section 4.3, the cycle demands $E(Y_i)$s are scaled by a constant k, and $Var(Y_i)$s are scaled by k^2; k is chosen such that the system load reaches a desired value.

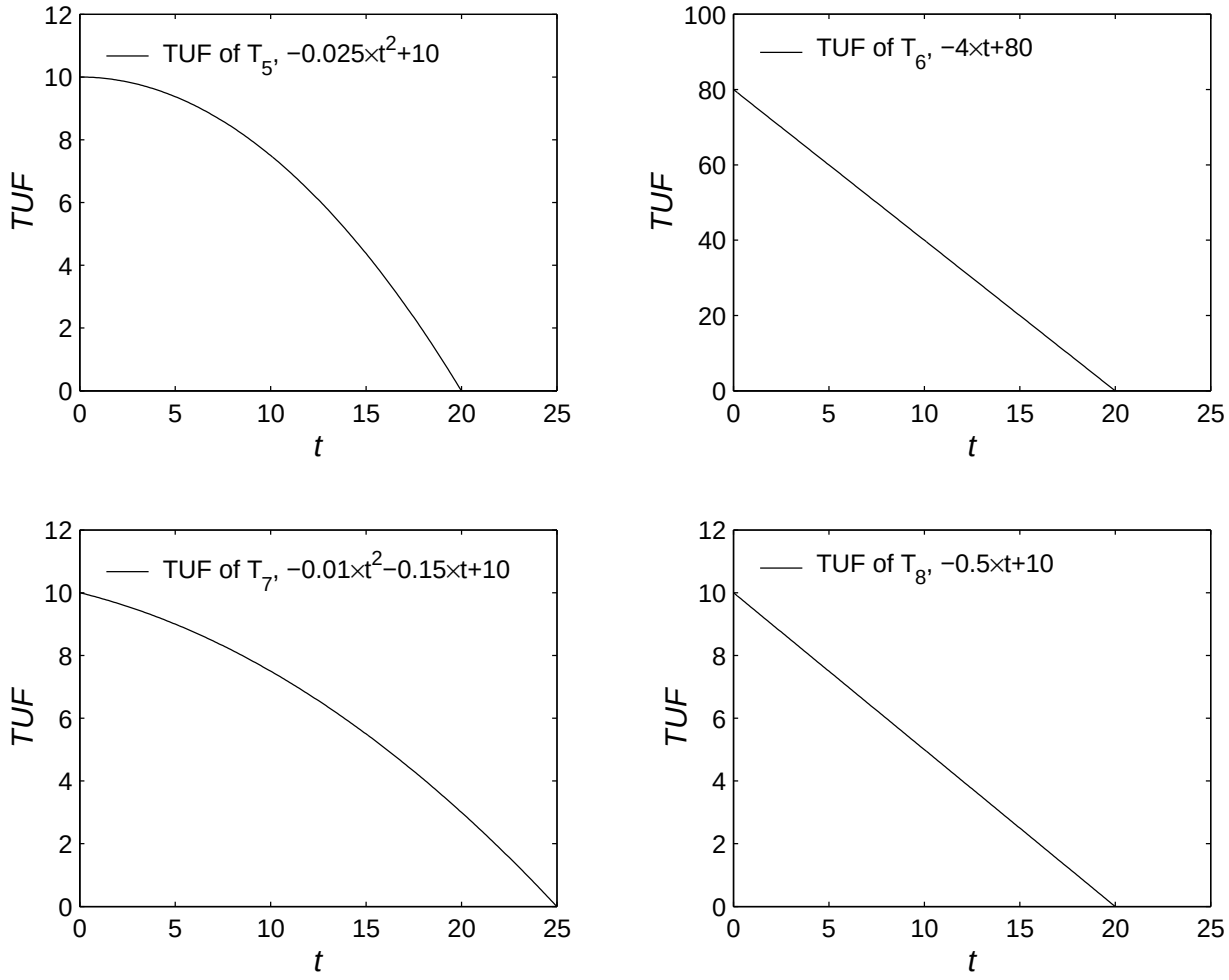


Figure 6.1: TUFs of Task T_5, T_6, T_7, and T_8 in Table 6.2

6.1.2 Hybrid Task Sets

We also generate task sets with more randomly selected parameters to conduct the simulation experiments. The tasks described in this section are UAM tasks with specified parameters $\langle a, P \rangle$. Note that when we generate periodic tasks, which are special cases of UAM tasks, we simply generate tasks with $\langle \overline{1}, P \rangle$.

We select task sets Gs with 10 to 50 tasks in three applications for our study. Their parameters are summarized in Table 6.3. Within each range, the time window P is uniformly

distributed, and a is the number of maximal possible job arrivals. The synthesized task sets simulate the varied mix of short and long time windows. The U^{max}s of the TUFs in A_1, A_2, and A_3 are uniformly generated within each range.

We decide which application a task T_i belongs to with the following method. Since all applications contain a total of $30 = 4 + 18 + 8$ tasks, we uniformly generate an integer between 1 and 30 for task T_i. If this integer is no larger than 4, T_i belongs to A_1; if it is in the range of $[5, 22]$, T_i belongs to A_2. Otherwise, T_i belongs to A_3.

For each task cycle demand Y_i, we generate normally-distributed values. The initial $E(Y_i)$ is normally distributed within the range $[f_m \times \frac{P_i}{10}, f_m \times P_i]$, and the initial $Var(Y_i)$ is set as $Var(Y_i) \approx 10 \times E(Y_i)$. Finally, according to the calculation of c_i in Section 4.3, the cycle demands $E(Y_i)$s are scaled by a constant k, and $Var(Y_i)$s are scaled by k^2; k is chosen such that the system load reaches a desired value.

Table 6.3: Task Settings

Applications	♯ tasks	UAM $\langle a, P \rangle$	U^{max}
A_1	4	$\langle 5, 22\text{–}28 \rangle$	$[50, 70]$
A_2	18	$\langle 8, 50\text{–}70 \rangle$	$[300, 400]$
A_3	8	$\langle 3, 2.4\text{–}9.6 \rangle$	$[1, 10]$

This type of method to generate real-time task sets has been used previously in the development and evaluation of the real-time embedded micro-kernel in [55] and the DVS approach in [12]. By means of generating a large number of distinct task sets with different task set loads, the simulations provide a relationship of energy consumption and accrued utility to the system load.

6.2 Experiments on Non-Energy-Bounded Systems

We first consider the case where during the mission time $[0, MT]$, the system has enough energy, i.e., $E_{bnd} \geq E_{rqd}$. For non-energy-bounded systems, EBUA can be compared with other energy-efficient real-time scheduling algorithms. In addition to EBUA, we implemented the following schemes for comparison: BaseEDF, LaEDF, StaticEDF, and LaEDF-NA.

BaseEDF is the EDF scheduler without any DVS support and uses the highest frequency. LaEDF is the Look-ahead RT-DVS for EDF scheduler in [7]. StaticEDF uses the constant speed given by Equation 4.2 and a "ceiling" up to the lowest suitable frequency in $\{f_1, f_2, \cdots, f_m\}$. StaticEDF switches to the lowest frequency whenever there is no ready task. Combining the static schemes in [3] and [7], when each task presents its worst-case workload to the processor at every instance, StaticEDF is the static optimal solution to the DVS problem for the periodic task model with step TUFs under the available frequency set. The previous three schemes abort infeasible tasks during overloads. Thus, LaEDF-NA is LaEDF with no abortion.

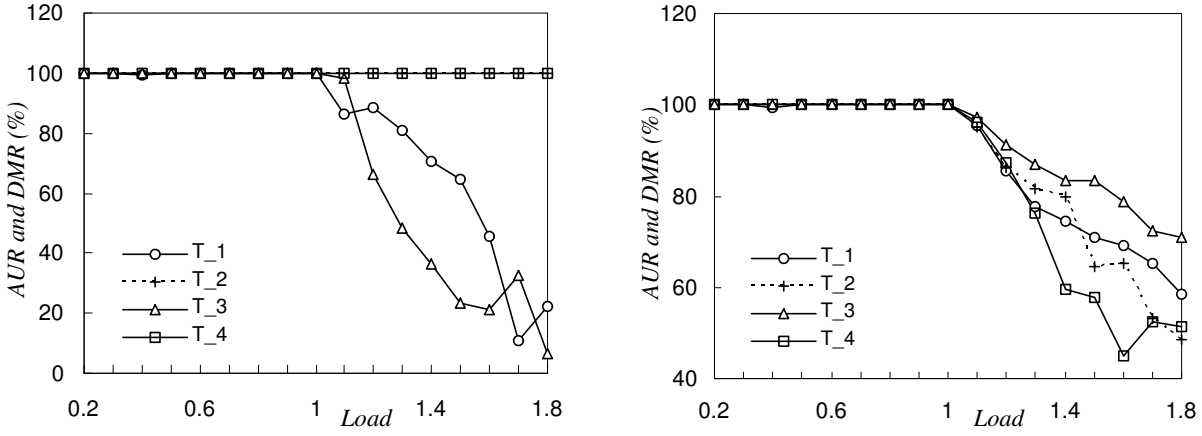
LaEDF, LaEDF-NA, and StaticEDF perform DVS on periodic tasks with known worst-case workload, which is unavailable in our application model. Thus, we use the minimum inter-arrival time and cycles allocated by EBUA as their inputs.

6.2.1 Performance Assurance

In our first set of simulation experiments, we evaluate the statistical performance assurances provided by EBUA. We first select tasks described in Section 6.1.1, and consider the task set $G_1 = \{T_1, T_2, T_3, T_4\}$ with the performance requirement of $\{(\nu_i = 1, \rho_i = 0.96), i = 1, \cdots, 4\}$. We consider downward step TUFs, since all the other algorithms compared can only deal with deadlines. As no strategies except EBUA consider the system-level energy consumption,

we only use the energy model E_1 in these experiments.

Figure 6.2(a) and 6.2(b) show the accrued utility ratio (AUR) and critical-time meet ratio (DMR) of each task under increasing *Load*. AUR is the ratio of accrued aggregate utility to the maximum possible utility, and DMR is the ratio of the jobs meeting their critical times to the total job releases of a task. For a task with a downward step TUF, its AUR and DMR are identical; so we show them in one plot. Note that the system-level AUR and DMR can be different due to the mix of different utility of tasks.



(a) EBUA

(b) LaEDF

Figure 6.2: AUR and DMR vs. *Load* of G_1 under E_1 and Step TUFs

As Figure 6.2(a) shows, with EBUA during under-loads, all tasks accrue 100% AUR and DMR, except task T_1, whose AUR and DMR is 99.23% at $Load = 0.3$. Thus, EBUA delivers the statistical performance assurance of being able to accrue 100% of task maximum utility with a probability at least 96% for all tasks. This also validates Theorem 5.10.

Comparing the results during overloads in Figure 6.2(a) and 6.2(b), we observe that EBUA still achieves near 100% AUR and DMR of task T_2 and T_4, but achieves less AUR and DMR of T_1 and T_3. One the other hand, LaEDF decreases the AUR and DMR of T_2 and T_4 more than the other two. This is because, T_2 and T_4 have TUFs with higher "heights" and thus

higher utility; so EBUA accrues more system-wide utility by completing these tasks before their termination times. Schemes based on EDF cannot make such scheduling decisions—T_2 and T_4 are not favored by LaEDF since they have longer critical times than T_1 and T_3. We show the comparison of utility accrual for various schemes in Section 6.2.3.

Figure 6.3 shows the system-level AUR and DMR of EBUA and LaEDF under increasing *Load*. According to Theorem 5.11, with large number of job releases, the system-level AUR should also be 96%. We observe that the AUR and DMR of EBUA during under-loads are above 98%. This validates Theorem 5.11. Further, EBUA accrues much more system-wide utility during overloads than LaEDF—although its DMR is lower than that of LaEDF, EBUA can obtain much higher AUR than LaEDF in such cases.

