Energy-Efficient, Utility Accrual Scheduling under Resource Constraints for Mobile Embedded Systems*

Haisang Wu⋆, Binoy Ravindran⋆, E. Douglas Jensen†, and Peng Li‡

⋆ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
{hswu02,binoy}@vt.edu

†The MITRE Corporation
Bedford, MA 01730, USA
jensen@mitre.org

‡Microsoft Corporation
Redmond, WA 98052, USA
pengli@microsoft.com

Abstract

We present an energy-efficient, utility accrual, real-time scheduling algorithm called ReUA. ReUA considers an application model where activities are subject to time/utility function time constraints, mutual exclusion constraints on shared non-CPU resources, and statistical performance requirements on individual activity timeliness behavior. The algorithm targets mobile embedded systems where *system-level* energy consumption is also a major concern. For such a model, we consider the scheduling objectives of (1) satisfying the statistical performance requirements and (2) maximizing the system-level energy efficiency, while respecting resource constraints. Since the problem is $\mathcal{NP}$-hard, ReUA allocates CPU cycles using statistical properties of application cycle demands, and heuristically computes schedules with a polynomial-time cost. We analytically establish several timeliness and non-timeliness properties of the algorithm. Further, our simulation experiments illustrate ReUA's effectiveness and superiority.

1 Introduction

Energy consumption has become one of the primary concerns in electronic system design due to the recent popularity of portable devices and the environmental concerns related

*Preliminary results of this work were presented at 2004 ACM CODES+ISSS [40] and 2004 ACM EMSOFT [41] conferences.

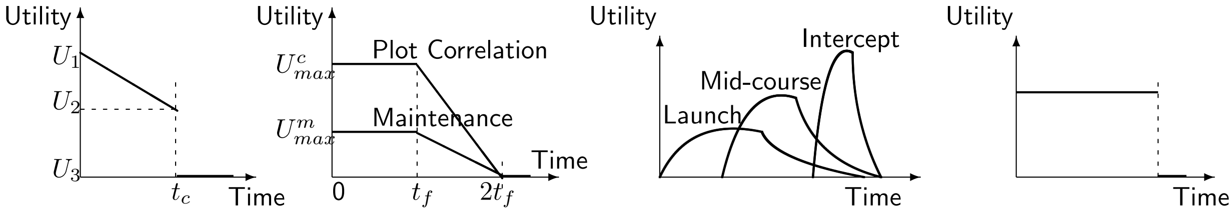
to desktops and servers. For mobile and portable embedded systems, minimizing energy consumption results in longer battery life. But intelligent devices usually need powerful processors, which consume more energy than those in simpler devices, thus reducing battery life. This fundamental trade-off between performance and battery life is critically important and has been addressed in the past [17, 30].

Past research addressed the issue of minimizing power in a given platform, which usually translates into minimizing energy consumption, and thus, longer battery life. Saving energy without substantially affecting application performance is crucial for embedded real-time systems that are mobile and battery-powered, because most real-time 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 [3, 11, 15, 16, 21, 26, 31–33, 39]. 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 [5]; 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 [14].

1.1 TUFs and UA Scheduling

In this paper, we focus on dynamic, adaptive, embedded real-time control systems at any level(s) of an enterprise—e.g., devices in the defense domain from phased array radars [13] up to entire battle management systems [12]. Such embedded systems include time constraints that are "soft" (besides those that are hard) in the sense that 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. 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. The optimality of the soft time



(a) Track Asso. (b) Corrln. & Maint. (c) Missile Control (d) A Step TUF

Figure 1: Example Time Constraints Specified Using Time/Utility Functions

constraints is generally as mission- and safety-critical as that of the hard time constraints.

Jensen's time/utility functions [20] (or TUFs) allow the semantics of soft time constraints to be precisely specified. A TUF, which generalizes the deadline constraint, specifies the utility to the system that is obtained from the completion of an activity as a function of that activity's completion time.

Figures 1(a), 1(b), and 1(c) show time constraints of two large-scale, dynamic, embedded real-time applications specified using TUFs. The applications include: (1) the AWACS (Airborne WArning and Control System) surveillance tracker [7] built by The MITRE Corporation and The Open Group; and (2) a coastal air defense system [29] built by General Dynamics and Carnegie Mellon University. Figure 1(a) shows the TUF of the *track association* activity of the AWACS; Figures 1(b) and 1(c) show TUFs of three activities of the coastal air defense system called *plot correlation*, *track maintenance*, and *missile control*. Note that Figure 1(c) shows how the TUF of the missile control activity dynamically changes as the guided interceptor missile approaches its target. The classical deadline constraint is a binary-valued, downward "step" shaped TUF. This is shown in Figure 1(d).

When activity time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing accrued utility from those activities—e.g., maximizing the sum of the activities' attained 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 respecting mutual exclusion constraints on shared (non-CPU) resources and precedence constraints. Several UA scheduling algorithms are presented in the literature. Examples include [6, 8, 22, 23, 25, 37].

1.2 System-Level Energy Consumption

Most of the past work on energy-efficient real-time scheduling using DVS only considers the CPU's energy consumption. However, the battery life of a system is determined by the *system's* energy consumption, and not just the CPU's energy consumption. Therefore, energy consumption models used in past efforts are not accurate for prolonging 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 [27, 28]. 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 2.6.

1.3 Contributions and Paper Outline

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 [3,14,32,38]), 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 [33, 38, 43].

The work in [33] considers the optimality criterion of maximizing collective value, where value is equivalent to our utility notion. However, [33] is restricted to step value functions or step TUFs (see Figure 1(d)). The work in [38] considers non-step TUFs, but it is restricted to resource-independent activities. The work in [43] considers voltage scheduling for periodic real-time tasks with non-preemptible blocking sections. However, [43] is restricted to deadlines and deadline-based timeliness optimality. The work in [38] considers system-level energy consumption, but it is restricted to resource-independent activities.

In this paper, we consider the problem that intersects: (1) UA scheduling under TUF time constraints, providing assurances on activity timeliness behavior; (2) scheduling activities under mutual exclusion resource constraints; and (3) CPU scheduling for reduced system-level energy consumption. This overlapped problem space has not been previously studied.

