Utility Accrual, Real-Time Scheduling Under the Unimodal Arbitrary Arrival Model with Energy Bounds

Haisang Wu, Binoy Ravindran, and E. Douglas Jensen

Abstract

In this paper, we consider timeliness and energy optimization in battery-powered, dynamic, embedded real-time systems, which must remain functional during an operation/mission with a bounded energy budget. We consider application activities that are subject to time/utility function time constraints, statistical assurance requirements on timeliness behavior, and an energy budget, which cannot be exceeded at run-time. To account for the inevitable variability in activity arrivals in dynamic systems, we describe arrival behaviors using the unimodal arbitrary arrival model (or UAM) [15]. For such a model, we present a DVS (dynamic voltage scaling)-based, CPU scheduling algorithm called *E̲nergy-B̲ounded U̲tility A̲ccrual Algorithm* (or EBUA). Since the scheduling problem is intractable, EBUA allocates CPU cycles, scales clock frequency, and heuristically computes schedules using statistical estimates of cycle demands, in polynomial-time. We analytically establish EBUA's properties including satisfaction of energy bounds, statistical assurances on individual activity timeliness behavior, optimal timeliness during under-loads, and bounded time for mutually exclusively accessing shared non-CPU resources. Our simulation experiments validate our analytical results and illustrate the algorithm's effectiveness and superiority over past algorithms.

Index Terms

Real-time systems, energy-efficient scheduling, time/utility functions, utility accrual scheduling

I. Introduction

With the proliferation of mobile and embedded devices that operate on limited battery power, power and energy management of embedded systems has become critically important. Most of such devices

Preliminary results of this work were published at IEEE/ACM DATE'2005 [39] and ACM SAC'2006 [40].

Haisang Wu is with Juniper Networks, Inc., Sunnyvale, CA 94089. E-mail: `hswu@ieee.org`. This work was performed while he was at Virginia Tech.

Binoy Ravindran is with the Real-Time Systems Laboratory, Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061. E-mail: `binoy@vt.edu`.

E. Douglas Jensen is with the MITRE Corporation, Bedford, MA 01730, E-mail: `jensen@mitre.org`.

have finite energy bounds, embodied by a battery that has a finite lifetime. An important technique used for optimizing the energy consumption of real-time embedded systems is dynamic voltage scaling (DVS). With DVS, an appropriate clock rate and voltage can be determined in response to dynamic application behaviors. This can result in quadratic energy savings at the expense of roughly, linearly increased application activity sojourn times [3], [11], [13], [14], [20], [21], [27], [30], [31], [33], [37], [45].

In this paper, 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 [9]. Such systems are fundamentally time-critical, as they must sense external objects in a timely manner and produce timely control responses (e.g., to avoid obstacles in the physical world).

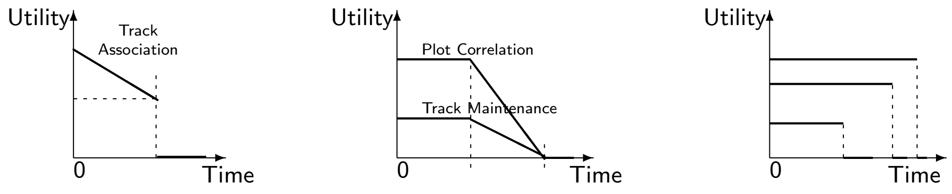
Further, they are energy-critical, as they operate on batteries, and are often subject to a finite energy budget for a mission's duration. This is typically due to the non-availability of battery recharging time and/or energy source. Hence, they must operate without violating their energy budgets. Doing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

Such systems often operate in environments with dynamically uncertain properties, which include transient and sustained resource overloads when application demands (e.g., for CPU cycles) exceed availability. This happens due to context-dependent, activity execution times and arbitrary activity arrival patterns. Nevertheless, they 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.

The most distinguishing property of such systems, however, is that they are subject to time constraints that are "soft" (besides 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 constraints is generally at least as mission-

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

Jensen's time/utility functions [18] (or TUFs) allow the semantics of soft time constraints to be precisely specified. A TUF, which is a generalization of the deadline constraint, specifies the utility to the system resulting from the completion of an activity as a function of its completion time. A TUF's utility values are derived from application-level quality of service metrics. Figures 1(a)–1(b) show some time constraints of two real applications in the defense domain [7], [29] specified using TUFs.[1] The classical deadline is a binary-valued, downward "step" shaped TUF; Figure 1(c) shows examples.



(a) (b) (c)

Fig. 1: Example TUF Time Constraints. (a): AWACS *association* TUF [7]; (b): Air defense *plot correlation* and *track maintenance* TUFs [29]; (c): Some Step TUFs.

When activity time constraints are expressed with TUFs, the timeliness optimality criteria are typically based on accrued activity utility—e.g., maximizing 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.