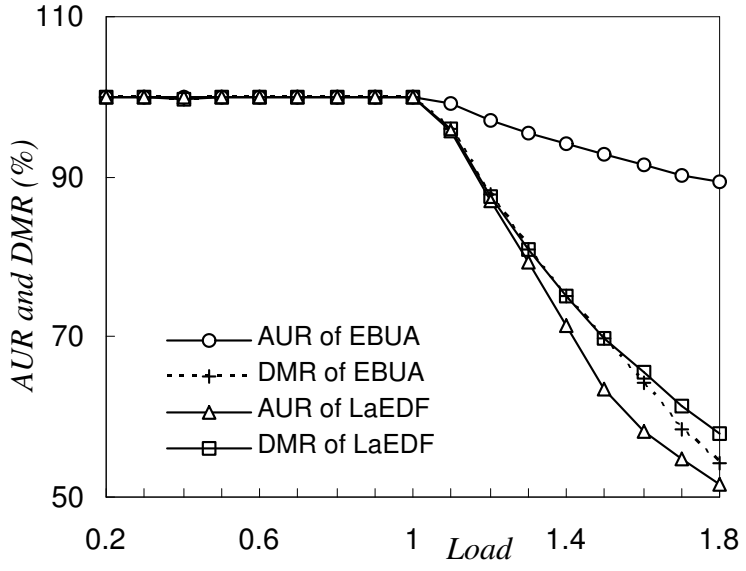
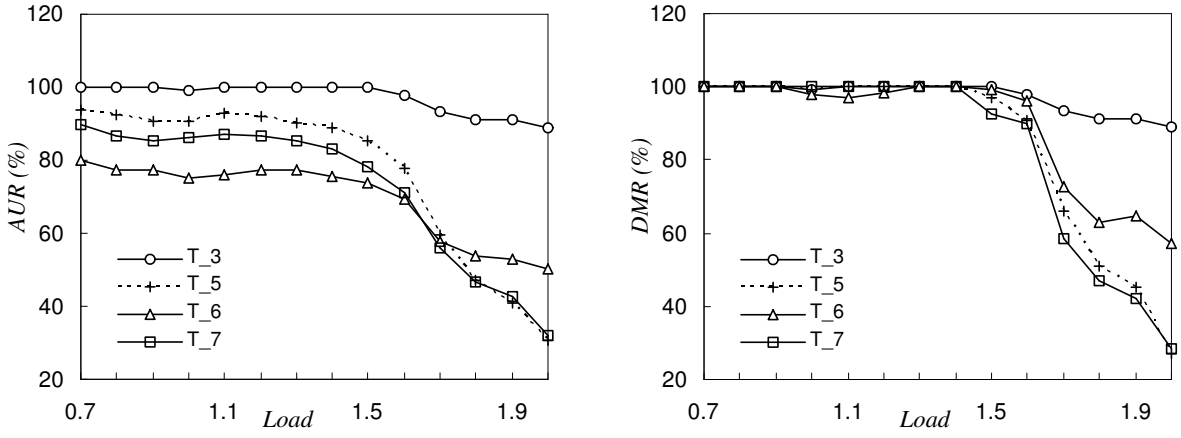


Figure 6.3: System-Level AUR and DMR vs. *Load* of G_1 under E_1 and Step TUFs

Besides G_1, we also consider the task set $G_2 = \{T_3, T_5, T_6, T_7\}$ that contains linear-shaped and parabolic-shaped TUFs (with non-increasing portion) as well as step TUFs. The performance requirements of G_2 are $\{(\nu_3 = 1.0, \rho_3 = 0.80), (\nu_5 = 0.55, \rho_5 = 0.80), (\nu_6 = 0.5, \rho_6 = 0.80), (\nu_7 = 0.55, \rho_7 = 0.80)\}$. Apparently with the task set G_2, $D_i < P_i$.

Figure 6.4 shows the AUR and DMR of each task in G_2 with *Load* varying from 0.7 to 2.0. As mentioned in Equation 4.3, *Load* is defined as $\frac{1}{f_m}\sum_{i=1}^{n}\frac{C_i}{D_i}$, while Equation 4.2 states $CPU_{dmd} = \sum_{i=1}^{n}\frac{C_i}{P_i}$. When $D_i < P_i$, according to the analysis in Section 4.5.7 starting from Page 39, the *Load* definition is conservative, while CPU_{dmd} definition is aggressive, which means the actual dynamic CPU utilization calculated from processor demand approach should be between $\frac{CPU_{dmd}}{f_m} = \frac{1}{f_m}\sum_{i=1}^{n}\frac{C_i}{P_i}$ and *Load*. We list the corresponding values of $\frac{CPU_{dmd}}{f_m}$ for this task set in Table 6.4.



(a) AUR vs. *Load*

(b) DMR vs. *Load*

Figure 6.4: AUR and DMR vs. *Load* of G_2 under E_1 and Non-Step TUFs

We consider task T_7 as an example to illustrate how EBUA delivers statistical performance assurances for non-step TUFs. As shown in Figure 6.4, when $Load \leq 1$, task T_7 is assured to accrue at least $\nu_7 = 55\%$ of its maximum utility with a probability no less than $\rho_7 = 80\%$. For example, at $Load = 1$, EBUA accrues AUR=86.28% and DMR=100%, which implies that it can complete all the demanded cycles of the task before their critical times. Furthermore, 86.28% of the maximum utility can be accrued at a probability 100%—this is much higher than the timeliness requirements.

But $Load \leq 1$ is not the necessary condition for delivering statistical performance assurances.

For example, at $Load = 1.6$ and $\frac{CPU_{dmd}}{f_m} = 1.01$, task T_7 can still accrue AUR=71.21% and DMR=89.91%. This is because, for a task with a non-step and non-increasing TUF, even if the task misses its critical time, the task can complete before its termination time and accrue some amount of utility, which depends on the TUF shape. Therefore, these experiments validate Theorem 5.12.

Table 6.4: $Load$ and $\frac{CPU_{dmd}}{f_m}$

$Load$	0.7	0.8	0.9	1.0	1.1	1.2	1.3
$\frac{1}{f_m} \times CPU_{dmd}$	0.44	0.5	0.57	0.6	0.7	0.76	0.83
$Load$	1.4	1.5	1.6	1.7	1.8	1.9	2.0
$\frac{1}{f_m} \times CPU_{dmd}$	0.89	0.95	1.01	1.06	1.13	1.2	1.26

Another major pattern that can be observed from Figure 6.4 is that, as $Load$ and $\frac{CPU_{dmd}}{f_m}$ increases, task T_3 with a step TUF accrues more AUR and DMR than the other tasks with non-step TUFs. This is because, T_3's full utility can be accrued as long as it is completed before its termination time, while completing other tasks just before their termination times may result in very low utility. In addition, among tasks T_5, T_6, and T_7 with non-step TUFs, the one with the highest maximum utility i.e., T_6, is favored by EBUA to accrue more system-wide utility. This is demonstrated by its higher AUR (Figure 6.4(a)) and DMR (Figure 6.4(b)) than those of the other two tasks.

It would also be interesting to observe the system-level AUR and DMR of EBUA with increasing $Load$ in Figure 6.5. Due to non-step TUFs of the task set, although when $Load \leq 1.5$ DMR is close to 100%, AUR is much lower than that. Another important fact is that, only after $Load \geq 1.6$ and the corresponding $\frac{CPU_{dmd}}{f_m} \geq 1.01$, AUR and DMR start to decrease dramatically. This implies that approximately when CPU_{dmd} is larger than the sysem's processing capacity f_m, the system-level timeliness decreases. This validates the efficiency of the definition of CPU_{dmd} as $\sum_{i=1}^{n} \frac{C_i}{P_i}$—as we mentioned in Section 4.5.7, estimating CPU_{dmd}

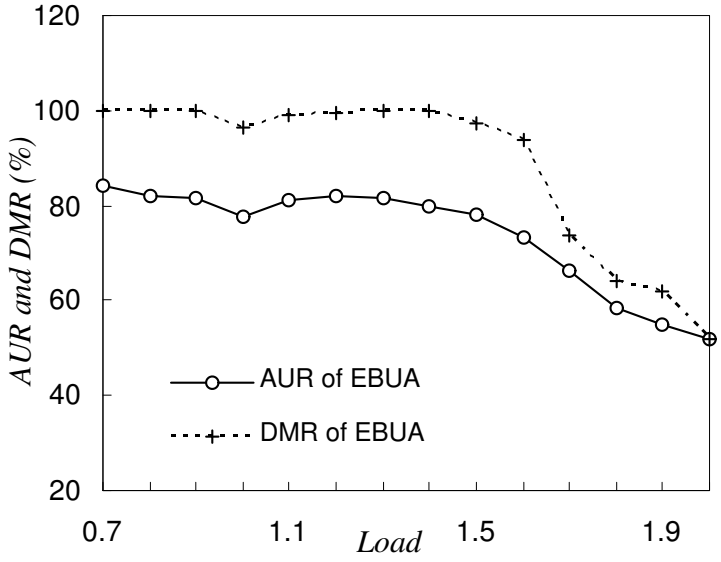


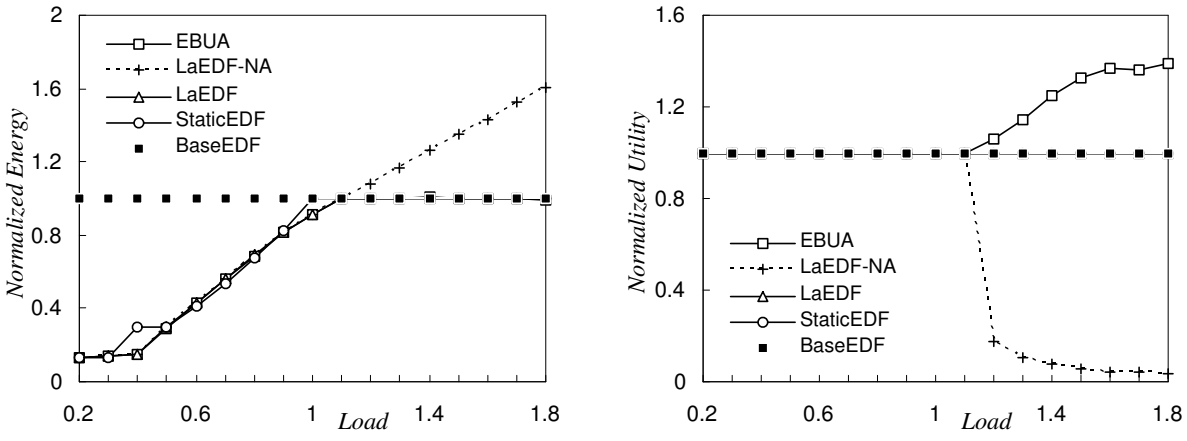
Figure 6.5: System-Level AUR and DMR vs. *Load* of G_2 under E_1 and Non-Step TUFs

as $\sum_{i=1}^{n} \frac{C_i}{D_i}$ is safe but very conservative, resulting in much less energy savings than the more aggressive estimation. Thus, our aggressive energy-conserving DVS approach is designed based on the measurement $CPU_{dmd} = \sum_{i=1}^{n} \frac{C_i}{P_i}$.

6.2.2 Impact of Energy Models

In the next experiment, we consider hybrid task sets and determine the system-level effects of our new energy model. We consider the task set G containing tasks described in Section 6.1.2, and apply different schemes on G under different energy settings. We consider downward step TUFs, since all the other algorithms compared can only deal with deadlines. Each task T_i has the statistical performance requirement of $\nu_i = 1$ and $\rho_i = 0.96$.

Figure 6.6, 6.7, and 6.8 show the energy consumption and the accrued utility normalized to those of BaseEDF under energy models E_1, E_2, and E_3, respectively, as *Load* varies from 0.2 to 1.8.



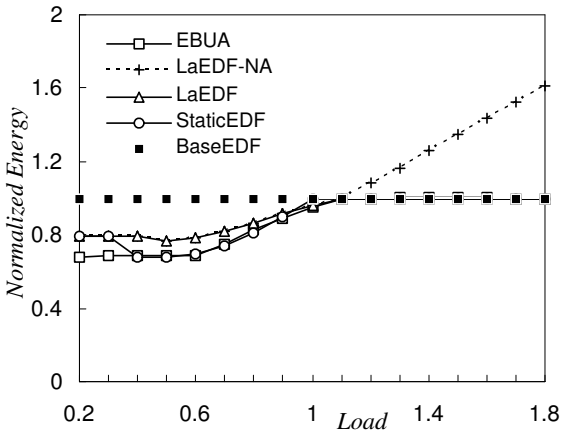
(a) Normalized Energy vs. *Load*

(b) Normalized Utility vs. *Load*

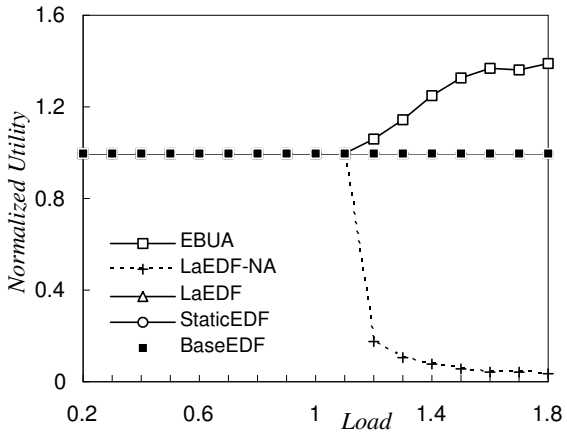
Figure 6.6: Normalized Energy and Utility vs. *Load* with Step TUFs under Energy Setting E_1

In Figure 6.6(a), 6.7(a) and 6.8(a), during under-loads, we observe that EBUA saves more energy than the other schemes. As the term S_0 in the energy model increases from E_1 to E_3, EBUA adjusts the selected frequency to adapt to the energy setting, and we observe more and more pronounced energy saving effects of EBUA. For example, in Figure 6.8(a), under E_3 when LaEDF, LaEDF-NA, and StaticEDF perform worse in terms of energy savings than BaseEDF, EBUA still outperforms BaseEDF during under-loads. During overloads, the results of all schemes except LaEDF-NA converge to 1. This is because all the algorithms select the highest frequency by DVS calculation during overloads. The energy consumption of LaEDF-NA increases linearly with *Load*, because it does not abort jobs and executes all jobs that arrive.