We consider repeatedly occurring application activities that are subject to TUF time constraints. Activities may share non-CPU resources which may be subject to mutual exclusion constraints. We consider statistical timeliness performance requirements including lower bounds on individual activity timeliness utilities that must be probabilistically satisfied. To better account for uncertainties in activity execution behaviors, we consider a stochastic model, where activity execution demand is stochastically expressed.

We consider Martin's system-level energy consumption model [27], where each system component's energy consumption is individually modeled and aggregated to obtain system-level energy consumption. We integrate run-time-based DVS [15,26,32] with UA scheduling using a single, system-level performance metric called *Utility and Energy Ratio* (or UER). UER facilitates optimization of timeliness objectives and energy efficiency in a unified way.

Given the metric of UER, our scheduling objective is two-fold: (1) satisfy the lower bounds on individual activity timeliness utilities; and (2) maximize the system's UER. This problem has not been studied in the past and is $\mathcal{NP}$-hard.

We present a polynomial-time, heuristic algorithm for this problem called the *Resource-constrained Energy-Efficient Utility Accrual Algorithm* (or ReUA). We analytically establish several timeliness and non-timeliness properties of the algorithm including optimal timeliness during under-loads, sufficiency on probabilistic satisfaction of timeliness utility lower bounds, lower-bounded system-wide utility, bounded blocking time for resource access, deadlock-freedom, and correctness. We also evaluate ReUA's performance through simulation. Our simulation studies confirm that ReUA provides statistical performance assurances on activity timeliness behavior. Further, the studies reveal that ReUA exhibits superior timeliness performance and system-level energy-efficiency under a broad range of conditions including resource overloads and energy settings where non-CPU devices consume power.

Thus, the contribution of the paper is the ReUA algorithm. To the best of our knowledge, we are not aware of any other efforts that solve the problem solved by ReUA.

The rest of the paper is organized as follows: In Section 2, we outline our activity, resource, and timeliness models, and the scheduling optimality criterion. We present ReUA in Section 3. In Section 4, we establish the algorithm's timeliness and non-timeliness properties. Section 5 discusses the simulation studies. Finally, we conclude the paper in Section 6.

2 Models and Objectives

2.1 Tasks and Jobs

We consider the application to consist of a set of tasks, denoted as $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$. Each task T_i has a number of instances, and these instances may be released either periodically or sporadically with a known minimal inter-arrival time. The period or minimal inter-arrival time of a task T_i is denoted as P_i.

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 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.

2.2 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) [34] and that for UA algorithms [8,23], 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.

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 [8,23], we allow such precedences to be programmed as resource dependencies.

2.3 Timeliness Model

A job's time constraint is specified using a TUF. Following [19], 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 [19, 23]. Thus, a thread 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.

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 Figure 1. TUFs which are not unimodal are multimodal. In this paper, we restrict our focus to *non-increasing*, unimodal TUFs i.e., those unimodal TUFs for which utility never increases as time advances. Figures 1(a), 1(b), and 1(d) show examples. Later, we justify this restriction in Section 2.4.

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 paper, 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 a job's termination is reached and its execution has not been completed, an exception is raised. Normally, this will cause the job's abortion and execution of exception handlers.

2.4 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 paper, we care about the upper bound. For this reason, we focus on non-increasing TUFs.

2.5 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 [7]. 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' demands can provide more realistic statistical performance assurances and more cost-effective resource utilization. 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 a task T_i's cycle demand. Estimating the demand distribution of the task 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 [2, 35, 44]. 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 $e_i = \frac{E(Y_i)}{f}$.

2.6 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 [27, 28, 38]. 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 [9]. 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}, 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, this term may appear because of variations in DC-DC regulator efficiency across the range of output power, CMOS leakage currents, and other second order effects [27].

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: $E_i = P \times e_i$, where e_i denotes T_i's expected execution time. Thus, the expected energy consumption per cycle is given by:

$$E(f) = S_3 \times f^2 + S_2 \times f + S_1 + \frac{S_0}{f} \tag{1}$$

2.7 Scheduling Optimality Criterion

Given the models previously described, we consider the UER metric to integrate timeliness performance and energy consumption. The UER of a job 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.

We define the *system-level* UER as the ratio of the total accrued utilities and total consumed energy of the system i.e., $UER = \frac{\sum_{i=1}^{n} U_i}{\sum_{i=1}^{n} E_i}$.

Thus, the ReUA algorithm that we present considers 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 the specified probability ρ_i; and (2) maximize the *system-level* UER, which implies the system's "energy efficiency."

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 [8].

3 The ReUA Algorithm

3.1 Determining Task Critical Time

To assure that tasks accrue their desired utility percentage and maximize the energy efficiency, ReUA 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 the task's statistical performance requirement can be represented as $Pr\left(U_i(s_{i,j}) \geq \nu_i \times U_i^{max}\right) \geq \rho_i$. 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. We call D_i "critical time" hereafter, and it 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. 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.

3.2 Statistical Estimation of Demand

ReUA's next step is to decide the number of cycles that must be allocated to each task. To provide statistical timeliness assurances while maximizing energy efficiency, ReUA 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, when $y \geq E(Y_i)$, we have:

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

Equation 2 is the direct result of the cumulative distribution function of the task T_i's cycle demands. Knowing that each job $J_{i,j}$ of task T_i should 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., $Pr[Y_i < C_i] \geq \rho_i$.

3.3 UA Scheduling with DVS

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.

The intuitive idea is to assign a uniform speed to execute all tasks until the task set changes. Assume that there are n tasks and each task is allocated C_i cycles within its D_i. The aggregate CPU demand of the task set is:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \tag{3}$$

million cycles per second (MHz). To meet this aggregate demand, the CPU only needs to run at speed $\sum_{i=1}^{n} \frac{C_i}{D_i}$. Equation 3 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 [3,32,42], because it saves more energy for jobs that complete early, and most jobs in its studied application use fewer cycles than allocated.

We consider the energy consumed by the *system* instead of that by just the processor and seek to maximize energy efficiency UER. Equation 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.

We assume that the processor can be operated at m frequencies $\{f_1, f_2, \cdots, f_m | f_1 < \cdots < f_m\}$. ReUA first decides the optimal frequency for each task T_i that maximizes the task's local UER. At each scheduling event, for all the n' jobs $\mathcal{J}_r = \{J_1, J_2, \cdots, J_{n'}\}$ currently in the scheduling queue, ReUA sorts them based on their UERs under the highest frequency f_m, in a non-increasing order. The algorithm then inserts the jobs into a tentative schedule in the order of their critical times (earliest critical time first), while respecting their resource dependencies.