Note that UA criteria directly facilitate adaptive behaviors during resource overloads, when (optimally or sub-optimally) completing activities that are more important than those which are more urgent is often desirable. UA algorithms that maximize summed utility under downward step TUFs (or deadlines), meet all activity deadlines during under-load situations [8], [26], [41]. When overloads occur, they favor activities that are more important (since more utility can be attained from them) than those which are more urgent. Thus, deadline scheduling's optimal timeliness behavior [10] is a special-case of UA scheduling.

Contributions. In this paper, we consider timeliness and energy optimization in dynamic real-

[1]More real-world TUFs exist, including those with more complex shapes, but they appear in classified US DoD systems and hence are not in the public domain.

time embedded systems with the previously mentioned characteristics. In particular, we focus on their two important aspects: (1) variability in arrival behaviors of application activities; and (2) a finite, *hard* budget for energy consumption for the duration of operational/mission life.

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) [3], [21], [31], [45], and those that consider UA criteria (e.g., maximizing summed utility) [33], [36], [42]–[44]. As far as we know, the only exception is [36], which allows aperiodic arrivals. However, [36] 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 these extremes are inappropriate for the applications/domains of interest to us[2].

In this paper, we bridge these extremes by considering the *u*nimodal *a*rbitrary arrival *m*odel (or UAM) [15]. UAM embodies a "stronger" adversary [22] than most traditional arrival models (e.g., periodic, frame-based, sporadic), and subsumes those models as special cases.

Most of the past efforts on energy-bounded real-time scheduling are restricted to deadlines, step TUFs, and deadline-based timeliness optimality. Examples include [32] (which optimizes total "reward" for step reward functions, where reward is equivalent to our utility notion), [19], and [2]. On the other hand, some energy-efficient real-time scheduling works do not allow energy budgets. These works include those that consider deadlines and deadline-based optimality criteria [3], [12], [31], and those that consider non-step TUFs and UA optimality criteria [36], [39], [42]–[44]. Thus, 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, we bridge these extremes by satisfying energy budgets under UA criteria. We consider repeatedly occurring application activities, which are subject to TUF time constraints. To better account for uncertainties in activity behaviors, we stochastically describe activity execution demands, and describe activity arrivals using UAM. We consider Martin's system-level energy consumption

[2]According to [6], an aperiodic task is a type of task that consists of a sequence of identical jobs (instances), activated at irregular intervals. A sporadic task is an aperiodic task characterized by a minimum inter-arrival time between consecutive instances.

model [28], where each system component's energy consumption is individually modeled and aggregated to account for system's energy consumption.

For such an activity model, we consider the UA objective of: (1) providing statistical assurances on timeliness behavior including probabilistically-satisfied lower bounds on individual activity utility; and (2) maximizing system-level energy efficiency; while ensuring that the system's energy consumption never exceeds a specified energy budget. When the 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.

This problem is $\mathcal{NP}$-hard. We present an algorithm for this problem called *Energy-Bounded Utility Accrual Algorithm* (or EBUA). EBUA allocates CPU cycles, scales clock frequency, and heuristically computes schedules using statistical estimates of cycle demands, in polynomial-time. We prove that EBUA never violates the energy budget, and probabilistically satisfies individual activity utility lower bounds. Further, we establish that EBUA's timeliness behavior subsumes EDF's optimal timeliness behavior [10] as a special case. We also upper bound the time needed for mutually exclusively accessing shared non-CPU resources under EBUA. Finally, our simulation experiments validate our analytical results and illustrate EBUA's effectiveness and superiority over past algorithms including OFC [2] and REW-Pack [32].

Thus, this paper's contribution is the EBUA algorithm. We are not aware of any other efforts that solve the *energy-bounded, TUF/UA scheduling problem under the UAM model* that is solved by EBUA.

The rest of the paper is organized as follows: Section II describes models and assumptions. Section III defines energy-bounded systems, states the scheduling objective, and presents EBUA. Section IV establishes EBUA's timeliness and non-timeliness properties; Section V discusses the simulation studies. We conclude the paper in Section VI.

II. Models and Assumptions

A. Power and Energy Consumption Model

We consider Martin's system-level energy consumption model to derive the energy consumption per CPU cycle (detailed model descriptions can be found in [28], [36], [43], [44]). In this model, when operating at a frequency f, a component's dynamic power consumption is denoted as P_d. P_d of CPU is given by $S_3 \times f^3$, where S_3 is constant.

Besides the CPU, *other* system components also consume energy. P_d of those that must operate at a fixed voltage (e.g., main memory) is given by $S_1 \times f$, while P_d of those that consume constant power with respect to the frequency (e.g., display devices) can be represented as a constant S_0. In practice, the quadratic term $S_2 \times f^2$ is also included to account for the appearance of variations in DC-DC regulator efficiency across the range of output power, CMOS leakage currents, and other second order effects [28].

Under a CPU frequency f (given in cycles per second), the execution time of one CPU cycle is $1/f$. Thus, summing the power consumption of all system components together, the system-level energy consumption per CPU cycle, denoted as $E^e(f)$, is obtained by $\frac{1}{f} \times (S_3 \times f^3 + S_2 \times f^2 + S_1 \times f + S_0)$. Therefore, we have the formula for $E^e(f)$ as:

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

We consider a single processor system that relies on battery power. The target variable voltage processor can be operated at m frequencies $\{f_1, \cdots, f_m | f_1 < \cdots < f_m\}$. 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 1.