On the other hand, from Figure 6.6(b), 6.7(b), and 6.8(b), we observe that during under-loaded situations, all schemes accrue the same (optimal) utility because of EDF's optimality [56] during such situations. But during overload situations, LaEDF-NA suffers domino effects and accrues almost no utility [23]. During overloads, EBUA seeks to schedule jobs with higher UERs, and thus accrues remarkably higher utility than the others.



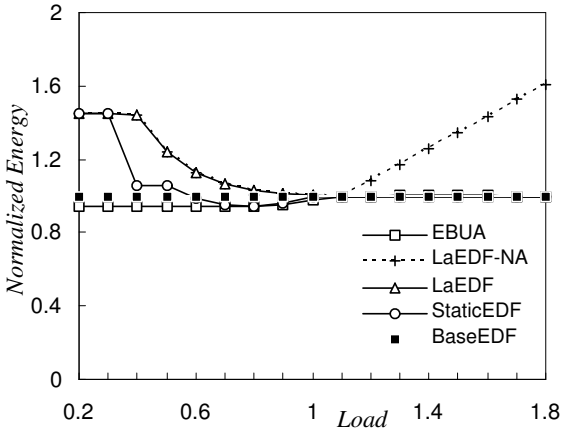
(a) Normalized Energy vs. *Load*



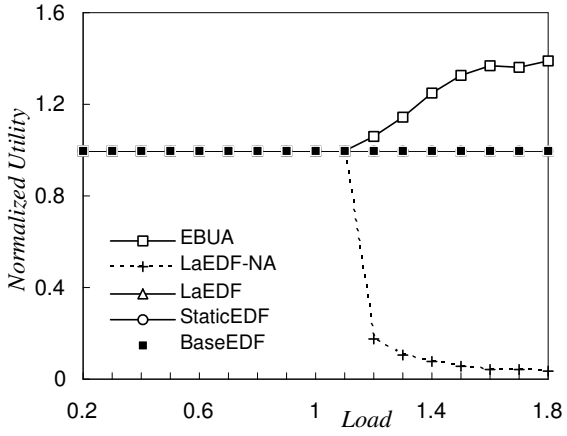
(b) Normalized Utility vs. *Load*

Figure 6.7: Normalized Energy and Utility vs. *Load* with Step TUFs under Energy Setting E_2

Thus, the performance gap demonstrates that EBUA accrues higher utility during overloads, and handles the dual criterion problem during under-loads—saving energy while achieving the given utility within the given time constraint. Furthermore, it shows that the scheduling metric UER is effective in integrating timeliness and energy consumption.



(a) Normalized Energy vs. *Load*



(b) Normalized Utility vs. *Load*

Figure 6.8: Normalized Energy and Utility vs. *Load* with Step TUFs under Energy Setting E_3

Since no strategies except EBUA consider the system-level energy consumption, we only use the energy model E_1 in the further simulation experiments of the rest of Section 6.2.

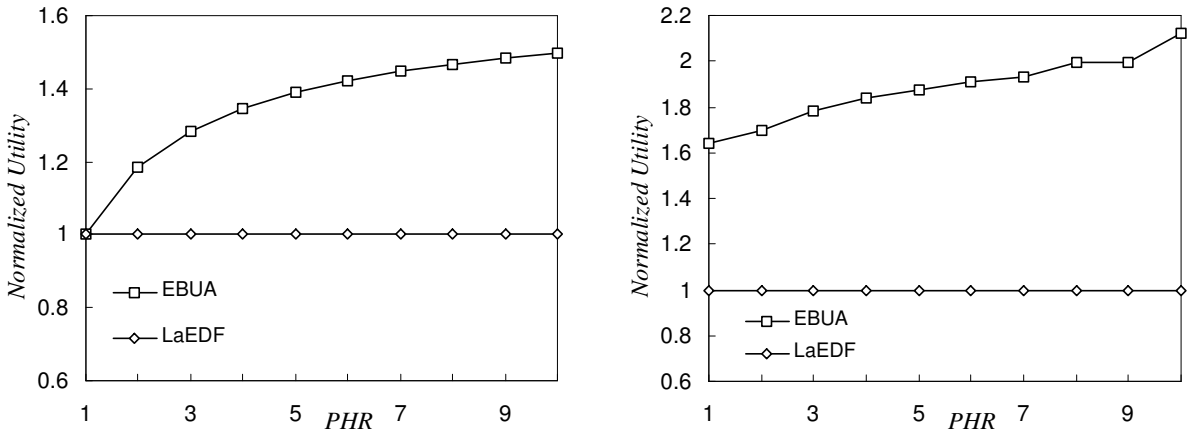
6.2.3 Effectiveness of Utility Accrual

From experiments of the previous sections, we observe that EBUA mimics the behavior of EDF during under-loaded situations. During overloads, all schemes tend to select f_m as the execution frequency by DVS, and thus have the same energy consumption. But EBUA seeks to accrue more utility during such situations than the other strategies. In this section, we vary the TUF shape of each task to demonstrate EBUA's utility accrual capability.

In this experiment, we consider simple task sets with tasks from Section 6.1.1. Therefore, we roughly define the ratio of the maximum and minimum heights of TUFs in a task set as *peak height ratio* (or PHR). We consider two task sets G_3 and G_4 with step TUFs and linear TUFs, respectively. G_3 is the set $G_3 = \{T_1, T_2, T_3, T_4\}$, where the heights of U_2 and U_4 are varied from 10 to 100. G_4 is the set $G_4 = \{T_6, T_8, T_9, T_{10}\}$, where the crossing points of the *utility*-axes with U_6 and U_{10} are varied from 10 to 100. In addition, the intersections with the t-axes of all TUFs in G_4 are maintained at $t = 20$. Thus, both G_3 and G_4 have PHRs varying from 1 to 10.

Figure 6.9(a) shows the accrued utilities for EBUA and LaEDF that are normalized to that of LaEDF under G_3 with $Load = 1.5$. During overloads, LaEDF, StaticEDF, and BaseEDF yield the same performance, so we only show LaEDF here. We observe that, at $PHR = 1$, EBUA makes the same scheduling decisions as LaEDF. But as PHR increases, EBUA obtains higher system-level utility than LaEDF.

Figure 6.9(b) shows the normalized utilities for EBUA and LaEDF under G_4 with $Load = 1.85$ and $\frac{CPU_{dmd}}{f_m} = 1.5$. We observe similar trends as those in Figure 6.9(a), but with larger performance gap as PHR increases. The two strategies' different scheduling criteria result



(a) G_3: Step TUFs

(b) G_4: Linear TUFs

Figure 6.9: Normalized Utility vs. PHR under E_1

in different performance even at $PHR = 1$.

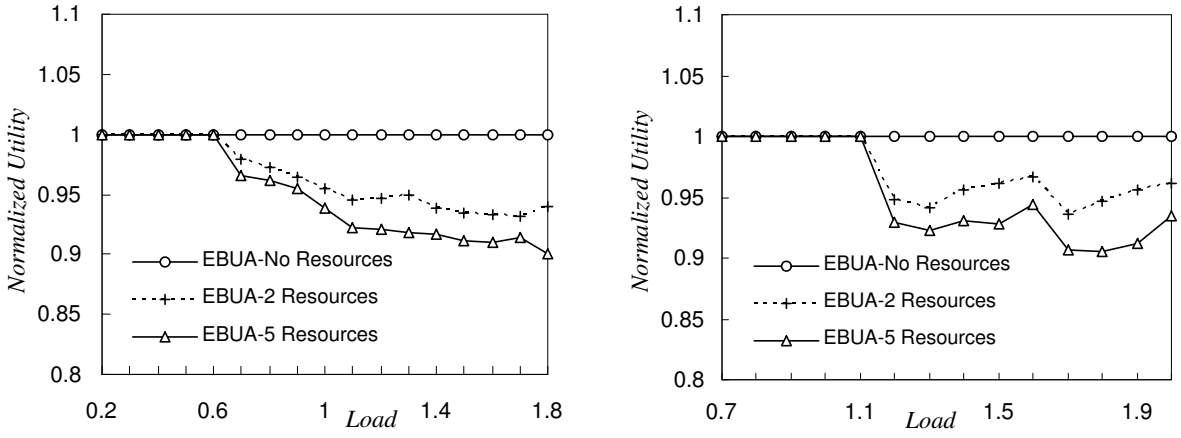
Since not all critical times can be satisfied during overloads, EBUA considers the UER of each job and seeks to schedule jobs with high UERs while maintaining the critical time order of jobs at the same time. But LaEDF simply schedules according to tasks' critical times, and conforms to the critical time order. In addition, during overloads, EBUA tends to abort jobs with low UERs in the feasibility check. This results in higher system-level utility than that obtained by LaEDF, which always aborts jobs with the largest critical time.

6.2.4 Results under Resource Dependency

To construct dependent task sets, we consider task sets G_1 and G_2 and have each job randomly request and release resources from some available set of resources during the job's life cycle. The resource request and release times are uniformly distributed within a job's life cycle.

We conducted experiments on the task sets, which are scheduled by EBUA under no re-

sources, two shared resources, and five shared resources. Figure 6.10(a) shows utilities normalized to the case of G_1, as *Load* varies from 0.2 to 1.8. Figure 6.10(b) shows the same metric for G_2, as *Load* varies from 0.7 to 2.0.



(a) Utility vs. *Load* of G_1

(b) Utility vs. *Load* of G_2

Figure 6.10: Normalized Utility with Resource Dependencies under E_1

From the figures, we observe that when *Load* increases, the performance of EBUA on dependent task sets decreases. The higher the number of shared resources, the more performance decrease can be observed. This is because EBUA respects resource dependencies in scheduling, which may bring additional blocking times to tasks. Such additional blocking times increase the sojourn times of jobs, resulting in decreased utilities. In the worst-case, resource dependencies may cause jobs to be executed in the reverse order of UERs or critical times. So with dependent task sets, EBUA cannot provide performance assurances and suffers utility losses, especially during high loads.

However, at very high *Load* and with five shared resources, normalized utilities of EBUA on the independent task sets are just better than those on dependent task sets by no more than 10%. This is because EBUA aborts a task when its expected completion time is less than its termination time. Thus, for the periodic or sporadic task sets we used in this section, the

job queue seen by the EBUA scheduler at any scheduling event has a length no more than the number of tasks. With our experimental settings, we have only limited performance loss in our simulation, but we expect more performance drop with larger and more complicated task sets.

We have finished studying the task-level performance of EBUA. In the rest experiments, we consider more complicated task set G, which consists of tasks described in Section 6.1.2.

6.2.5 Performance with Non-Increasing TUFs and UAM

We now consider non-step and non-increasing TUFs with EBUA, and study the impact of UAM model on the energy consumption. We allocate a linear TUF to each task, and its slope is calculated as $-\frac{U^{max}}{P}$, where P is the time window. We set $\{\nu_i = 0.3, \rho_i = 0.9\}$ to each task, and use the energy model E_1 in the experiments of this section.

We change the parameter a_i in the UAM model $\langle a_i, P_i \rangle$ for each task from 1 to 3, and run EBUA on the task set. Thus, the maximum job arrivals in each time window P_i is increased from 1 to 3. Figure 6.11 shows the energy consumption of EBUA under different system loads. The energy consumption is normalized to the results of EBUA without DVS, which always selects f_m.