We define the *system load* (Load) as

$$Load = \frac{1}{f_m} \sum_{i=1}^{n} \frac{C_i}{P_i} \tag{4}$$

and define the *critical time-based load* (Cload) as

$$Cload = \frac{1}{f_m} \sum_{i=1}^{n} \frac{C_i}{D_i} \tag{5}$$

For downward step TUFs, $Cload = Load$.

If the system is overloaded, it is possible that the queue $\mathcal{J}_r$, whose *queue load* (Qload) is 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. Thus, upon inserting a job, ReUA checks the tentative schedule's feasibility and ensures feasibility by dropping some jobs; that is, the predicted completion time of each job in the tentative schedule never exceeds its termination time.

To calculate a CPU frequency for the currently selected job i.e., the one at the head of the tentative schedule, we adopt a stochastic DVS technique similar to the Look-Ahead EDF (LaEDF) technique discussed in [32]. The calculated value is compared with the job's local optimal frequency, and the higher one is selected as the CPU frequency. This process is elaborated in Section 3.4.

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 3 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 section.

3.4 Procedural Description

3.4.1 Overview

The scheduling events of ReUA 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. To describe ReUA, we define the following variables and auxiliary functions:

- $\mathbf{T}$ is the task set. D_i^a is task T_i's current invocation's absolute critical time; C_i^r denotes its remaining computation cycles for the current job.
- $\mathcal{J}_r$ is the current unscheduled job set; σ is the ordered 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; $J_k.C$ is its remaining cycle.
- $T(J_k)$ 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. `selectFreq(`x`)` returns the lowest frequency $f_i \in \{f_1, f_2, \cdots, f_m \big| f_1 < \cdots < f_m\}$, 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 with 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, determined under the highest

frequency f_m, must not exceed its termination time.

A description of ReUA at a high level of abstraction is shown in Algorithm 3.1. The procedure `offlineComputing()` of line 3 is shown in Algorithm 3.2, which calculates D_i and C_i for each task. It also computes the optimal frequency $f^o_{T_i}$ for each task T_i that maximizes the task LoUER. LoUER is defined as $U_i(t+\frac{C_i}{f})\big/(C_i \times E(f))$, where $E(f)$ is derived using Equation 1. This calculation is performed at $t = 0$.

Algorithm 3.1: ReUA: High Level Description

1: **input** : $\mathbf{T} = \{T_1, \cdots, T_n\}$, $\mathcal{J}_r = \{J_1, \cdots, J_{n'}\}$;
2: **output** : selected job J_{exe} and frequency f_{exe};
3: `offlineComputing` $(\mathbf{T})$;
4: *Initialization:* $t := t_{cur}$, $\sigma := \emptyset$;
5: **switch** *triggering event* **do**
6:     **case** *task_release(T_i)* $C^r_i = C_i$;
7:     **case** *task_completion(T_i)* $C^r_i = 0$;
8:     **otherwise** Update C^r_i;
9: **for** $\forall J_k \in \mathcal{J}_r$ **do**
10:     **if** `feasible`(J_k)=*false* **then**
11:         `abort`(J_k);
12:     **else**
13:         $J_k.Dep$:= `buildDep`(J_k);
14: **for** $\forall J_k \in \mathcal{J}_r$ **do**
15:     $J_k.UER$:=`calculateUER`(J_k, t);
16: σ_{tmp} :=`sortByUER`$(\mathcal{J}_r)$;
17: **for** $\forall J_k \in \sigma_{tmp}$ *from head to tail* **do**
18:     **if** $J_k.UER > 0$ **then**
19:         σ := `insertByECF`(σ, J_k);
20:     **else**
21:         **break**;
22: J_{exe}:=`headOf`(σ);
23: f_{exe}:=`decideFreq`$(\mathbf{T}, J_{exe}, t)$;
24: **return** J_{exe} and f_{exe};

When ReUA is invoked at time t_{cur}, the algorithm first updates each task's remaining cycle (the `switch` starting from line 5). The algorithm then checks the feasibility of the jobs. If the earliest predicted completion time of a job is later than its termination time, it can be safely aborted (line 11). We assume that aborted tasks are free, in the sense that they neither contribute or detract from utility directly. Thus, an aborted task accrues zero utility. Otherwise, ReUA builds the dependency list for the job (line 13).

The UER of each job is computed by `calculateUER()`, and the jobs are then sorted by their UERs (line 15 and 16). In each step of the `for` loop from line 17 to 21, the job with the largest UER and its dependencies are inserted into σ, if it can produce a positive UER. The output schedule σ is then sorted by the jobs' critical times by the procedure `insertByECF()`.

Algorithm 3.2: `offlineComputing()`

input : Task set $\mathbf{T}$;
output : D_i, C_i, $f^o_{T_i}$;
$D_i = U_i^{-1}(\nu_i \times U_i^{max})$;
$C_i = E(Y_i) + \sqrt{\frac{\rho_i \times Var(Y_i)}{1-\rho_i}}$;
Decide $f^o_{T_i}$, such that $U_i(\frac{C_i}{f^o_{T_i}}) / \left(C_i \times E(f^o_{T_i})\right) = \texttt{max}(U_i(\frac{C_i}{f_j}) / (C_i \times E(f_j))), \forall j \in \{1, 2, \cdots, m\}$;

Finally, ReUA analyzes the demands of the task set and applies DVS to decide the CPU frequency f_{exe} (line 23). The selected job J_{exe}, which is at the head of σ, is executed at f_{exe} (line 22–24).

3.4.2 Resource and Deadlock Handling

Before ReUA can compute job partial schedules, the dependency chain of each job must be determined, as shown in Algorithm 3.3.

Algorithm 3.3 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 3.3 stops either because a predecessor job does not need any resource or the requested resource is free. Note that "**·**" denotes an append operation. Thus, the dependency list starts with J_k 's farthest predecessor and ends with J_k.