B. Task and Resource Models

We consider a preemptive real-time system which consists of a set of tasks, denoted as $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$. 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 $\langle \overline{\underline{1}}, P_i \rangle$, 1 being both the upper and lower bound.

We refer to the j^{th} job (or invocation) of task 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. A job may be aborted for different reasons, which are described in the later sections.

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], [24], 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 are 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 paper) 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. We allow such precedences to be programmed as resource dependencies; the detailed descriptions can be found in [8] and [24].

C. Timeliness Model

A task's time constraint is specified using a TUF. Following [17], 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 [17], [24]. 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).

We use $U_i(\cdot)$ to denote task T_i's TUF. $U_i(\cdot)$ has the same shape as the TUF of each job of task T_i, i.e., $U_{i,j}(\cdot)$. Without being task specific, U_{J_k} means the TUF of a job J_k; completion of J_k at a time t will yield a utility $U_{J_k}(t)$. In this paper, we restrict our focus to *non-increasing*, unimodal TUFs i.e., those TUFs for which utility never increases as time advances. Figures 1(a), 1(b), and 1(c) show examples.

Each TUF $U_{i,j}, i \in \{1, \cdots, n\}$ has an initial time $I_{i,j}$ and a termination time $X_{i,j}$. Initial and termination times are the earliest and the latest times for which the TUF is defined, respectively. We assume that $I_{i,j}$ is equal to the arrival time of $J_{i,j}$, and $X_{i,j} - I_{i,j}$ is equal to the sliding time window P_i of the task T_i.

If a job's termination time is reached and its execution has not been completed, an exception is raised. Normally, this exception will cause the job's abortion and execution of exception handlers [24].

D. Statistical Timeliness 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 a probability of at least ρ_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.

During some situations, it is possible that such statistical assurances cannot be provided. When that happens, the objective is to maximize the total utility per system-level energy consumption.

E. Activity Cycle Demands

Both UA scheduling and DVS depend 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]; (2) worst-case workload is usually a very conservative prediction of the actual workload [3], resulting in excessive resource capacity; 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. These reasons are appropriate for soft real-time activities in dynamic systems, even when the activities are mission-critical.

Let Y_i be the random variable representing T_i's cycle demand i.e., the number of processor cycles required by T_i. We assume that the mean and variance of Y_i, denoted as $E(Y_i)$ and $Var(Y_i)$ respectively, are finite and determined through either online or off-line profiling. Under a frequency f (given in cycles per second), the expected execution time of a task T_i is given by $e_i = \frac{E(Y_i)}{f}$.

III. The EBUA Algorithm

A. 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 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.

Definition 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 this 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 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], [43], [44]). 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. On the other hand, during under-loads, the algorithm delivers the timeliness assurances. Thus, to meet the scheduling objective, we need to solve the dual criterion problem—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 [8].

B. Task Critical Time and Demand

For non-increasing TUFs, satisfying a designated ν_i requires that task T_i's sojourn time is upper bounded by a "critical time" (D_i). D_i is calculated from $\nu_i = \frac{U_i(D_i)}{U_i^{max}}$. If there are more than one points on the time axis that correspond to $\nu_i \times U_i^{max}$, we choose the latest point. Note that $P_i = D_i$ for a downward step TUF whose utility drops to a zero value at time P_i. Figure 2 illustrates the calculation of D_i with a simple decreasing TUF.

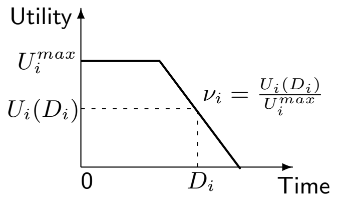


Fig. 2: Task T_i's Critical Time D_i

Knowing the mean and variance of task T_i's demand Y_i, by a one-tailed version of the Chebyshev's inequality, we have $Pr[Y_i < c_i] \geq \frac{(c_i - E(Y_i))^2}{Var(Y_i) + (c_i - E(Y_i))^2}$, when $c_i \geq E(Y_i)$. To satisfy the requirement ρ_i, we let ρ_i equal the right half of the inequality, 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., $Pr[Y_i < c_i] \geq \rho_i$.

C. Energy-Bounded UA Scheduling

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 *<u>U</u>tility and <u>E</u>nergy <u>R</u>atio* (UER), which is defined to integrate timeliness and energy consumption.

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 1. Equation 1 indicates that there is an optimal value for clock frequency to maximize T_i's LoUER.

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, the aggregate CPU demand of the task set $\mathbf{T}$ is defined as $CPU_{dmd} = \sum_{i=1}^{n} C_i/P_i$, and the system load is defined as $Load = \frac{1}{f_m}\sum_{i=1}^{n}\frac{C_i}{D_i}$. Note that we do not denote CPU_{dmd} as $\sum_{i=1}^{n} C_i/D_i$, and the reason is elaborated in Section III-G.