We observe that, during overloads, the normalized energy consumption does not change with a_i, because even with DVS, EBUA tends to select the highest frequency. However, during under-loads at $Load = 0.2$, 0.5, and 0.8, as a_i increases, EBUA's energy consumption increases, even under the same $Load$. For example, when $Load = 0.5$, the normalized energy consumption of $\langle 1, P \rangle$ is 0.26; that of $\langle 2, P \rangle$ is 0.41, and with $\langle 3, P \rangle$, this number increases to 0.61. This is because DVS is dependent on the prediction of future workload and slack time estimation. Our DVS approach `decideFreq()` (Algorithm 4.7) uses $CPU_{dmd} = \sum_{i=1}^{n} \frac{C_i}{P_i}$, where $C_i = a_i c_i$. The parameter a_i only describes the number of maximum possible job

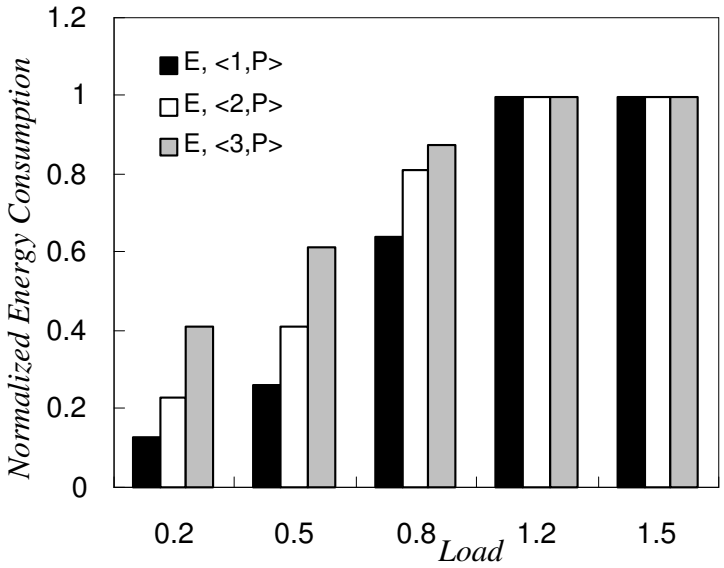


Figure 6.11: Energy Consumption of Different UAM Settings

arrivals in the time window P_i. Thus, CPU_{dmd} is derived from *the worst case* arrivals. When a_i increases, the actual number of arrivals can be any integers in the range of $[0, a_i]$. Therefore, more complicated job arrivals negatively affect the accurate estimation of slack times, and thus decrease the energy saving from DVS.

6.3 Experiments on Energy-Bounded Systems

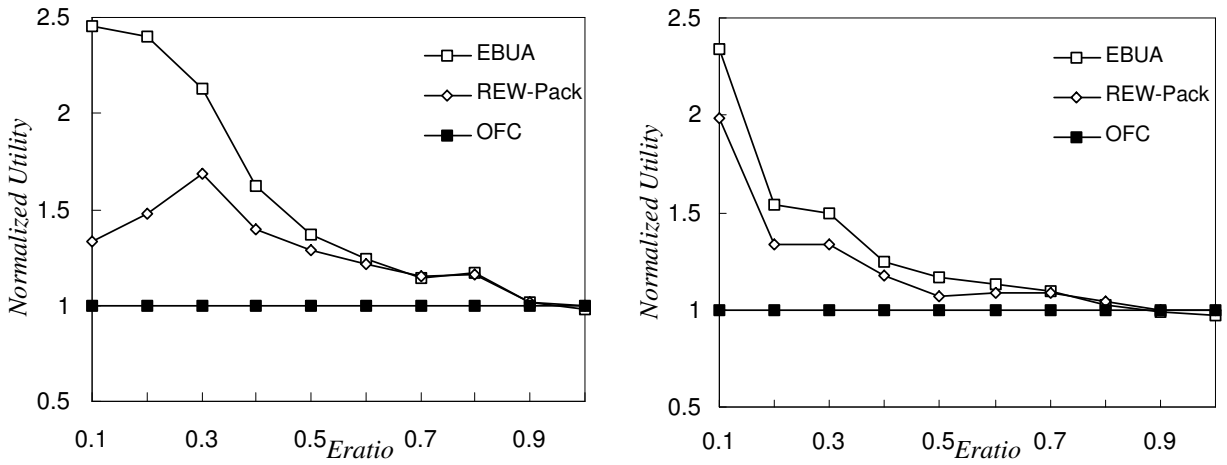
In this section, we consider the energy-bounded system, i.e., $E_{bnd} \leq E_{rqd}$ during the mission time $[0, MT]$. To evaluate the energy efficiency of EBUA, we perform experiments to compare EBUA with other energy-bounded algorithms, i.e., OFC [37] and REW-Pack [35]. OFC statically (off-line) calculates each task's expected energy consumption during the mission time, and selects tasks with the heuristic of Larger Reward Density (LRD), based only on the worst-case workload information. REW-Pack works for frame-based or periodic tasks. In all experiments of this section, we consider task set G with tasks described in Section 6.1.2.

Note that we can set parameters $\langle a = \overline{1}, P \rangle$ in Table 6.3 to generate periodic tasks.

6.3.1 Performance with Step TUFs

We first evaluate the performance with step TUFs, so that EBUA can be compared with the other strategies. We set $\{\nu_i = 1, \rho_i = 0.96\}$, and apply different schemes on independent periodic task sets under different energy settings. We vary *Eratio*, which is defined as the ratio of E_{bnd} to E_{rqd}, from 0.1 to 1.0, and show the accrued utility at $Load = 0.7$ and $Load = 1.5$, respectively. Note that REW-Pack requires the hyper-period of periodic tasks, which can be very large due to our synthesized task sets, so we approximate it.

When $Eratio < 1$, the system is energy-bounded; when $Load > 1$, the system is CPU overloaded. Thus, by changing *Eratio* and *Load*, we can generate interesting scenarios to study the tradeoffs between energy and utility.



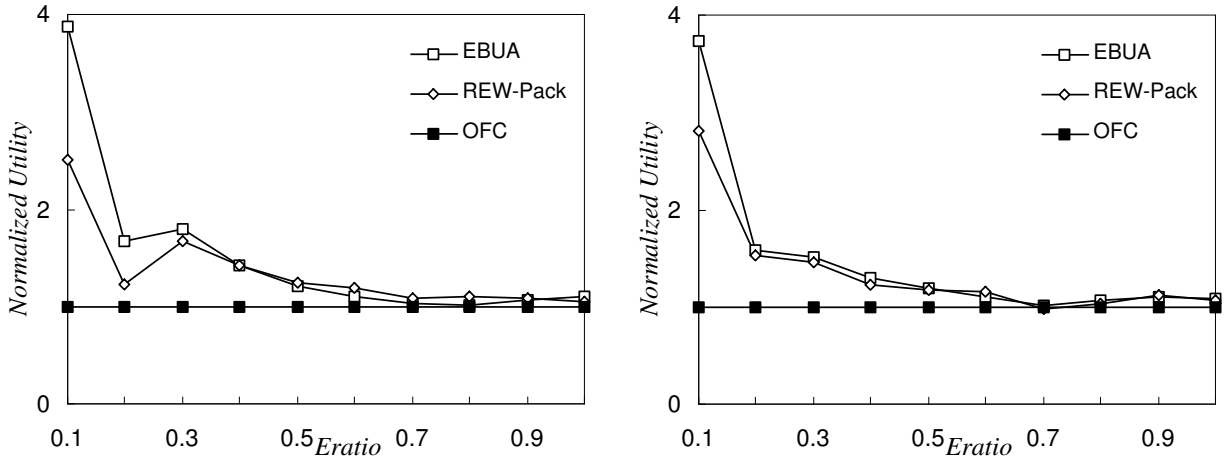
(a) $Load = 0.7$, E_1 (b) $Load = 0.7$, E_3

Figure 6.12: Normalized Utility vs. *Eratio* during under-loads under E_1 and E_3

Figure 6.12 shows the utilities normalized to those of OFC under energy settings E_1 and E_3. It shows the performance of different strategies in the scenario of CPU under-loads

($Load = 0.7$) in an energy-bounded system. From Figure 6.12(a) and Figure 6.12(b), we observe that when E_{bnd} is low, the system has a strict energy bound and it is crucial to utilize excess energy due to early completions. Hence, under such conditions the difference in performance is significant. As E_{bnd} increases, the system becomes less energy-bounded. The schemes converge when $Eratio = 100\%$, because the system has enough energy to meet all the critical times and accrue its maximum utility, due to EDF's optimality [18] in such cases.

Figure 6.13(a) and Figure 6.13(b) show the scenarios of CPU overloads ($Load = 1.5$) in an energy-bounded system, under energy settings E_1 and E_3, respectively. Plots in Figure 6.13 bear the same trends as those in Figure 6.12. But when $Eratio = 100\%$, the plots in Figure 6.13 do not converge. This is because, although the system is not energy-bounded, during CPU overloads different schemes still show different abilities in utility accrual.



(a) $Load = 1.5$, E_1 (b) $Load = 1.5$, E_3

Figure 6.13: Normalized Utility vs. $Eratio$ during overloads under E_1 and E_3

In Figure 6.12 and Figure 6.13, E_{bnd} is recalculated as a percentage of E_{rqd}, which is also a function of the system load. In our next set of experiments, E_{bnd} is set to a fixed value, namely the energy required to meet all the critical times when $Load = 0.7$. We represent this

value as $E_{rqd}\,(Load = 0.7)$. Figure 6.14 shows the normalized utilities of the three schemes with $E_{bnd} = E_{rqd}\,(Load = 0.7)$, when *Load* varies from 0.2 to 1.8 under energy setting E_1.

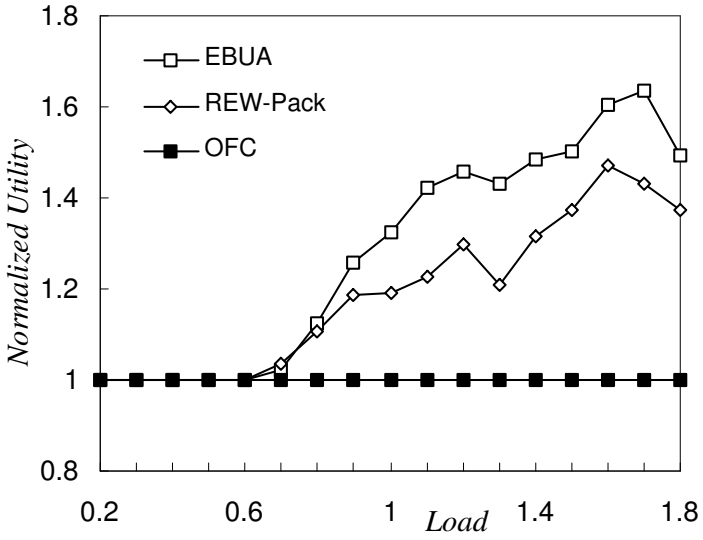


Figure 6.14: Utility vs. *Load* with Fixed E_{bnd} under E_1

Figure 6.14 shows more complicated combinations of system energy requirement and CPU loads. When $Load \leq 0.7$, the system has enough energy to meet all critical times (i.e., $E_{bnd} \geq E_{rqd}$, and system is under-loaded). Therefore, all schemes yield the same utility. As *Load* increases beyond 0.7 but below 1.0, the system becomes effectively more energy-bounded, but is still under-loaded. When *Load* exceeds 1.0, the system is both energy-bounded and overloaded. The performance gap with the increase of *Load* shown in Figure 6.14 demonstrates that EBUA accrues higher utility with fixed energy bound.

6.3.2 Performance with Non-Increasing TUFs and UAM

We then consider non-step and non-increasing TUFs, and UAM tasks with EBUA and EUA∗ [38]. each task is allocated a linear TUF, and its slope is calculated as $-\frac{U^{max}}{P}$, P being the time window. We set $\{\nu_i = 0.3, \rho_i = 0.9\}$ to each task, and use the energy model

E_1 in the experiments of this section.

In our experiments, we study the operation time of EBUA and EUA∗, within which the system remains functional. We still set $E_{bnd} = E_{rqd}\,(Load = 0.7)$, and vary $Load$ from 0.2 to 1.8 to study how EBUA handles the energy-bound. Figure 6.15 shows operation times of EBUA and EUA∗, normalized to the results of EBUA.