Algorithm 3.3: `buildDep()`: Build Dependency List

input : Job J_k;
output : $J_k.Dep$;
Initialization : $J_k.Dep := J_k$; $Prev := J_k$;
while $\mathtt{reqRes}(Prev) \neq \emptyset \bigwedge \mathtt{owner}(\mathtt{reqRes}(Prev)) \neq \emptyset$ **do**
    /* add new owner at the head of the list */
    $J_k.Dep :=$ `owner`$(\mathtt{reqRes}(Prev)) \cdot J_k.Dep$;
    $Prev :=$ `owner`$(\mathtt{reqRes}(Prev))$;

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 [34] assumes that the highest priority of jobs accessing a resource is known. Likewise, the Stack Resource policy [4] 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 3.4: Deadlock Detection and Resolution

input : Requesting job J_k, t_{cur};
`/* deadlock detection */`
$Deadlock :=$ **false**;
$J_l :=$ `owner(`$reqRes(J_k)$`)`;
while $J_l \neq \emptyset$ **do**
  $J_l.LoUER := U_{J_l}(t_{cur} + \frac{J_l.C}{f_m})/(J_l.C \times E(f_m))$;
  if $J_l = J_k$ **then**
    $Deadlock :=$ **true**;
    break;
  else
    $J_l :=$ `owner(`$reqRes(J_l)$`)`;
`/* deadlock resolution if any */`
if $Deadlock =$ **true then**
  `abort(`*The job J_m with the minimal $LoUER$ in the cycle*`)`;

The deadlock detection and resolution algorithm (Algorithm 3.4) 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. ReUA aborts the job with the minimal LoUER in the cycle to resolve a deadlock.

3.4.3 Manipulating Partial Schedules

The `calculateUER()` algorithm (Algorithm 3.5) 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 3.5: `calculateUER()`

1: **input** : J_k, t_{cur};
2: **output** : $J_k.UER$;
3: *Initialization :* $C_c := 0$, $E := 0$, $U := 0$;
4: **for** $\forall J_l \in J_k.Dep$, *from head to tail* **do**
5:     $C_c := C_c + J_l.C$;
6:     $U := U + U_{J_l}(t_{cur} + \frac{C_c}{f_m})$;
7: $E := E(f_m) \times C_c$;
8: $J_k.UER := U/E$;
9: **return** $J_k.UER$;

To compute J_k's UER at time t_{cur}, ReUA 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 19 of Algorithm 3.1 are shown in Algorithm 3.6. `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.

Algorithm 3.6: `insertByECF()`

1: **input** : J_k and an ordered job list σ;
2: **output** : the updated list σ;
3: **if** $J_k \notin \sigma$ **then**
4:     copy σ into σ_{tent}: $\sigma_{tent} := \sigma$;
5:     `insert`(J_k, σ_{tent}, $J_k.D$);
6:     $CuCT = J_k.D$;
7:     **for** $\forall J_l \in \{J_k.Dep - J_k\}$ *from tail to head* **do**
8:         **if** $J_l \in \sigma_{tent}$ **then**
9:             CT=`lookup`(J_l, σ_{tent});
10:             **if** $CT < CuCT$ **then continue**;
11:             **else** `remove`(J_l, σ_{tent}, CT);
12:         $CuCT$:=`min`($CuCT$, $J_l.D$);
13:         `insert`(J_l, σ_{tent}, $CuCT$);
14:     **if** *feasible*(σ_{tent}) **then**
15:         $\sigma := \sigma_{tent}$;
16: **return** σ;

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 13 of Algorithm 3.6.

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 3.6). 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 $J_k.Dep$ must complete by then (lines 12–13, Algorithm 3.6). Finally, if this insertion produces a feasible schedule, then the jobs are included in the schedule; otherwise, not (lines 14–15).

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 jobs' critical time is smaller or equal to its termination time. 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 5.3.

3.4.4 Deciding the Processor Frequency

ReUA applies the stochastic DVS technique LaEDF [32], and extends it to deal with overloads. The strategy is shown in Algorithm 3.7.

ReUA keeps track of the remaining computation cycles C_i^r, as updated from line 5 to line 8 of Algorithm 3.1. Unlike LaEDF, ReUA uses the aggregate CPU demand shown in Equation 3 during the process of DVS. From line 3 to line 10, the algorithm considers the interval until the next task critical time and tries to "push" as much work as possible beyond the critical time. ReUA considers the tasks in the latest-critical-time-first order in line 5.

x is the minimum number of cycles that the task must execute before the closest critical time, D_n^a, in order for it to complete by its own critical time (line 7), assuming worst-case aggregate CPU demand CPU_{dmd} by tasks with earlier critical times. The aggregate demand CPU_{dmd} is adjusted to reflect the actual demand of the task for the time after D_n^a (line 8). 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 all tasks to meet their critical times (line 9). In line 10, the CPU frequency is set just fast enough to execute s cycles over this interval.

Algorithm 3.7: `decideFreq()`

1: **input** : $\mathbf{T}$, J_{exe}, t_{cur};
2: **output** : f_{exe};
3: $CPU_{dmd} := CPU_{dmd}^{bak} := C_1/D_1 + \cdots + C_n/D_n$;
4: $s := 0$;
5: **for** $i = 1$ *to* n, $T_i \in \{T_1, \cdots, T_n \big| D_1^a \geq \cdots \geq D_n^a\}$ **do**
   `/* reverse EDF order of tasks */`
6: $CPU_{dmd} := CPU_{dmd} - C_i/D_i$;
7: $x :=$`max`$(0,\ C_i^r - (f_m - CPU_{dmd}) \times (D_i^a - D_n^a))$;
8: $CPU_{dmd} := \begin{cases} CPU_{dmd}^{bak}, & \text{if } D_i^a - D_n^a = 0 \\ CPU_{dmd} + \frac{C_i^r - x}{D_i^a - D_n^a}, & \text{otherwise} \end{cases}$;
9: $s := s + x$;
10: $f :=$`min`$(f_m,\ s/(D_n^a - t_{cur}))$;
11: $f_{exe}:=$`selectFreq` (f);
12: $f_{exe}:=$`max`$(f_{exe},\ f_{T(J_{exe})}^o)$;

Thus, `decideFreq()` capitalizes on early task completion by deferring work for future tasks in favor of scaling the current task. In addition, in line 8, we consider the case that jobs of different tasks have the same absolute critical times, which sometimes occurs, especially during overloads. Also, it is possible that during overloads, the required frequency may be higher than f_m and `selectFreq()` would fail to return a value. In line 10, we solve this by setting the upper limit of the required frequency to be f_m.