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. 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.
- 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 J_k. Thus, if $T_i = T(J_k)$, then $J_k.D = D_i^a$.
- 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 $x \leq f_i$.
- `offlineComputing()` runs at $t = 0$. It computes c_i and D_i as described in Section III-B, and determines its optimal frequency $f_{T_i}^o \in \{f_1, \cdots, f_m\}$, which maximizes T_i's LoUER. Each task initially runs at the speed $f_{T_i}^{ini} =$ `max(`$f_{T_i}^o$`, selectFreq(`CPU_{dmd}`))`.
- `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 denoting schedule σ's feasibility. For 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 $\mathbf{T}$.

Algorithm 1: EBUA: High Level Description

```
1:  input    : $\mathbf{T} = \{T_1, \cdots, T_n\}$, $\mathcal{J}_r = \{J_1, \cdots, J_{n'}\}$, $E_{rem}$
2:  output   : selected job $J_{exe}$ and frequency $f_{exe}$
3:  offlineComputing ($\mathbf{T}$);
4:  Initialization: $t := t_{cur}$, $\sigma := \emptyset$, 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:  foreach $J_k \in \mathcal{J}_r$ do
10:     if feasible($J_k$)=false then
11:         $c^r_{T(J_k)} := 0$;
12:         abort($J_k$);
13:     else
14:         $J_k.Dep$ := buildDep($J_k$);
15: foreach $J_k \in \mathcal{J}_r$ do
16:     $J_k.UER$:=calculateUER($J_k$, $t$);
17: $\sigma_{tmp}$ :=sortByUER($\mathcal{J}_r$);
18: foreach $J_k \in \sigma_{tmp}$ from head to tail do
19:     if $J_k.UER > 0$ then
20:         $\sigma$ := insertByECF($\sigma$, $J_k$);
21:     else
22:         break;
23: updateSel($\mathbf{T}$);
24: $J_{exe}$:=headOf($\sigma$);
25: $f_{exe}$:=decideFreq($\mathbf{T}$, $J_{exe}$, $t$);
26: return $J_{exe}$ and $f_{exe}$;
```

A high level description of EBUA is shown in Algorithm 1. We include the procedure `offlineComputing()` in line 3, but this sub-routine is only executed at $t = 0$. When EBUA is invoked at time t_{cur}, it first updates each task's remaining cycle (line 5–8). The algorithm then checks the feasibility of the jobs. If a job is infeasible, then it can be safely aborted (line 10–12). Otherwise, EBUA builds the dependency list for the job (line 14).

The UER of each job is computed by `calculateUER()`, and the jobs are then sorted by their UERs (line 15-17). In each step of the `for` loop from line 18 to 22, the job with the largest UER and its dependencies are inserted into σ, if it can produce a positive UER. The output 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.

At line 23, 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 17, 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 24 to 26, 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} at the head of 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.

D. Resource and Deadlock Handling

Before EBUA can compute job partial schedules, the dependency chain of each job must be determined, as shown in Algorithm 2.

Algorithm 2 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 2 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 2: `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) $\neq \emptyset \bigwedge$ owner(reqRes(Prev)) $\neq \emptyset$ do
       /* add new owner at the head of the list */
5:     $J_k.Dep :=$ owner(reqRes(Prev)) $\cdot J_k.Dep$;
6:     $Prev :=$ owner(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 on which we focus. 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 on which we focus (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: 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 $J_l \neq \emptyset$ do
5:      $J_l.LoUER := U_{J_l}(t_{cur} + \frac{J_l.c}{f_m})$  $(J_l.c \times E^e(f_m))$;
6:      if $J_l = J_k$ then
7:          $Deadlock$ := true;
8:          break;
9:      else
10:         $J_l$ := owner(reqRes($J_l$));
    /* deadlock resolution if any */
11: if $Deadlock$ = true then
12:     abort(The job $J_m$ with the minimal LoUER in the cycle);
```

The deadlock detection and resolution algorithm (Algorithm 3) 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.

E. Manipulating Partial Schedules

The `calculateUER()` algorithm (Algorithm 4) 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: `calculateUER()`

```
1: input    : $J_k$, $t_{cur}$;
2: output   : $J_k.UER$;
3: Initialization : $C_c := 0$, $E := 0$, $U := 0$;
4: foreach $J_l \in J_k.Dep$, 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: $J_k.UER := U \quad E$;
9: 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 $J_k.Dep$.

The details of `insertByECF()` in line 20 of Algorithm 1 are shown in Algorithm 5. `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,

Algorithm 5: `insertByECF()`

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

must precede it in the schedule. The index values of the dependencies can be changed with `insert()` in line 13 of Algorithm 5.

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 5). 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 5). 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 14–15). We will explain how to monitor the dynamic energy consumption in Section III-F.

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 III-B. 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()`.

F. 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 14 of Algorithm 5.

`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 II-A. 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*.