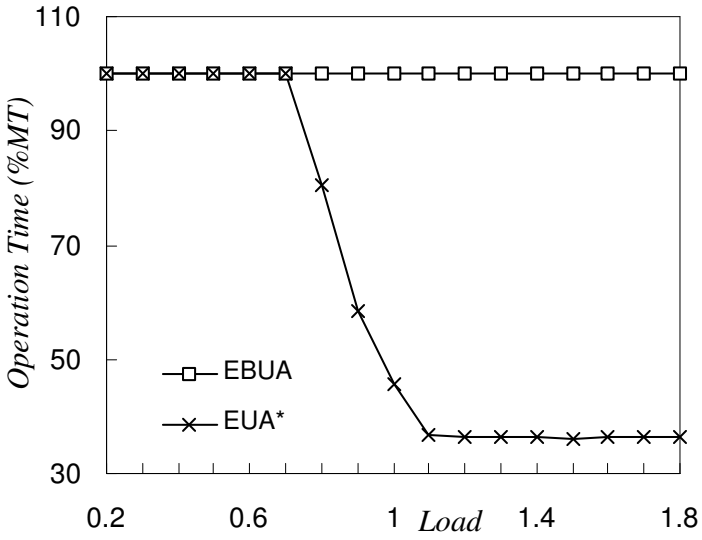


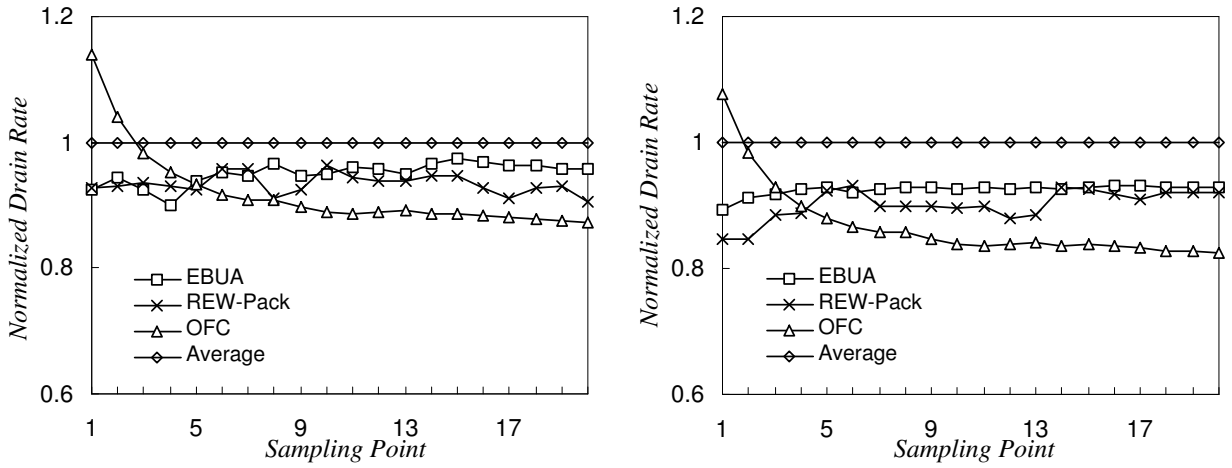
Figure 6.15: Operation Time vs. $Load$

We observe that, the operation time of EBUA is always the whole mission time MT, since EBUA dynamically monitors energy consumption and keeps system functional during $[0, MT]$. When $Load \leq 0.7$, EUA∗ can remain functional during $[0, MT]$, because the system is neither energy-bounded nor overloaded. But when $Load \geq 0.7$, the operation time of EUA∗ decreases to far below MT until $Load = 1.0$, as it cannot deal with energy-bounded systems. Note that when $Load \geq 1.0$, the operation time of EUA∗ becomes constant. This is because during system overloads, DVS always picks the highest frequency f_m, and the system-level energy consumption, hence E_{rqd}, also becomes constant. This also means that $Eratio = \frac{E_{bnd}}{E_{rqd}}$ is constant, when $Load \geq 1.0$.

6.3.3 Comparison of Dynamic Energy Drain Rates

For a battery, its discharge rate decides its life. Usually we hope this discharge rate can be as constant as possible, which is represented by small variances on the discharge curve. In this section, we approximately measure and compare the dynamic energy drain rates of different mechanisms. According to the relationship between energy, voltage, and current, a constant energy drain rate implies a constant battery discharge rate in terms of discharge current, assuming that the battery operates under a constant voltage.

For the same task sets as those in Section 6.3.1, we uniformly take 20 sampling time points during the interval $[0, MT]$, and measure the dynamic energy drain rate at each sampling point t_i. This rate is measured as $\frac{E^e(0,t_i)-E^e(0,t_{i-1})}{t_i-t_{i-1}}$, where $E^e(0,t_i)$ is the energy consumed from time 0 up to t_i, and $t_i - t_{i-1}$ is the sampling interval. In our experiments, we set $Load = 0.7$ and $E_{bnd} = 0.5 \times E_{rqd}\,(Load = 0.7)$. The measurements are normalized to the average energy drain rate, E_{bnd}/MT, and the normalized results under energy settings E_1 and E_2 are shown in Figure 6.16. Note that these figures only represent our qualitative study.



(a) $Load = 0.7$, E_1 (b) $Load = 0.7$, E_2

Figure 6.16: Normalized Energy Drain Rates with $E_{bnd} = \frac{1}{2}E_{rqd}\,(Load = 0.7)$ under E_1 and E_2

We observe from the plots in both Figure 6.16(a) and Figure 6.16(b) that, all strategies have dynamic energy drain rates less than the average rate. This is because the energy consumptions of all methods during the interval $[0, MT]$ are no larger than E_{bnd}. From the figures, EBUA has a constant energy drain rate. The initial drain rate of OFC is larger than the average rate, because before the mission starts, OFC has off-line selected tasks to execute, and at the beginning those tasks may generate a high energy drain rate. Also note that the drain rate of EBUA is higher than that of OFC and REW-Pack. This implies that the amount of energy consumed by EBUA is closer to the energy budget E_{bnd}. Thus, EBUA is more effective for energy-bounded systems, in the sense of exploiting as much as available energy for timeliness.

6.4 Experiments on Overhead of EBUA

Our next experiments focus on the impact of scheduler overhead on the performance of energy-efficient, real-time scheduling algorithms, including EBUA and other EDF-based ones (BaseEDF, LaEDF, and StaticEDF). Preliminary experiments in this section study the overhead of different algorithms without DVS—i.e., the algorithms work with a unique frequency. These results exploit important parameters such as execution times for our future work.

6.4.1 Modeling Scheduler Overhead

At each scheduling event, the total scheduler overhead O_t comprises the overhead of scheduling the ready job queue, i.e., queue-scheduling overhead O_q, and the overhead for possible frequency/voltage changes, i.e., switching overhead O_s. In number of cycles, O_q depends on the number of jobs n in the ready queue as well as the scheduling process, and O_s is decided by the specific processor.

O_s is zero when there is no frequency switching, and it varies with switching between different levels of speeds. Switching from a lower frequency to a higher one usually requires more overhead than switching in the other direction, since this requires both frequency and voltage changes thus more cycles, while switching in the other direction may require only frequency changes but no voltage changes.

The time and energy overhead for O_t is decided by the frequency selected by the operating system to run the scheduler. In this dissertation, we assume that O_s can be obtained through processor data sheets, and focus on queue-scheduling overhead O_q.

6.4.2 Effect of Scheduling Overhead without DVS

To study queue-scheduling overhead O_q, we implement the algorithms in our previously developed scheduling framework called *meta-scheduler* [57]. The meta-scheduler is an application-level framework for implementing UA schedulers on POSIX RTOSes, without RTOS modifications. We use QNX Neutrino 6.2.1 RTOS [52] in our study. All the algorithms are implemented in QNX without DVS—they are executed under a unique frequency. Thus, LaEDF, BaseEDF, and StaticEDF have the same overhead as EDF.

We only consider step TUFs in this experiment, and select task sets consisting of 10 to 20 periodic tasks, whose parameters are similar to those of task sets selected for task-level simulations, as described in Section 6.1.1. To investigate the impact of different scheduler overhead for applications with a broad magnitude of time constraints, we introduce a metric called "Termination Time Miss Load" (XML). The XML of a scheduler is the load after which the scheduler begins to miss task termination times.

In the case of independent periodic tasks, EDF is assured to meet all deadlines during under-loads. Thus, an ideal EDF scheduler (without any overhead) should have XML as 1.0. Similarly, since EBUA are equivalent to EDF during under-loads, it should have XML

also equal to 1.0. However, an actual implementation in the RTOS incurs certain overhead. Furthermore, the scheduler overhead tends to manifest itself for shorter execution time tasks. That is, the shorter task execution times, the lower XML is. Thus, the relationship between XML and the average task execution time provides a way of evaluating the actual overhead of a particular scheduler.

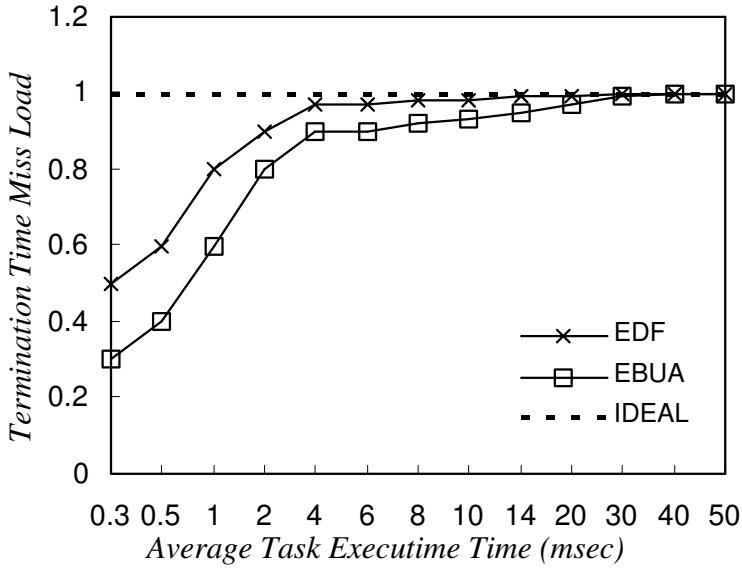


Figure 6.17: Termination Time Miss Load

We have measured the XMLs of LaEDF, BaseEDF, StaticEDF, and EBUA implemented in QNX on a 450 MHz Pentium II PC. With a unique frequency, LaEDF, BaseEDF, and StaticEDF have identical overhead as that of EDF. Thus, plots for EDF and EBUA are presented in Figure 6.17, where the ideal EDF scheduler (denoted as IDEAL) is also shown in the plot for comparison. The results show that our implementation of simple scheduling algorithms in QNX, such as EDF, is negligible for tasks with average execution time of as small as a few hundred microseconds. For complex scheduling algorithms, such as EBUA, the average task execution time shall be at least in the magnitude of a few milliseconds so that the impact of scheduler overhead is negligible.

Chapter 7

Past and Related Efforts

Most of the past efforts on energy-efficient real-time scheduling consider deadline-based timeliness optimality criteria such as meeting all or some deadlines, and apply DVS to save CPU energy. The only exception is the PA-BTA algorithm [29], which considers TUFs in real-time scheduling. But PA-BTA heuristically makes scheduling decisions without explicitly applying DVS. Besides PA-PTA, there are few efforts on energy-efficient UA scheduling prior to this Ph.D. dissertation work, although separately, UA scheduling and energy-efficient real-time scheduling have been well studied.

In this chapter, we broadly overview the algorithms in each category, and compare and contrast them with EBUA.

7.1 Deadline Scheduling and UA Scheduling

7.1.1 Deadline Scheduling

The problem of optimal scheduling for satisfying all task deadlines has been extensively studied in the literature. The Rate Monotonic scheduling algorithm [51] and the Earliest Deadline First (EDF) scheduling algorithm [18] have been established to be optimal for fixed and dynamic priority scheduling, respectively. Furthermore, deadline scheduling has been extended to variants of the hard timeliness optimality criterion of satisfying all deadlines. For instance, in [58], the author presents an overload management scheme that can satisfy deadlines of at least m instances of a periodic task within k consecutive releases, which is called the (m, k) *firm guarantee*.

The aforementioned scheduling algorithms only deal with sharing CPU cycles among real-time tasks that are subject to deadlines. When real-time tasks need both CPU cycles and other shared resources simultaneously (which is a fairly common scenario), interferences due to resource dependencies must be bounded by using real-time resource access protocols.

In the context of lock-based access, examples of real-time resource access protocols include the Priority Inheritance Protocol [42], the Priority Ceiling Protocol [42], and the Stack Resource Policy [48]. Furthermore, shared resources may be accessed in a lock-free manner. In such cases, instead of being blocked upon a failed access, a task continuously attempts to access the resource. The lock-free access scheme is a viable alternative to the lock-based access scheme, as shown in [59, 60].

7.1.2 UA scheduling

Overload scheduling algorithms deal with deadlines as well as non deadline timing constraints such as non-step TUFs, wherever proper. Many existing UA algorithms for overload scheduling consider step TUFs. In the event of no overloads, EDF is optimal with respect to meeting all deadlines. Thus, these algorithms aim to mimic the behavior of EDF during under-loaded situations as closely as possible. Furthermore, they seek to optimize other performance metrics during overloaded situations, since all deadlines cannot be satisfied during overloads.