Finally, the result of `selectFreq()` is compared with the optimal frequency of $T(J_{exe})$ decided in `offlineComputing()` (line 12). The higher frequency is selected to preserve the statistical performance assurance and maximize system-level UER.

3.5 Computational Complexity

To analyze the complexity of ReUA (Algorithm 3.1), we assume that m, the available number of frequencies for the processor is limited, and thus does not affect the complexity significantly. Then 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 9 to 13 requires $O(n^2)$ time. Also, the `for`-loop containing `calculateUER()` (line 14–15) 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 17 is dominated by `insertByECF()`

(Algorithm 3.6). Its complexity is dominated by the `for`-loop (line 7–13, Algorithm 3.6), 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 RUA algorithm is $2 \times O(n^2) + O(n \log n) + n \times O(n \log n) = O(n^2 \log n)$.

4 Algorithm Properties

4.1 Non-Timeliness Properties

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

ReUA 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 1 *ReUA 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. ReUA does not allow such a cycle by deadlock detection and resolution; so it is deadlock free. □

Lemma 1 *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 $J_k.D$ by the operation with $CuCT$. According to the definition of `insert()`, after advancing the critical time, J_l will be inserted before J_k. □

Theorem 2 *When a job J_k that requests a resource R is selected for execution by ReUA, J_k's requested resource R will be free. We call this ReUA's correctness property.*

Proof From Lemma 1, the output schedule σ is correct. Thus, ReUA is correct. $\square$

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 ReUA's mutual exclusion property by a corollary.

Corollary 1 *ReUA satisfies mutual exclusion constraints in resource operations.*

4.2 Timeliness Properties

With Corollary 1, 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 ReUA.

Theorem 3 *Under ReUA, 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) [34]. In Algorithm 3.6, 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 13, 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 [34]. $\square$

We also consider timeliness properties under no resource dependencies, where ReUA can be compared with a number of well-known algorithms. Specifically, we consider the following two conditions: (1) a set of independent periodic tasks, where each task has a single computational thread with a downward step TUF (such as the one shown in Figure 1(d)); and (2) there are sufficient processor cycles for meeting all task termination times—i.e., there is no overload.

Theorem 4 *Under conditions (1) and (2), a schedule produced by EDF [18] is also produced by ReUA, yielding equal total utilities. Not coincidentally, this is simply a termination time ordered schedule.*

Proof We prove this by examining Algorithms 3.1 and 3.6. 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 19 of Algorithm 3.1 is termination time ordered.

The TUF termination time that we consider is analogous to a deadline in [18]. As proved in [18,24], 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. $\square$

Some important corollaries about ReUA's timeliness behavior during non-overload situations can be deduced from EDF's optimality [10].

Corollary 2 *Under conditions (1) and (2), ReUA always meets all task termination times.*

Corollary 3 *Under conditions (1) and (2), ReUA yields the minimum possible maximum lateness.*

ReUA also provides statistical performance assurances under possible conditions. With condition (1), the utility requirement of a task can only take $\nu = 0$ or $\nu = 1$. From Corollary 2, we can derive the properties of ReUA on performance assurances.

Theorem 5 *Under conditions (1) and (2), ReUA meets all statistical performance requirements.*

Proof From Corollary 2, under conditions (1) and (2), ReUA can meet all task termination times. This ensures that $\nu_i = 1$ can be satisfied for each task. Based on the results of Equation 2, at least ρ_i demanded processor cycles of task T_i are less than the allocated cycles. From Corollary 2, all the allocated cycles can be completed before their termination times. Thus, for task T_i, ReUA can meet at least ρ_i termination times; i.e., ReUA accrues ν_i utility with a probability at least ρ_i. $\square$

From Theorem 5, we can derive its counterpart for non-increasing TUFs with the definitions of Equations 4 and 5.

Theorem 6 *For a set of independent periodic tasks, where each task has a single computational thread with a non-increasing TUF, $Cload \leq 1$ is the sufficient condition for ReUA to meet all statistical performance requirements.*

Proof With ν_i and ρ_i of task T_i, ReUA converts the performance assurance problem to the problem of meeting critical times. If $Cload \leq 1$, according to the result of Theorem 5, the assertion holds. □

Note that Theorem 6 only states that $Cload \leq 1$ is the sufficient condition. Actually, it is not the necessary condition. We illustrate this with an example in Section 5.

We also establish the relationship between task-level assurances and the system-level utility ratio in Theorem 7.

Theorem 7 *For a set of independent periodic tasks, if ReUA 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 ReUA 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 $\frac{\sum_{i=1}^{n} \rho_i \nu_i U_i^{max}}{\sum_{i=1}^{n} U_i^{max}}$. □

5 Experimental Results

In order to experimentally evaluate the performance of ReUA, we developed a simulator for the operation of hardware capable of DVS, and performed extensive simulations. We first present the simulation methodology, and then discuss the results.

5.1 Simulation Methodology

Our simulator is written with the simulation tool OMNET++ [36], which provides a discrete event simulation environment. The simulator takes as input a task set, specified with the period or minimum inter-arrival time (abbreviated as P/I.A.), and real-time requirements. The tasks' time constraints i.e., means/variances of the cycle demands and TUFs are also specified as the input. The tasks contained in a task set G are selected from Table 1. The table also summarizes these tasks' input parameters.

Table 1: Experimental Task Parameters

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	$\begin{cases} -0.025t^2 + 10, & 0 \le t \le 20 \\ 0, & \text{otherwise} \end{cases}$
T_6	124	22	$\begin{cases} -4x + 80, & 0 \le t \le 20 \\ 0, & \text{otherwise} \end{cases}$
T_7	137	25	$\begin{cases} -0.01x^2 - 0.15x + 10, & 0 \le t \le 25 \\ 0, & \text{otherwise} \end{cases}$
T_8	124	21	$\begin{cases} -0.5x + 10, & 0 \le t \le 20 \\ 0, & \text{otherwise} \end{cases}$
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. For each demand Y_i, we keep $Var(Y_i) \approx E(Y_i)$, and generate normally-distributed cycle demands.

The energy consumption per cycle at a particular frequency is calculated using Equation 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 [27, 28, 38].