The above equation means that, at a future time D_{tent}, the energy that has been depleted, $E_{bnd} - E_{rem}$, plus the predicted energy consumption of executing σ_{tent} i.e., $E^e(t_{cur}, D_{tent})$, will not exceed the prorated energy budget, $\frac{D_{tent}}{MT} \times E_{bnd}$. Therefore, EBUA monitors and controls the energy consumption online through this equation.

G. 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" [5] to analyze the feasibility of tasks with stochastic parameters and UAM arrival model.

Theorem *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 $\frac{C_i}{D_i}$ [39]. 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 [5], [6]. 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:

$$f \geq \sum_{i=1}^{n} \frac{C_i}{L}\left(\frac{L}{P_i} + \frac{P_i - D_i}{P_i}\right) = \sum_{i=1}^{n} \frac{C_i}{P_i} + \sum_{i=1}^{n}\left(\frac{P_i - D_i}{L} \times \frac{C_i}{P_i}\right), \forall L > 0. \tag{2}$$

We study the part $\sum_{i=1}^{n}\left(\frac{P_i-D_i}{L} \times \frac{C_i}{P_i}\right)$ in Equation 2. 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 2, $\sum_{i=1}^{n}\left(\frac{P_i-D_i}{L} \times \frac{C_i}{P_i}\right)$ 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 validated the efficiency of the definition of CPU_{dmd} through experimental comparison. We observed in 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, `decideFreq()`, in Algorithm 6.

In line 3–5 of Algorithm 6, 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

Algorithm 6: `decideFreq()`

```
1:  input: $\mathbf{T}$, $J_{exe}$, $t_{cur}$; output: $f_{exe}$ ;
2:  $CPU_{dmd} := 0$;
3:  foreach $T_i \in \mathbf{T}$ do
4:      if $SEL_i = selected$ then
5:          $CPU_{dmd} := CPU_{dmd} + C_i/P_i$;
6:  $s := 0$;
7:  for $i \leftarrow 1$ to $n$, $T_i \in \{T_1, \cdots, T_n\ D^a_1 \geq \cdots \geq D^a_n\}$ do
8:      if $SEL_i = skipped$ then
9:          continue;
        /* reverse EDF order of tasks */
10:     $CPU_{dmd} := CPU_{dmd} - C_i/D_i$;
11:     $x :=$max$(0,\ C^r_i - (f_m - CPU_{dmd}) \times (D^a_i - D^a_n))$;
12:     $CPU_{dmd} := \begin{cases} f_m, & \text{if } D^a_i - D^a_n = 0 \\ CPU_{dmd} + \frac{C^r_i - x}{D^a_i - D^a_n}, & \text{otherwise} \end{cases}$ ;
13:     $s := s + x$;
14: $f :=$min$(f_m,\ s/(D^a_n - t_{cur}))$;
15: $f_{exe}$:=selectFreq $(f)$;
16: $f_{exe}$:=max$(f_{exe},\ f^o_{T(J_{exe})})$;
```

as $C^r_i = \texttt{min}((a'_i - 1)c_i + c^r_i, (a_i - 1)c_i + c^r_i)$. 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^r_i is updated from line 5–8 of Algorithm 1.

In line 6–14, EBUA considers the interval until the next task critical time and attempts to "push" as much work as possible beyond the critical time. Similar to LaEDF [31], the algorithm considers the tasks in the latest-critical-time-first order in line 7. But since there may be more than one job of T_i in P_i, D^a_i is set to be the earliest invocation's absolute critical time. LaEDF [31] updates each task's D^a_i 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. s reflects the minimum number of cycles that must be executed by D^a_n in order for the selected tasks to meet their critical times (line 13).

Thus, `decideFreq()` capitalizes on early task completion by deferring work for future tasks in favor of scaling the current task's frequency. Also, during overloads, the required frequency may be higher than f_m, and `selectFreq()` would fail to return a value. In line 14, we solve this by setting the upper limit of the required frequency to be the highest frequency f_m. Finally, f_{exe} is compared with $f^o_{T(J_{exe})}$. The higher frequency is selected to preserve the timeliness assurances. Note that we cannot decrease f_{exe}, because it reflects the minimum frequency required for the selected tasks to meet their critical times. But if f_{exe} is less than $f^o_{T(J_{exe})}$, at line 16, we heuristically increase it to

$f^o_{T(J_{exe})}$, so as to maximize the energy efficiency, since $f^o_{T(J_{exe})}$ maximizes LoUER of $T(J_{exe})$.

H. Computational Complexity

To analyze the complexity of EBUA (Algorithm 1), 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 9 to 14 requires $O(n^2)$ time. Also, the `for`-loop containing `calculateUER()` (line 15–16) can be repeated $O\left(n^2\right)$ times in the worst case. The complexity of procedure `sortByUER()` is $O(n \log n)$.

Complexity of the `for`-loop body starting from line 18 is dominated by `insertByECF()` (Algorithm 5). Its complexity is dominated by the `for`-loop (line 7–13, Algorithm 5), 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 $2 \times O(n^2) + O(n \log n) + n \times O(n \log n) = O(n^2 \log n)$.