One important performance metric considered by many algorithms during overload situations is the sum of utility (or "value") accrued by all tasks. In [61], the authors show that the upper bound on the competitive factor of any on-line scheduling algorithm is $1/(1+\sqrt{k})^2$, where k is the *importance ratio* of the task set. k is defined as the ratio of the maximum to the minimum task value density of the task set.

The $1/(1+\sqrt{k})^2$ upper bound on the competitive factor is achieved by the D^{over} algorithm presented in [25]. However, this does not lead to the best performance of D^{over}. This is because, D^{over} may overly reject tasks to cope with the worst-case task sequences so that it can achieve the highest competitive factor, no matter how strong the "adversary" is.

Besides the optimal D^{over} algorithm, heuristic algorithms have also been developed for effective UA scheduling during overloaded situations. The LBESA algorithm presented by Locke in [23] uses the notion of value density, which is complemented with feasibility tests. Locke's work is extended by several others including [62] and [63]. These variants are similar to LBESA in the sense that they reject tasks by ascending order of task value densities, or a variant metric of value density, to resolve the overload situation. Performance of the algorithms presented in [62] is better than LBESA's, but in general, it is very close.

Apart from the step time/utility model with one segment of execution per task, the *impre-*

cise computational model has also been proposed in [64] as an effective scheme to handle overloads. It assumes that the task execution consists of mandatory and optional parts. The scheduling objective is then to satisfy all deadlines of the mandatory parts, with the option of rejecting optional parts. Essentially, the imprecise computation model assumes two versions for each task: execution of the mandatory part is necessary to yield a result, but execution of both mandatory and optional parts can produce higher quality results. As an example, the RED (Robust Earliest Deadline first) algorithm presented in [65] combines many features including graceful performance degradation during overload, deadline tolerance, and resource reclaiming.

Our task model fundamentally differs from the RED model in that there is no deadline tolerance specifications. Rather, a task may yield any utility for completing anytime before the deadline time (termination time). EBUA fundamentally differs from all the aforementioned algorithms in that we consider non-increasing TUFs *and* mutual exclusion resource constraints. All the previously mentioned algorithms, except for the LBESA algorithm, only consider step TUFs. Furthermore, the LBESA algorithm does not consider mutual exclusion constraints.

In [66], the authors present the concept of "timeliness-functions." Unlike a step TUF, which drops to a zero utility at the task deadline (termination time), a timeliness-function linearly decreases to zero when the laxity of a task reaches zero. In [63], the same authors show that scheduling the task with the highest Dynamic Timeliness-Density (DTD) is more effective than scheduling the highest value density task. The DTD heuristic is echoed in a special case of EBUA, where tasks do not share resources and only one frequency is available; i.e., no DVS is performed. For that scenario, the EBUA algorithm selects a task only if the task UER is positive, which essentially requires positive laxity as in DTD.

Non-step TUFs have also been explored in the literature. For instance, the UPA [27] and CMA [26] algorithms consider non-increasing TUFs in the context of non-preemptive schedul-

ing of independent activities. Further, GUS [22], CUA [67], and RUA [28] consider arbitrary TUFs (including non-increasing functions), and allow preemption and mutual exclusion resource dependencies. These UA scheduling algorithms maximize the collective utility attained by all activities, but they provide no assurance on individual timeliness behavior such as a lower bound on individual activity utility that is probabilistically satisfied. Again, note that EBUA allows preemption, non-increasing TUFs, and mutual exclusion resource dependencies, while providing task-level performance assurance. To our best knowledge, besides EBUA and the author's work leading to EBUA, the only energy-efficient UA algorithm is PA-BTA [29], which considers non-increasing TUFs. However, PA-BTA is restricted to independent activities, and provides no performance assurance.

In the context of overload scheduling, some past efforts consider shared resources that have mutual exclusion constraints. The DASA algorithm [24] considers shared resources with mutual exclusion constraints, but only for step TUFs, while GUS [22], CUA [67], and RUA [28] consider mutual exclusion resource dependencies and arbitrary TUFs. EBUA maximizes the system-level energy-efficiency as well as providing performance assurances. Thus, unlike DASA, GUS and RUA, which considers *potential value/utility density* as the metric in scheduling, EBUA considers the UER as a metric in constructing the schedule to achieve maximum system-level energy efficiency.

7.2 DVS in Real-Time Systems

DVS is a common method used to save CPU energy. DVS mechanisms used for general-purpose applications heuristically predict the workload based on the average CPU utilization, and adjust the voltage and frequency [8, 4, 68]. But they are unsuitable for dynamic real-time applications due to the timing constraint and demand variations of such applications. As an example, Grunwald concluded in [8] that no heuristic policy that they examined achieved

the goal of saving energy without affecting the timeliness performance.

DVS schemes for aperiodic tasks were proposed in [69] and [70]. Yao et al. provided a static off-line DVS scheduling algorithms [69] for aperiodic tasks, assuming aperiodic tasks and worst-case execution times (WCET). Hong et al. proposed heuristics for on-line scheduling of aperiodic tasks while not affecting the feasibility of periodic requests [70].

Stochastic DVS is used to handle runtime demand variations [71, 6, 9, 11]. It starts a job slowly and accelerates as the job progresses. [9] and [71] aim at general-purpose applications. In [9], the DVS technique called PACE is performed with applying soft deadlines and estimating task work distributions, while in [71], DVS is based on prediction of episodic interaction. These methods show good results for maintaining short response times in human-interactive and multimedia applications, but are not intended for the stricter timeliness constraints of real-time systems. [6] and [11] propose real-time DVS techniques. In [6], off-line analysis is combined with online slack-time stealing [72] and cycle-based, stochastic DVS for hard real-time systems. The GRACE-OS proposed in [11] obtains the demand distribution via online profiling and estimation, and seeks to find a speed for each cycle based on the demand distribution of applications. Unlike most of the other strategies that present simulations only, GRACE-OS implements the stochastic DVS on a PC platform. The cycle-based speed change performed by [6] and [11] is within a job execution, and this is not suitable for dynamic, embedded real-time systems with limited computing resources.

Most other real-time DVS strategies adopt *per-task* or *per-job* DVS techniques, which change speed only when starting a task/job. Among them, [10] and [73] only take into account a single execution and deadline for a task. As such, they can only handle tasks that execute just once. Furthermore, it is doubtable whether new tasks entering the system can be handled in a timely manner. This is because with single-shot tasks, adapting closely to the current task set for DVS may use up the system's computing resources. The DVS scheduler in [73] is implemented using a modified StrongARM embedded system board, as described in [74].

Per-job DVS techniques can handle the most important, canonical model of real-time systems, which use periodic real-time tasks. Such DVS techniques, often integrated with CPU scheduling, usually derive workload from worst-case CPU demands of real-time applications [75, 3, 7, 12, 13]. In [75], the worst-case CPU demand is calculated off-line. The online scheduler further reduces frequency and voltage when tasks use less than their requested computing quota, but can still provide deadline guarantees. This mechanism is complicated and cannot deal effectively with dynamic task sets. Since the tasks' demands are not always the worst-case, some reclamation techniques have been proposed to reclaim the unused cycles to save more energy for both dynamic-priority real-time systems [3, 7, 12, 13] and fixed-priority systems [7, 13]. These reclamation techniques first run the CPU fast (assuming the worst-case demand) and then decelerate when a job finishes early. In addition, more aggressive techniques are also proposed in [3, 7] with the assumption that *the current and future instances of a task will most probably present a computational demand which is lower than the worst-case*. The aggressive techniques start a job with the lowest frequency required to meet the minimum work that must be done now, and then accelerate as it progresses. Furthermore, the real-time DVS schemes proposed in [7] have also been implemented and demonstrated in a real working system.

Per-job stochastic DVS techniques are proposed in [76, 77]. Such techniques change speed for each job of a task based on a stochastic model (e.g., Markov process) of the task's CPU demands.

Thus, EBUA supports a per-job stochastic DVS strategy for both UAM and periodic (as a special case of UAM) real-time tasks. EBUA extends the Look-Ahead RT-DVS in [7], and changes speed for each job of a task based on stochastic properties of the tasks' CPU cycle demands, such as the expected values and variances. Such CPU cycle demand analysis is performed through processor demand approach. In addition, EBUA considers reduction of the system-level energy consumption when applying DVS. Further, EBUA integrates overload

handling into the DVS mechanism to achieve higher system-level energy efficiency. Therefore, EBUA is the only energy-efficient real-time scheduling algorithm that considers the system's energy consumption, and provides assurance on tasks' timeliness performance with UA scheduling and per-job stochastic DVS strategy.

7.3 Energy-Efficient Real-Time Scheduling

Prior to EBUA and its related research, most of the past efforts on energy-efficient real-time scheduling focus on the deadline time constraint and deadline-based timeliness optimality criteria (e.g., meeting all or some percentage of deadlines [16, 7, 3, 11]), resource-independent activities—i.e., activities that do not share non-CPU resources which are subject to mutual exclusion constraints, and minimizing only the CPU's energy consumption (as mentioned before). Exceptions include [14, 29, 39].

The work in [14] considers the optimality criterion of maximizing collective value, where value is equivalent to our utility notion. However, [14] is restricted to step value functions or step TUFs (see Figure 1.1(a)). The work in [29] considers non-step TUFs, but it is restricted to resource-independent activities. The work in [39] considers voltage scheduling for periodic real-time tasks with non-preemptive blocking sections. However, [39] is restricted to deadlines and deadline-based timeliness optimality. The work in [29] considers system-level energy consumption, but it is restricted to resource-independent activities, and provides no assurances on timeliness behaviors.

The aforementioned restrictions on deadline-based optimality criteria and independent task sets were overcome in energy-efficient UA real-time scheduling works [32, 33, 38], which were proposed in the development of EBUA, and led to EBUA.

Most past efforts on energy-efficient, real-time scheduling consider activity arrival models

that are either periodic, or frame-based (where all periods are equal), or sporadic. These include past works that consider deadline-based timeliness optimality criteria (e.g., meeting all or some percentage of deadlines) [11, 7, 3, 12, 13], and those that consider UA criteria (e.g., maximizing summed utility) [14, 29, 32, 33]. As far as we know, the only exception is [29], which allows aperiodic arrivals. However, [29] provides no timeliness assurances. Thus, prior efforts are concentrated on two extremes: (1) those that provide timeliness assurances, but under highly restrictive periodic, frame-based, or sporadic arrivals; or (2) those that allow aperiodic arrivals, but provide no timeliness assurances. Both extremes are inappropriate for the applications/domains of interest to us.

EBUA bridges these extremes by considering the UAM model [34]. UAM embodies a "stronger" adversary than most traditional arrival models (e.g., periodic, frame-based, sporadic), and subsumes those models as special cases. The first energy-efficient UA scheduling work to considers UAM, which is a subset of EBUA, appeared in [38]. Later, the author extended [38] to develop EBUA.

None of the works mentioned above considers scheduling under fixed energy budgets. Many dynamic real-time embedded systems are subject to a finite energy budget for the entire duration of their operational/mission life. Consequently, they must operate without violating their energy budgets, which is a *hard* constraint. Energy-bounded real-time scheduling algorithms are proposed for real-time systems with fixed energy budgets.

Most of the past efforts are restricted to deadlines, step TUFs, and deadline-based timeliness optimality. For example, [35] considers maximizing the total reward in a system under a fixed energy budget, with periodic and frame-based tasks (where reward is equivalent to our utility notion). Static scheduling of periodic tasks with continuous resource/utility functions and a fixed budget is explored in [36]. In [37], static and dynamic solutions for energy-constrained periodic task systems with or without DVS capacity are investigated. All these works are restricted to either deadlines or step TUFs (see Figure 1.1(a)).

It is interesting to note that energy-efficient real-time scheduling efforts which do not consider energy budgets are restricted to deadline-based optimality criteria [16, 7, 3]. This restriction was overcome in energy-efficient UA real-time scheduling works [29, 32, 33, 38]. However, none of these works consider scheduling under fixed energy budgets.

Therefore, from the energy budget standpoint, past works are concentrated on two extremes: (1) those that satisfy energy budgets, but for deadlines and deadline-based optimality; or (2) those that allow non step TUFs and UA optimality, but do not satisfy energy budgets.

Again, EBUA bridges these extremes by satisfying energy budgets under UA criteria. If the system's energy budget is exceeded by the tasks' demands, EBUA seeks to defer or reject some jobs in a controlled fashion, so as to enable maximal utility to be accrued by the mission, without adversely affecting the system's functionality during the mission. Further, EBUA tries to minimize the system-level energy consumption and maximize the system's energy efficiency as much as possible at all times, and provides assurances on activity timeliness behavior whenever possible. We are not aware of any other efforts that solve the *energy-bounded, TUF/UA scheduling problem under the UAM and system-level energy model* that is solved by EBUA.