We use experimental settings that are similar to that in Martin's PhD thesis [27], but de-normalize the terms. For comparison, the experiments are carried out under three energy model settings, as shown in Table 2. Note that E_1 is the same as the traditional energy model, which only considers the energy consumed by the processor.

Other parameters that are supplied to the simulator include the processor specification.

Table 2: 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	$0.5f_m^3$

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 [1].

In addition to ReUA, we implemented the following schemes for comparison: BaseEDF, LaEDF, StaticEDF, and LaEDF-NA.

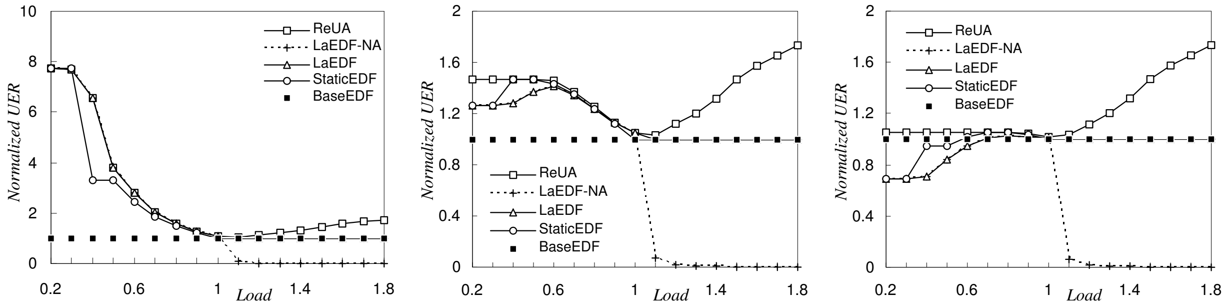
BaseEDF is the EDF scheduler without any DVS support and uses the highest frequency. LaEDF is the Look-ahead RT-DVS for EDF scheduler in [32]. StaticEDF uses the constant speed given by Equation 3 and a "ceiling" up to the lowest suitable frequency in $\{f_1, f_2, \cdots, f_m\}$. StaticEDF switches to the lowest frequency whenever there is no ready task. Combining the static schemes in [3] and [32], 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 ReUA as their inputs.

5.2 Impact of Energy Models

In our first set of simulation experiments, we determine the effects of our new energy model. We consider the task set $G_1 = \{T_1, T_2, T_3, T_4\}$, and apply different schemes on G_1 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 2 shows the UER for all the DVS schemes normalized to the BaseEDF under energy model settings E_1, E_2, and E_3, as *Load* varies from 0.2 to 1.8. We observe that under all three energy settings, ReUA performs the best among all strategies under all loads, especially during overloads. We also observe that LaEDF-NA yields almost zero UER during overloads.

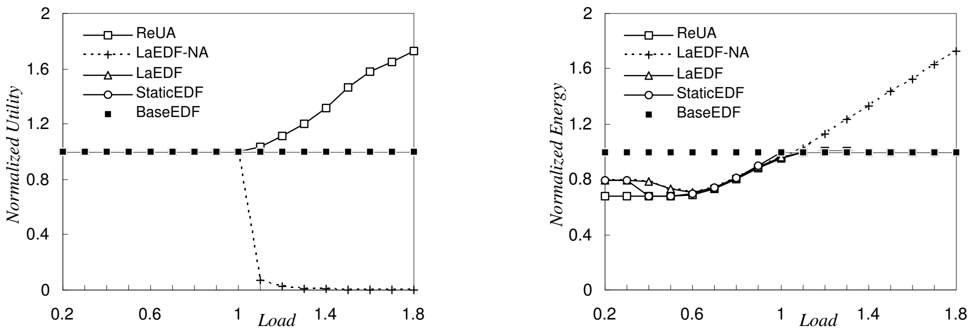


(a) Energy Setting E_1 (b) Energy Setting E_2 (c) Energy Setting E_3

Figure 2: Normalized UER vs. *Load* with Step TUFs under Various Energy Model Settings

As the figure shows, during overloads, the normalized UERs produced by LaEDF, StaticEDF, and BaseEDF converge to 1. This is because, all three algorithms select the highest frequency by DVS calculation during overloads, and bear no difference in scheduling. As the term S_0 in the energy model increases, ReUA adjusts the selected frequency to accrue more UER. This effect is more pronounced under E_3, when LaEDF, LaEDF-NA, and StaticEDF perform worse than BaseEDF, while ReUA still outperforms BaseEDF during all loads.

We speculate that, the UER gap between ReUA and the other schemes is because ReUA saves more energy during under-loads, and accrues higher utility during overloads. Our speculation is verified in Figure 3, which shows the accrued utility and energy consumption normalized to BaseEDF, under energy model setting E_2.



(a) Norm. Utility vs. *Load* (b) Norm. Energy vs. *Load*

Figure 3: Normalized Energy and Utility vs. *Load* with Step TUFs under Energy Setting E_2

From Figure 3(a), we observe that during under-loaded situations, all schemes accrue the same (optimal) utility because of EDF's optimality [10] during such situations. But during overload situations, LaEDF-NA suffers domino effects and accrues almost no utility [25]. On the other hand, ReUA seeks to schedule jobs with higher UERs, and thus accrues remarkably higher utility than the others.

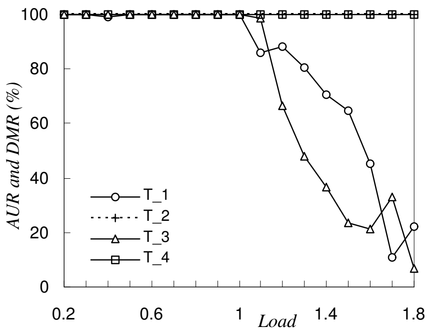
In Figure 3(b), during under-loads, we observe that ReUA saves more energy than the other schemes. Further, this portion of the curves is nearly symmetric to the corresponding portion of Figure 2(b). The energy consumption of LaEDF-NA increases linearly with *Load*, because it performs no abortion and executes every job that arrives.

Since no strategies except ReUA consider the system-level energy consumption, we only use the energy model E_1 in our further simulation experiments.