Our implementation experience with EBUA [38]—this is not discussed here due to space limitations—revealed that the algorithm's overhead is typically in the magnitude of a few hundred microseconds. As discussed in Section I, the real-time systems that we consider in this paper 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 scheduling overhead than many traditional energy-efficient real-time scheduling algorithms such as LaEDF [31], which has a complexity of $O(n)$, the higher asymptotic cost of the algorithm and the consequent higher overhead are justified for application systems with longer execution time magnitudes, such as those on which we focus in this paper.

IV. Algorithm Properties

A. Non-Timeliness Properties

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

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, and we have Theorem 2.

Theorem *2:* EBUA ensures deadlock-freedom.

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.

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 Consider job J_k and its dependent J_l—J_l must be executed before J_k. 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. Thus, `insertByECF()` seeks to maintain an output queue ordered by jobs' critical times, while respecting resource dependencies. $\square$

From Lemma 1, we can derive another property of EBUA, as described in Theorem 3.

Theorem *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.

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 *1:* EBUA satisfies mutual exclusion constraints in resource operations.

B. Timeliness Properties

We first establish EBUA's assurance on energy-bounded systems. As described in Section III-F, EBUA's dynamic monitoring of energy consumption assures that, at any scheduling event between $[0, MT]$, the energy consumption of selected jobs never exceeds the allowed portion of E_{bnd}. Thus, we have the following theorem.

Theorem *4:* EBUA assures that the consumed energy during a mission interval $[0, MT]$ never exceeds the energy bound E_{bnd}.

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 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 EBUA.

***Theorem** 5:* Under EBUA for a non-energy-bounded system, a job J_k can be blocked for at most the duration of $\texttt{min}(n, m)$ 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 5, 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 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 $\langle \underline{1}, P_i \rangle$ and step TUFs; and (3) absence of CPU overloads (under Liu and Layland's condition [25]), we can establish EBUA's timeliness properties.

***Theorem** 6:* Under conditions (1), (2), and (3), a schedule produced by EDF [16] is also produced by EBUA, yielding equal total utilities. This is a critial time ordered schedule.

Proof We prove this by examining Algorithms 1 and 5. For a job J without dependencies, $J.Dep$ only contains J itself. For periodic tasks with step TUFs during non-overload situations, σ from line 20 of Algorithm 1 is critical time ordered. The step TUF critical time that we consider is analogous to a deadline in [16]. As proved in [16], [25], 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 EBUA's timeliness behavior during under-loads can be deduced from EDF's optimality [16].

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

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

***Theorem** 7:* Under conditions (1), (2), and (3), EBUA meets the statistical performance requirements.

The proofs can be found in [44]. In Theorem 8, we also derive the above theorems' counterparts for non-step and non-increasing TUFs, with which critical times are less than termination times.

***Theorem** 8:* In a non-energy-bounded system, for a set of independent periodic tasks, where each task has a single computational thread with a non-increasing TUF, the task set is schedulable and can meet all statistical performance requirements under the condition of Baruah, Rosier, and Howell [5].

Proof The proof is similar to validating a sufficient condition for EDF schedulability of a task set, where task deadlines are less than task periods. The proof can be found in [5]. □

V. Experimental Results

A. Experimental Settings

To evaluate the energy efficiency of EBUA, we perform simulation experiments to compare EBUA with other energy-bounded algorithms, i.e., OFC [2] and REW-Pack [32]. 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.

Our simulator is written with the simulation tool OMNET++ [35], which provides a discrete event simulation environment. We simulate the AMD k6 processor with `PowerNow!` mechanism [1]. The CPU can operate at seven different frequencies, $\{360, 550, 640, 730, 820, 910, 1000 \text{ MHz}\}$.

We select task sets with 10 to 50 tasks in three applications for our study. Their parameters are summarized in Table I. Within each range, the time window P is uniformly distributed. The synthesized task sets simulate the varied mix of short and long time windows. For each task cycle demand Y_i, we keep $Var(Y_i) \approx E(Y_i)$, and generate normally-distributed demands. Finally, according to the calculation of c_i in Section III-B, 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. The U^{max} of the TUFs in A_1, A_2, and A_3 are uniformly generated in the range $[50, 70]$, $[300, 400]$, and $[1, 10]$, respectively.

This type of method to generate real-time tasks has been used previously in the development and evaluation of the real-time embedded micro-kernel in [31] and the DVS approach in [20]. By means of

TABLE I: Task Settings

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

TABLE II: Energy 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	$0.5f_m^3$

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.