Chapter 8

Conclusions and Contributions

This dissertation presents the design and evaluation of EBUA, a resource-constrained, energy-bounded, utility-accrual real-time scheduling algorithm for dynamic mobile embedded systems, which must remain functional during an operation/mission with a bounded energy budget. EBUA considers application activities that are subject to TUF time constraints, resource dependencies, UAM arrival model, statistical timeliness requirements, and bounds on system-level energy consumption.

The key underpinning of EBUA is the observation that embedded real-time applications usually exhibit large variations in their *actual* cycle demands. This provides opportunities for providing statistical, timeliness performance assurances, while respecting resource dependencies, and for improving system-level energy efficiency. To realize this, the algorithm statistically allocates cycles to individual application tasks and executes their allocated cycles at different speeds with DVS. EBUA makes such stochastic decisions based on the statistical properties of the task demands. If the system's energy budget is exceeded by the tasks' demands, EBUA dynamically skips less important jobs for execution to achieve timeliness objectives, while assuring that the system remains functional until the end of its mission.

During CPU overload situations, the algorithm heuristically schedules tasks to maximize collective utility so as to improve system-level energy efficiency.

We establish several timeliness and non-timeliness properties of the algorithm such as timeliness optimality during under-loads, deadlock-freedom, correctness, and mutual exclusion.

We conduct experimental studies by simulating the algorithm on the DVS-enabled AMD k6 processor model, and by implementing it on QNX Neutrino 6.2.1 RTOS. Our experimental results illustrate that EBUA never violates the energy budget, provides statistical performance assurances when possible, and improves system-level energy efficiency. Further, they confirm EBUA's superiority over other energy-efficient real-time scheduling algorithms on timeliness and energy consumption behaviors.

8.1 Contributions

The contribution of the dissertation is the EBUA algorithm. EBUA solves the scheduling problem that intersects the following individual scheduling problems: (1) CPU scheduling under TUF time constraints to optimize UA optimality criteria including providing assurances on individual activity timeliness behavior, while satisfying mutual exclusion resource constraints; (2) DVS-based CPU scheduling for bounded and reduced system-level energy consumption; and (3) CPU scheduling under the UAM for activity arrival behaviors.

This overlapping scheduling problem solved by EBUA has not been studied in the past (though each of the individual scheduling problems has been studied). Thus, EBUA is the only energy-efficient real-time scheduling algorithm that considers limited energy bounds, and integrates run-time-based per-job DVS with UA scheduling. It provides assurances on tasks' timeliness performance and maximizes the system-level energy efficiency, by considering utility maximization under energy bounds. The algorithm dynamically scales voltage and

frequency to reduce system-level energy consumption, and to obtain additional energy savings for selecting new jobs in a dynamic fashion. EBUA uses UER as the single system-level scheduling metric, which integrates timeliness and energy consumption.

8.2 Future Directions

The EBUA algorithm developed in this thesis can be used to significantly improve the power dissipation and energy efficiency of a mobile, embedded system. Some aspects of the work are directions for further research. Such research can be conducted through algorithm design, simulation, analytical study and kernel implementation.

1. Consider More General Task Arrival Models

One direction would be to consider more general task arrival models than UAM. UAM only states the number of maximum job arrivals in a time window. More dynamic arrival models can be described through the probabilistic distribution of job arrivals such as the probabilistic UAM (or PUAM) model presented in [22]. In PUAM, a task's inter-arrival time is stochastically described. The probability density function or mean and variance of the inter-arrival time are given as inputs to the scheduler.

With a PUAM model, optimization goals can include satisfying tasks' performance requirements and maximizing system-level energy efficiency. Furthermore, stochastic schedulability analysis can be conducted, which allows us to establish stochastic properties (e.g., probability distribution functions) for utilities of the system and for each activity.

2. Consider Reward-based Scheduling with UA Scheduling

Another interesting direction is to combine TUF/UA scheduling with reward-based scheduling. Reward-based scheduling refers to the problem in which there is a reward associated

with the execution of a task [78]. It encompasses scheduling frameworks including *Imprecise Computations* [64, 79] and *IRIS (Increasing Reward with Increasing Service)* [80] models.

Reward-based scheduling proposes optional parts in addition to the mandatory parts of task execution times; a non-decreasing reward function is associated with the execution of each optional part. The concept of reward-based scheduling is widely applicable in areas such as multimedia applications, image and speech processing, time-dependent planning, real-time heuristic search, and database query processing, where a partial or approximate but timely result is usually acceptable.

Reward-based scheduling is different from UA scheduling, because in reward-based scheduling, the longer the optional part executes, the higher the reward; while in UA scheduling the utility (reward) can only be accrued by an activity when it is completed, and the utility value is decided by the completion time.

If reward functions are defined for our application, the number of executed cycles of each task affects the reward that can be attained. In addition, the number of cycles allocated to each task and the speed to execute them affect energy-saving and timeliness performance in terms of utility accrual. This results in very challenging problems to solve. How reward-based scheduling techniques can be incorporated into TUF/UA scheduling is an open problem, and this may be a future extension to our algorithm.

3. Consider Non-Blocking Synchronization

Our dissertation considers activities that desire atomic behaviors under concurrent, mutually exclusive sharing of logical resources (e.g., data objects), and solves the resource sharing through lock-based synchronization. Lock-based synchronization has several drawbacks including blocking times for resource accesses, reduced concurrency, increased system overhead, and potential for deadlocks (due to nested resource requests and process crashes). Non-blocking synchronization—e.g., wait-free, lock-free—have been studied in [59, 60]. Non-

blocking synchronization overcomes drawbacks of lock-based, at the expense of increased space and time costs.

A future avenue of research is to consider non-blocking synchronization mechanisms for energy-efficient UA scheduling. With non-blocking synchronization, the absence of blocking times, increased concurrency, and reduced system overhead may lead to better optimization of UA criteria such as higher attained activity utility. In addition, they may result in more slack time that can be used by DVS calculation to further reduce the system energy consumption, without compromising the timeliness performance. Also, further investigation is necessary in order to identify the tradeoffs of non-blocking synchronization versus lock-based for EBUA.

4. Consider Variable Cost Functions

Considering variable cost scheduling with UA scheduling is another future direction. By variable cost scheduling we mean scheduling activities that have durations (e.g., tasks with execution times) which vary while being performed—e.g., depending on when they begin, or on how long they have been running, or on other factors. In this context, cost means the duration of the activity—a term that comes from one of the interesting and important applications for such scheduling. The variable cost is specified by a variable cost function (or VCF). Thus, even if there were no new activity arrivals, the load to be scheduled changes while the activities are being performed.

Past efforts on deadline-based scheduling do not deal with variable cost scheduling. [81] considers the intersection of UA scheduling and variable cost scheduling. Handling VCFs with TUFs and energy constraints requires significant additional research. When the execution cycles of activities are described with VCFs, they may vary as the activities are being performed. With the additional constraint on execution cycles, further investigation into algorithms is necessary in order to optimize our scheduling criteria.

5. Kernel Implementation

To perform the energy and utility evaluation of EBUA, we have used a normalized approach and conducted simulation experiments. On the other hand, algorithm overhead studies with implementation in Section 6.4 and [82] only consider queue-scheduling overhead O_q. To account for the impact of DVS overhead O_s on the system's energy saving and performance, it would be interesting to evaluate the algorithms with full DVS supports.

Energy-efficient real-time algorithms with DVS are implemented on the IBM PowerPC 405LP embedded board [83] and on the PC architecture [7, 11]. The work in [7, 11] considers a Hewlett-Packard laptop with a single mobile AMD processor. We can adopt methods similar to those in [7, 11] for the prototype implementation of EBUA. The hardware platform can be a Hewlett-Packard Pavilion laptop with an AMD Athlon^{TM} processor. This processor features the `PowerNow!` technology and supports multiple frequencies. Further, its frequency and voltage can be adjusted dynamically under software control.

The prototype EBUA and its DVS functions can be implemented as a set of loadable kernel modules (LKMs) that can be dynamically loaded into the Linux kernel 2.4. RedHat 7.3 (Valhalla) can be considered as the software platform. Although it is not a real-time operating system, Linux is easily extended through modules, and provides a robust development environment familiar to us.

Interesting future experiments should include measuring the actual power consumption of EBUA. Similar to methods in [7], the laptop battery can be removed to have the system running using the external DC power adapter. In this situation, current probes and a digital oscilloscope shall be used to measure the power consumption of the laptop as the product of current and voltage supplied. EBUA's performance can be analyzed by comparing the measured energy savings with simulation results in this dissertation.

Bibliography

[1] P. J. M. Havinga and G. J. M. Smith, "Design Techniques for Low-power Systems," in *Journal of Systems Architecture*, 2000, vol. 46:1.

[2] M. Pedram, "Power Minimization in IC Design: Principles and Applications," in *ACM Transactions on Design Automation of Electronics Systems*, January 1996, vol. 1:1, pp. 3–56.

[3] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 2001, pp. 95 –105.

[4] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for Reduced CPU Energy," in *Proceedings of The USENIX Symposium on Operating Systems Design and Implementation*, November 1994, pp. 13–23.

[5] K. Flautner and T. Mudge, "Vertigo: Automatic Performance-Setting for Linux," in *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[6] F. Gruian, "Hard Real-time Scheduling for Low Energy Using Stochastic Data and DVS Processors," in *Proceedings of International Symposium on Lower-Power Electronics and Design*, August 2001.

[7] P. Pillai and K. G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," in *Proceedings of the ACM symposium on Operating Systems Principles*, 2001, pp. 89–102.

[8] D. Grunwald, P. Levis, K. Farkas, C. M. III, and M. Neufeld, "Policies for Dynamic Clock Scheduling," in *Proceedings of 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[9] J. Lorch and A. Smith, "Improving Dynamic Voltage Scaling Algorithms with PACE," in *Proceedings of ACM SIGMETRICS 2001 Conference (Cambridge, MA)*, June 2001, pp. 50–61.

[10] T. Pering, T. Burd, and R. Brodersen, "Voltage Scheduling in the lpARM Microprocessor System," in *Proceedings of International Symposium on Lower-Power Electronics and Design*, July 2000.

[11] W. Yuan and K. Nahrstedt, "Energy-Efficient Soft Real-Time CPU Scheduling for Mobile Multimedia Systems," in *Proceedings of the ACM Symposium on Operating Systems Principles*. 2003, pp. 149–163, ACM Press.

[12] W. Kim, J. Kim, and S. L. Min, "A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis," in *Proceedings of IEEE/ACM Design, Automation and Test in Europe (DATE)*, March 2002.

[13] W. Kim, J. Kim, and S. L. Min, "Dynamic Voltage Scaling Algorithm for Fixed-Priority Real-Time Systems Using Work-Demand Analysis," in *Proceedings of International Symposium on Lower-Power Electronics and Design*, August 2003.

[14] C. Rusu, R. Melhem, and D. Mosse, "Multi-version Scheduling in Rechargeable Energy-Aware Real-Time Systems," in *Proceedings of IEEE Euromicro Conference on Real-Time Systems*, July 2003.

[15] A. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS Digital Design," in *IEEE Journal of Solid-State Circuits*, April 1992, vol. 27, pp. 473–484.

[16] R. Graybill and R. Melhem, *Power Aware Computing*, Kluwer Academic/Plenum Publishers, 2002.

[17] R. K. Clark, E. D. Jensen, and N. F. Rouquette, "Software Organization to Facilitate Dynamic Processor Scheduling," in *Proceedings of IEEE Parallel and Distributed Processing Symposium*, April 2004.

[18] W. Horn, "Some Simple Scheduling Algorithms," *Naval Research Logistics Quaterly*, vol. 21, pp. 177–185, 1974.

[19] E. D. Jensen, C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1985, pp. 112–122.

[20] R. Clark, E. D. Jensen, A. Kanevsky, J. Maurer, P. Wallace, T. Wheeler, Y. Zhang, D. Wells, T. Lawrence, and P. Hurley, "An Adaptive, Distributed Airborne Tracking System," in *Proceedings of The IEEE Workshop on Parallel and Distributed Systems*. April 1999, vol. 1586 of *LNCS*, pp. 353–362, Springer-Verlag.

[21] D. P. Maynard, S. E. Shipman, R. K. Clark, J. D. Northcutt, R. B. Kegley, B. A. Zimmerman, and P. J. Keleher, "An Example Real-Time Command, Control, and Battle Management Application for Alpha," Tech. Rep., Department of Computer Science, Carnegie Mellon University, December 1988, Archons Project Technical Report 88121.