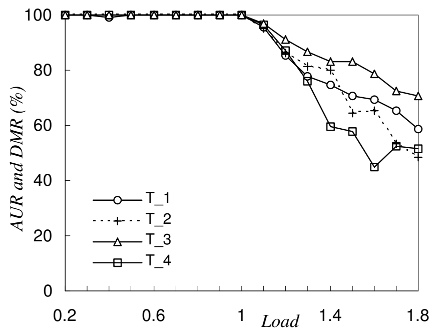
5.3 Statistical Performance Assurance

To evaluate the statistical performance assurances provided by ReUA, we first consider the task set G_1 with the performance requirement of $\{(\nu_i = 1, \rho_i = 0.96), i = 1, \cdots, 4\}$.

Figure 4 shows 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) ReUA



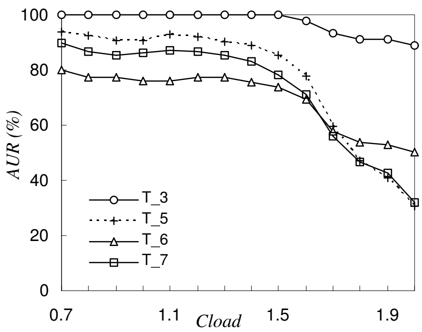
(b) LaEDF

Figure 4: AUR and DMR vs. *Load* of G_1 under E_1

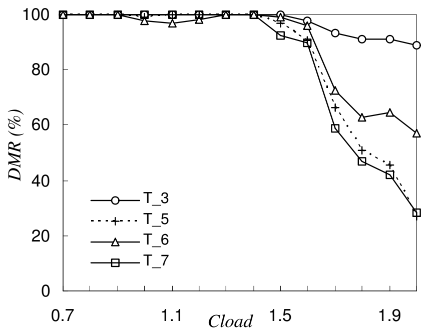
As Figure 4(a) shows, with ReUA during under-loads, all tasks accrue 100% AUR and DMR, except task T_1, whose AUR and DMR is 99.23% at $Load = 0.3$. Thus, ReUA delivers the statistical performance assurance of being able to accrue 100% of task maximum utility with a probability at least 96% for all tasks. This also validates Theorem 5.

Comparing the results during overloads in Figure 4(a) and 4(b), we observe that ReUA still achieves near 100% AUR/DMR of task T_2 and T_4, but achieves less AUR/DMR of T_1 and T_3. One the other hand, LaEDF decreases the AUR/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 ReUA 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 5.4.

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 $\{(\nu_3 = 1.0, \rho_3 = 0.80), (\nu_5 = 0.55, \rho_5 = 0.80), (\nu_6 = 0.5, \rho_6 = 0.80), (\nu_7 = 0.55, \rho_7 = 0.80)\}$.



(a) AUR vs. *Cload*



(b) DMR vs. *Cload*

Figure 5: AUR and DMR vs. *Cload* of G_2 under E_1

Figure 5 shows the AUR and DMR of each task in G_2 with *Cload* varying from 0.7 to 2.0. System *Load* also changes with *Cload*, and the corresponding values are shown in Table 3.

Table 3: *Cload* and *Load*

Cload	0.7	0.8	0.9	1.0	1.1	1.2	1.3
Load	0.44	0.5	0.57	0.6	0.7	0.76	0.83
Cload	1.4	1.5	1.6	1.7	1.8	1.9	2.0
Load	0.89	0.95	1.01	1.06	1.13	1.2	1.26

We consider task T_7 as an example to illustrate how ReUA delivers statistical performance assurances. As shown in Figure 5, when $Cload \leq 1$, 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 $Cload = 1$, ReUA accrues AUR=86.97% and DMR=100%, which implies that it can complete all the demanded cycles of the task before their critical times. Furthermore, 86.97% of the task maximum utility can be accrued at a probability 100%—much more than the performance requirements.

But $Cload \leq 1$ is not the necessary condition for delivering statistical performance assurances. For example, at $Cload = 1.6$ and $Load = 1.02$, 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 6.

Another major pattern that can be observed from Figure 5 is that, as $Cload$ and $Load$ 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 ReUA to accrue more system-wide utility.

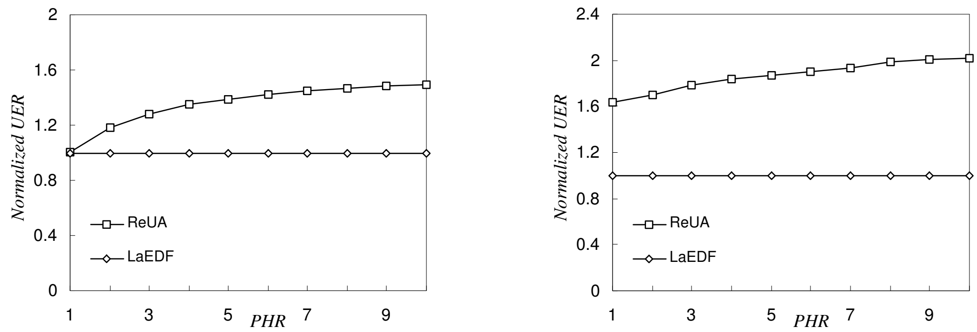
5.4 Utility Accrual Effectiveness

From experiments of the previous sections, we observe that ReUA 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. Thus, the higher UER produced by ReUA than the others is due to the fact that ReUA seeks to accrue more utility during such situations. In this section, we vary the TUF shape of each task to demonstrate ReUA's utility accrual capability.

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(a) shows the UERs for ReUA and LaEDF that are normalized to 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 $PHR = 1$, ReUA makes the same scheduling decisions as LaEDF. But as PHR increases, ReUA obtains higher system-level UER than LaEDF.



(a) G_3: Step TUFs (b) G_4: Linear TUFs

Figure 6: Normalized UER vs. PHR under E_1

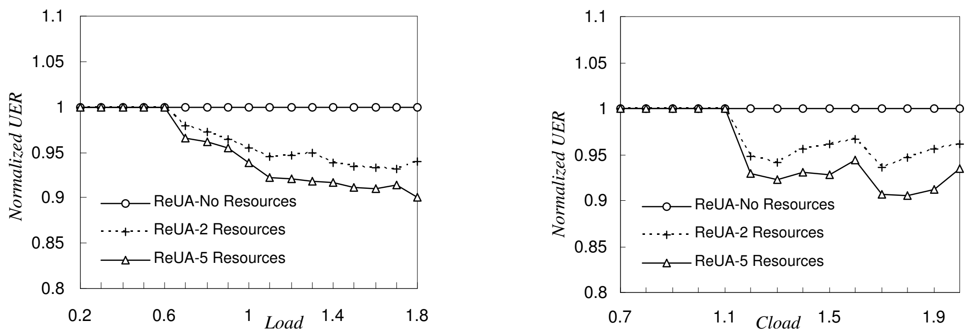
Figure 6(b) shows the normalized UERs for ReUA and LaEDF under G_4 with $Load = 1.5$ and $Cload = 1.85$. We observe similar trends as that in Figure 6(a), but with larger performance gap as PHR increases. The two strategies' different scheduling criteria result in different performance even at $PHR = 1$.