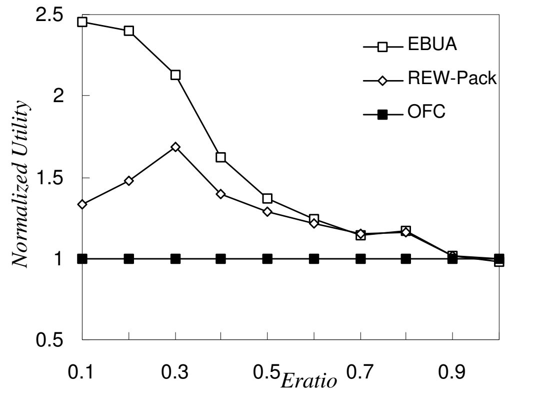
The energy consumption per cycle at a particular frequency is calculated using Equation 1. In practice, the S_3, S_2, S_1, and S_0 terms depend on the power management state of the system and its subsystems [28], [36], [43]. We test three energy settings similar to those in [43], as shown in Table II. These experimental settings are similar to those in Martin's PhD dissertation [28], but with de-normalized terms. 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 conventional energy model, which only considers the CPU's energy consumption.

B. Performance with Step TUFs

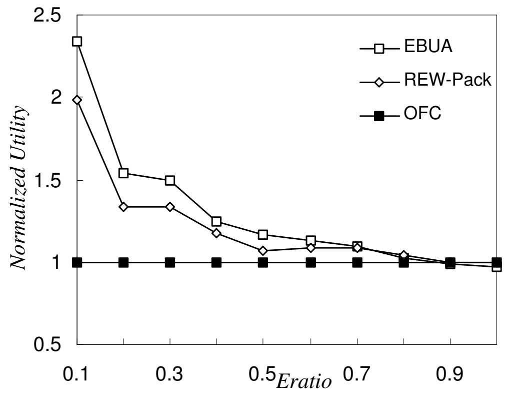
We first evaluate the performance with step TUFs, so that EBUA can be compared with the other strategies. We set $\{\nu_i = 1, \rho_i = 0.96\}$, and apply different schemes on independent periodic task sets under different energy settings. We vary *Eratio*, which is defined as the ratio of E_{bnd} to 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 to be MT in such cases.

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.

Figure 3 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 3(a) and Figure 3(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 [16] in such cases.



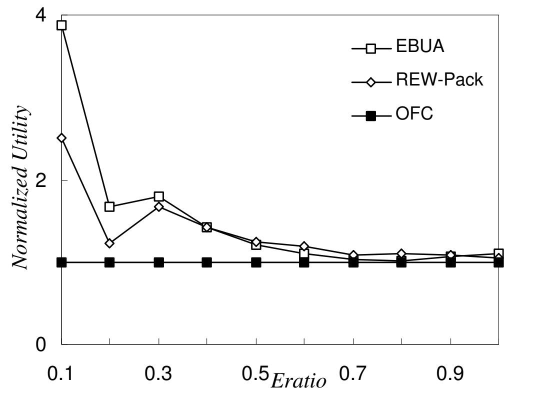
(a) $Load = 0.7$, E_1



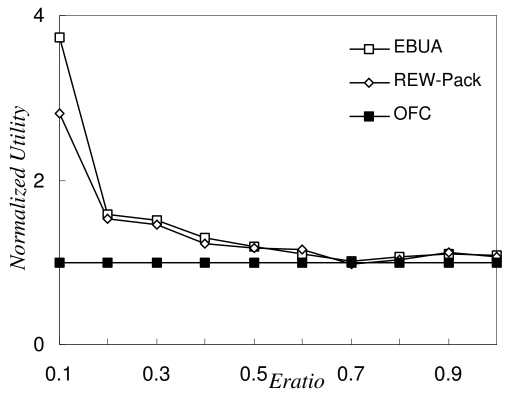
(b) $Load = 0.7$, E_3

Fig. 3: Normalized Utility vs. $Eratio$ during under-loads under E_1 and E_3

Figure 4(a) and Figure 4(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 4 bear the same trends as those in Figure 3. But when $Eratio = 100\%$, the end points of plots in Figure 4 do not coincide with each other. 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

Fig. 4: Normalized Utility vs. $Eratio$ during overloads under E_1 and E_3

In Figure 3 and Figure 4, 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 5 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 5 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 5 demonstrates that EBUA accrues higher utility with fixed energy bound.

C. Performance with Non-Increasing TUFs and UAM

We then consider non-step and non-increasing TUFs, and UAM tasks with EBUA and EUA$*$ [39]. 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.