[22] P. Li, *Utility Accrual Real-Time Scheduling: Models and Algorithms*, Ph.D. thesis, Virginia Tech, 2004, `http://scholar.lib.vt.edu/theses/available/etd-08092004-230138/` (last accessed: June 22, 2005).

[23] C. D. Locke, *Best-Effort Decision Making for Real-Time Scheduling*, Ph.D. thesis, Carnegie Mellon University, 1986, CMU-CS-86-134, `http://www.real-time.org` (last accessed: June 22, 2005).

[24] R. K. Clark, *Scheduling Dependent Real-Time Activities*, Ph.D. thesis, Carnegie Mellon University, 1990, CMU-CS-90-155, `http://www.real-time.org` (last accessed: June 22, 2005).

[25] G. Koren and D. Shasha, "D-Over: An Optimal On-line Scheduling Algorithm for Overloaded Real-Time Systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1992, pp. 290–299.

[26] K. Chen and P. Muhlethaler, "A Scheduling Algorithm for Tasks Described by Time Value Function," *Journal of Real-Time Systems*, vol. 10, no. 3, pp. 293–312, May 1996.

[27] J. Wang and B. Ravindran, "Time-Utility Function-Driven Switched Ethernet: Packet Scheduling Algorithm, Implementation, and Feasibility Analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 2, pp. 119–133, February 2004.

[28] H. Wu, B. Ravindran, E. D. Jensen, and U. Balli, "Utility Accrual Scheduling under Arbitrary Time/utility Functions and Multi-unit Resource Constraints," in *Proceedings of 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, August 2004, pp. 80–98.

[29] J. Wang, B. Ravindran, and T. Martin, "A Power Aware Best-Effort Real-Time Task Scheduling Algorithm," in *Proceedings of The IEEE Workshop on Software Technologies for Future Embedded Systems, IEEE International Symposium on Object-oriented Real-time Distributed Computing*, May 2003, pp. 21–28.

[30] T. Martin, *Balancing Batteries, Power and Performance: System Issues in CPU Speed-Setting for Mobile Computing*, Ph.D. thesis, Carnegie Mellon University, August 1999.

[31] T. Martin and D. Siewiorek, "Non-ideal Battery and Main Memory Effects on CPU Speed-Setting for Low Power," *IEEE Transactions on VLSI Systems*, vol. 9, no. 1, pp. 29–34, February 2001.

[32] H. Wu, B. Ravindran, E. D. Jensen, and P. Li, "CPU Scheduling for Statistically-Assured Real-Time Performance and Improved Energy Efficiency," in *Proceedings of 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, September 2004, pp. 110–115.

[33] H. Wu, B. Ravindran, E. D. Jensen, and P. Li, "Energy-Efficient, Utility Accrual Scheduling under Resource Constraints for Mobile Embedded Systems," in *Proceedings of Fourth ACM International Conference on Embedded Software (EMSOFT)*, September 2004, pp. 64–73.

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

[35] C. Rusu, R. Melhem, and D. Mosse, "Maximizing the System Value while Satisfying Time and Energy Constraints," *IBM Journal of Research and Development*, vol. 47, no. 5/6, pp. 689–702, September/November 2003.

[36] D.-I. Kang, S. P. Cargo, and J. Suh, "A Fast Resource Synthesis Technique for Energy-Efficient Real-Time Systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 2002.

[37] T. A. AlEnawy and H. Aydin, "On Energy-Constrained Real-Time Scheduling," in *Proceedings of IEEE Euromicro Conference on Real-Time Systems*, June 2004, pp. 165–174.

[38] H. Wu, B. Ravindran, and E. D. Jensen, "Energy-Efficient, Utility Accrual Real-Time Scheduling Under the Unimodal Arbitrary Arrival Model," in *Proceedings of IEEE/ACM Design, Automation and Test in Europe (DATE)*, March 2005, pp. 474–479.

[39] F. Zhang and S. T. Chanson, "Blocking-Aware Processor Voltage Scheduling for Real-Time Tasks," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 2, pp. 307–335, May 2004.

[40] E. D. Jensen, "A Timeliness Paradigm for Mesosynchronous Real-Time Systems," *Invited Talk, The IEEE Real-Time Technology and Applications Symposium*, 2003, `http://www.real-time.org`.

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

[42] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[43] E. D. Jensen, "Asynchronous Decentralized Real-Time Computer Systems," in *Real-Time Computing*, NATO Advanced Study Institute. Springer Verlag, October 1992.

[44] J. M. Anderson, W. E. Weihl, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, and C. A. Waldspurger, "Continuous Profiling: Where Have All the Cycles Gone?," in *Proceedings of 16th Symposium on Operating Systems Principles*, October 1997, pp. 1–14.

[45] 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*, December 2002.

[46] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System Support for Automated Profiling and Optimization," in *Proceedings of 16th Symposium on Operating Systems Principles*, October 1997, pp. 15–26.

[47] D. Zhu, N. AbouGhazaleh, D. Mosse, and R. Melhem, "Power Aware Scheduling for AND/OR Graphs in Multi-Processor Real-Time Systems," in *Proceedings of The IEEE International Conference on Parallel Processing*, August 2002.

[48] T. P. Baker, "Stack-based Scheduling of Real-Time Processes," *Journal of Real-Time Systems*, vol. 3, no. 1, pp. 67–99, March 1991.

[49] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor," *Real-Time Systems*, vol. 2, no. 4, pp. 301–324, November 1990.

[50] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, 2nd edition, 2005.

[51] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[52] QNX, "QNX Neutrino RTOS," `http://www.qnx.com/` (last accessed: June 22, 2005).

[53] A. Varga, "OMNeT++ Discrete Event Simulation System," `http://www.omnetpp.org/` (last accessed: June 22, 2005).

[54] Advanced Micro Devices Corporation, "Mobile AMD-K6-2+ Processor Data Sheet," Publication #23446, June 2000.

[55] K. M. Zuberi, P. Pillai, and K. G. Shin, "EMERALDS: A Small-Memory Real-Time Microkernel," in *Proceedings of the ACM symposium on Operating Systems Principles*, 1999, pp. 277–291.

[56] M. Dertouzos, "Control Robotics: the Procedural Control of Physical Processes," *Information Processing*, vol. 74, 1974.

[57] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A formally verified application-level framework for real-time scheduling on posix real-time operating systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 613–629, September 2004.

[58] P. Ramanathan, "Overload Management in Real-Time Control Applications Using the (m, k) Firm Guarantee," *IEEE Transactions on Computers*, vol. 10, no. 6, pp. 549–559, 1999.

[59] J. H. Anderson, S. Ramamurthy, and K. Jeffay, "Real-Time Computing with Lock-Free Shared Objects," *ACM Transactions on Computer Systems*, vol. 15, no. 2, pp. 134–165, May 1997.

[60] H. Cho, B. Ravindran, and E. D. Jensen, "A Space-Optimal, Wait-Free Real-Time Synchronization Protocol," in *Proceedings of IEEE Euromicro Conference on Real-Time Systems*, July 2005, pp. 79–88.

[61] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha, "On-line Scheduling in the Presence of Overload," in *Proceedings of IEEE Annual Symposium on Foundations of Computer Science*, October 1991, pp. 101–110.

[62] D. Mosse, M. E. Pollack, and Y. Ronen, "Value-Density Algorithm to Handle Transient Overloads in Scheduling," in *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, June 1999, pp. 278–286.

[63] S. A. Aldarmi and A. Burns, "Dynamic Value-Density For Scheduling Real-Time Systems," in *Proc. of Euromicro Conference on Real-Time Systems*, June 1999, pp. 270–277.

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

[65] G. Buttazzo and J. Stankovic, "RED: Robust Earliest Deadline Scheduling," in *Proceedings of International Workshop on Responsive Computing Systems*, September 1993, pp. 100–111.

[66] S. A. Aldarmi and A. Burns, "Time-Cognizant Value Functions for Dynamic Real-Time Scheduling," Tech. Rep., Department of Computer Science, The University of York, U.K., 1998, YCS-306.

[67] H. Wu, B. Ravindran, and E. D. Jensen, "Utility Accrual Scheduling Under Joint Utility and Resource Constraints," in *Proceedings of The IEEE International Symposium on Object-oriented Real-time distributed Computing*, May 2004, pp. 307–314.

[68] T. Pering, T. Burd, and R. Brodersen, "The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms," in *Proceedings of International Symposium on Lower-Power Electronics and Design*, June 1998, pp. 76–81.

[69] F. Yao, A. Demers, and S. Shankar, "A Scheduling Model for Reduced CPU Energy," in *Proceedings of IEEE Symposium on Foundations of Computer Science*, 1995, pp. 374–382.

[70] I. Hong, M. Potkonjak, and M. B. Srivastava, "On-Line Scheduling of Hard Real-Time Tasks on Variable Voltage Processors," in *Proceedings of International Conference on Computer-Aided Design (ICCAD'98)*, 1998, pp. 653–656.

[71] K. Flautner, S. Reinhardt, and T. Mudge, "Automatic Performance-setting for Dynamic Voltage Scaling," in *Proceedings of The 7th Conference on Mobile Computing and Networking MOBICOM'01 (Rome, Italy)*, July 2001.

[72] J. P. Lehoczky and S. Ramos-Thuel, "Algorithms for Scheduling Hard-Aperiodic Tasks in Fixed-Priority Systems using Stack Stealing," in *Proceedings of The IEEE Real-Time Systems Symposium*, 1994.

[73] J. Pouwelse, K. Langendoen, and H. Sips, "Energy Priority Scheduling for Variable Voltage Processors," in *Proceedings of International Symposium on Lower-Power Electronics and Design ISLPED'01*, August 2001.

[74] J. Pouwelse, K. Langendoen, and H. Sips, "Dynamic Voltage Scaling on a Low-Power Microprocessor," in *Proceedings of The 7th Conference on Mobile Computing and Networking MOBICOM'01 (Rome, Italy)*, July 2001.

[75] C. M. Krishna and Y.-H. Lee, "Voltage-Clock-Scaling Techniques for Low Power in Hard Real-Time Systems," in *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, May 2000, pp. 156–165.

[76] T. Simunic, L. Benini, A. Acquaviva, P. W. Glynn, and G. D. Micheli, "Dynamic Voltage Scaling and Power Management for Portable Systems," in *Proceedings of Design Automation Conference*, June 2001, pp. 524–529.

[77] A. Sinha and A. Chandrakasan, "Dynamic Voltage Scaling Using Adaptive Filtering of Workload Traces," in *Proceedings of 4th International Conference on VLSI Design*, January 2001.

[78] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Optimal Reward-Based Scheduling for Periodic Real-Time Tasks," *IEEE Transactions on Computers*, vol. 50, no. 2, pp. 111–130, Feburary 2001.

[79] J. W. S. Liu, K. J. Lin, W. K. Shih, A. Yu, C. Chung, J. Yao, and W.Zhao, "Algorithms for Scheduling Imprecise Computations," *IEEE Computer*, vol. 24, no. 5, pp. 129–139, May 1991.

[80] J. K. Dey, J. F. Kurose, and D. Towsley, "On-Line Scheduling Policies for a Class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks," *IEEE Transactions on Computers*, vol. 45, no. 7, pp. 802–813, July 1996.

[81] H. Wu, U. Balli, B. Ravindran, and E. D. Jensen, "Utility Accrual Real-Time Scheduling Under Variable Cost Functions," in *Proceedings of 11th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2005.

[82] H. Wu, B. Ravindran, and E. D. Jensen, "The Impact of Scheduler Overhead on the Performance of Mobile, Embedded Real-Time Systems," in *Proceedings of IEEE Euromicro Conference on Real-Time Systems, Work-in-Progress Session*, July 2005.

[83] Y. Zhu and F. Mueller, "Feedback EDF Scheduling Exploiting Hardware-Assisted Asynchronous Dynamic Voltage Scaling," in *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2005.

Vita

Haisang Wu is currently a Ph.D. candidate in the Bradley Department of Electrical and Computer Engineering at Virginia Polytechnic Institute and State University (Virginia Tech), USA. He received his B.E. (cum laude) and M.S.E. (summa cum laude) both in Electronic Engineering, from Tsinghua University, Beijing, China in 1999 and 2002, respectively.

His Ph.D. dissertation research focuses on energy-efficient and power-aware scheduling for real-time embedded systems. His other research interests include real-time operating systems and distributed systems, real-time and network security, IPv6, and network traffic engineering and routing.

Mr. Wu had been the recipient of "Excellent Student Prize" during all years of his study at Tsinghua University, from 1995 to 2002, for his distinguished academic performance. He is also a recipient of the Virginia Tech Graduate Student Travel Fund Program (TFP) Award of 2005.

Mr. Wu is a student member of the IEEE.