Since not all critical times can be satisfied during overloads, ReUA 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, ReUA 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.

5.5 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 a pool of resources during the job's lifetime. The resource request and release times are uniformly distributed within a job's lifetime.

We conducted experiments on the task sets, which are scheduled by ReUA under no resources, three shared resources, and five shared resources. Figure 7(a) shows UERs normalized to the case of G_1 with no resources, as *Load* varies from 0.2 to 1.8. Figure 7(a) shows the same metric for G_2, as *Cload* varies from 0.7 to 2.0.



(a) UER vs. *Load* of G_1 (b) UER vs. *Cload* of G_2

Figure 7: Normalized UER with Resource Dependencies under E_1

From the figures, we observe that when *Load* or *Cload* increases, the performance of ReUA on dependent task sets decreases. Higher the number of shared resources, the more performance decrease can be observed. This is because, ReUA respects resource dependencies in scheduling, which in the worst-case may cause jobs to be executed in the reverse order of UERs or critical times. So with dependent task sets, ReUA cannot provide performance assurances and suffers UER losses, especially during high loads.

However, at very high *Load* or *Cload* and with five shared resources, normalized UERs of ReUA on the independent task sets are just better than those on dependent task sets by no more than 10%. This is because, ReUA aborts a task when its expected completion time is less than its termination time. Thus, the job queue seen by ReUA 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 task sets.

6 Conclusions, Future Work

This paper presents the design and evaluation of ReUA, a resource-constrained, energy-efficient, utility-accrual real-time scheduling algorithm for mobile embedded systems. ReUA considers application activities that are subject to TUF time constraints, resource dependencies, and system-level energy consumption concerns.

The key underpinning of ReUA 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. ReUA makes such stochastic decisions based on the statistical properties of the task demands. During overload situations, the algorithm heuristically schedules tasks to maximize collective utility.

We establish ReUA's several timeliness and non-timeliness properties such as optimal timeliness during under-loads under step TUFs, sufficiency on probabilistic satisfaction of timeliness utility lower bounds, lower-bounded, system-wide total utility, upper bounded blocking time for resource access, deadlock-freedom, correctness, and mutual exclusion. Our simulation experiments confirm that ReUA provides statistical performance assurances on activity timeliness behavior. Further, the studies reveal that ReUA exhibits superior timeliness performance and system-level energy-efficiency under a broad range of conditions including resource overload situations, and energy settings where non-CPU devices also consume power besides the CPU.

Several aspects of the work are interesting directions for further research. One direction is to consider an upper bound on energy consumption as an optimization constraint. Another direction is to consider tasks with more general arrival patterns.

Acknowledgements

This work was supported by the U.S. Office of Naval Research under Grant N00014-00-1-0549 and by The MITRE Corporation under Grant 52917.

References

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

[2] 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*, pages 1–14, October 1997.

[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*, pages 95 –105, December 2001.

[4] T. P. Baker. Stack-based Scheduling of Real-Time Processes. *Journal of Real-Time Systems*, 3(1):67–99, March 1991.

[5] A. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power CMOS Digital Design. In *IEEE Journal of Solid-State Circuits*, volume 27, pages 473–484, April 1992.

[6] K. Chen and P. Muhlethaler. A Scheduling Algorithm for Tasks Described by Time Value Function. *Journal of Real-Time Systems*, 10(3):293–312, May 1996.

[7] 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*, volume 1586 of *LNCS*, pages 353–362. Springer-Verlag, April 1999.

[8] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990. CMU-CS-90-155, `http://www.real-time.org` (last accessed: October 22, 2004).

[9] D. M. D. Zhu, N. AbouGhazaleh 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.

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

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

[12] GlobalSecurity.org. BMC3I Battle Management, Command, Control, Communications and Intelligence. `http://www.globalsecurity.org/space/systems/bmc3i.htm/` (last accessed: October 22, 2004).

[13] GlobalSecurity.org. Multi-Platform Radar Technology Insertion Program. `http://www.globalsecurity.org/intell/systems/mp-rtip.htm/` (last accessed: October 22, 2004).

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

[15] 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.

[16] 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.

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

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

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

[20] 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*, pages 112–122, December 1985.

[21] 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.

[22] 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*, pages 290–299, December 1992.

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

[24] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[25] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986. CMU-CS-86-134, `http://www.real-time.org` (last accessed: October 22, 2004).

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

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

[28] T. Martin and D. Siewiorek. Non-ideal Battery and Main Memory Effects on CPU Speed-Setting for Low Power. *IEEE Transactions on VLSI Systems*, 9(1):29–34, February 2001.

[29] 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. Technical report, Department of Computer Science, Carnegie Mellon University, December 1988. Archons Project Technical Report 88121.

[30] M. Pedram. Power Minimization in IC Design: Principles and Applications. In *ACM Transactions on Design Automation of Electronics Systems*, volume 1:1, pages 3–56, January 1996.

[31] 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.

[32] 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*, pages 89–102, 2001.

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

[34] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[35] 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.

[36] A. Varga. OMNeT++ Discrete Event Simulation System. `http://www.omnetpp.org/` (last accessed: October 22, 2004).

[37] 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*, 15(2):119–133, February 2004.

[38] 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*, pages 21–28, May 2003.

[39] 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*, pages 13–23, November 1994.

[40] 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)*, pages 110–115, September 2004.

[41] 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)*, pages 64–73, September 2004.

[42] 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*, pages 149–163. ACM Press, 2003.

[43] F. Zhang and S. T. Chanson. Blocking-Aware Processor Voltage Scheduling for Real-Time Tasks. *ACM Transactions on Embedded Computing Systems*, 3(2):307–335, May 2004.

[44] 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*, pages 15–26, October 1997.