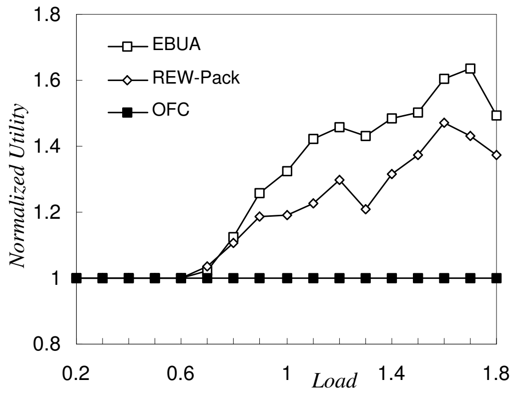


Fig. 5: Utility vs. $Load$ with Fixed E_{bnd}

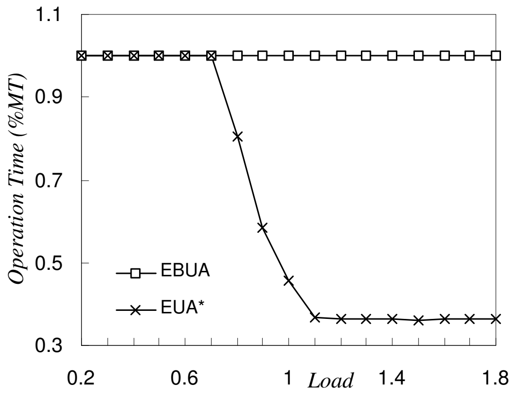


Fig. 6: Operation Time vs. $Load$

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 shows operation times normalized to the results of EBUA.

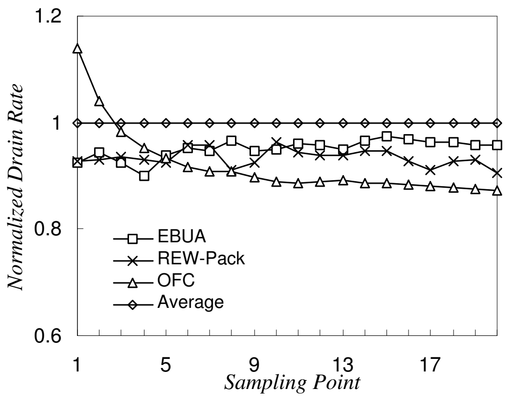
We observe that, the operation time of EBUA is always the whole mission time MT, since EBUA dynamically monitors energy consumption and keeps system functional during $[0, MT]$. When $Load \leq$

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 $Load \geq 1.0$.

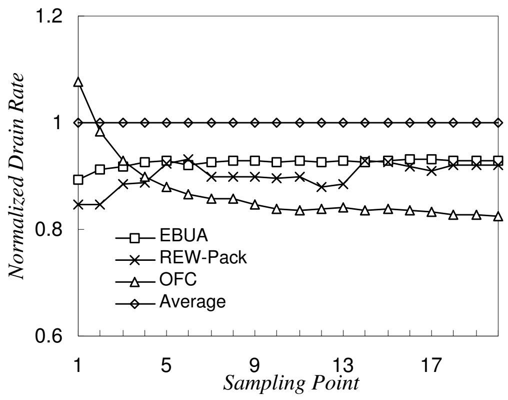
D. Comparison of Dynamic Energy Drain Rates

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 V-B, 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 7. Note that these figures only represent our qualitative study.



(a) $Load = 0.7$, E_1



(b) $Load = 0.7$, E_2

Fig. 7: Normalized Energy Drain Rates with $E_{bnd} = 0.5 \times E_{rqd}\,(Load = 0.7)$ under E_1 and E_2

We observe from the plots in both Figure 7(a) and Figure 7(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 energy as is available for timeliness.

VI. Conclusions

Many emerging battery-powered, dynamic, embedded real-time systems are subject to energy bounds, embodied by a battery that has a finite lifetime. Further, they operate in environments with dynamically uncertain properties, including resource overloads that occur due to context-dependent, activity execution times and arbitrary activity arrival patterns. In this paper, we consider optimization of timeliness performance and energy consumption in such systems, which must remain functional during an operation/mission with a bounded energy budget. We present a UA real-time scheduling algorithm called EBUA that considers activities subject to TUF time constraints, UAM arrival model, statistical timeliness requirements, and bounds on system-level energy consumption.

EBUA considers utility maximization under energy bounds, and dynamically skips less important jobs for execution to achieve the performance objectives of timeliness and energy, while assuring that the system remains functional until the end of the mission. EBUA applies DVS to reduce system-level energy consumption to obtain additional excess energy for selecting new jobs in a dynamic fashion. We analytically establish several properties of the algorithm including satisfaction of energy consumption bounds and timeliness bounds. Our simulation experiments confirm EBUA's (analytical) assurances on energy consumption and timeliness performance, and improvement and superiority in timeliness and system-level energy efficiency over past algorithms.

EBUA has a higher overhead than many traditional energy-efficient real-time scheduling algorithms. This high cost is justified for application systems with longer execution time magnitudes such as those on which we focus in this paper. For systems with smaller execution time magnitudes, optimization of the algorithm for reduced overhead such as using strategies described in [23] would be necessary.

Several aspects of the work are directions for further research. Examples include considering more general task arrival models (than UAM), and reward functions for tasks (besides TUFs), where tasks accrue reward as a function of their execution cycles.

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. Co-author E. Douglas Jensen's contributions to this work were sponsored by the MITRE Corp. Technology Program.

References

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

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

[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] 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*, 2(4):301–324, November 1990.

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

[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 *Lecture Notes in Computer Science*, 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: June 22, 2006).

[9] 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.

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

[13] 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.

[14] 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.

[15] 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*, pages 360–369, 1998.

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

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

[18] 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.

[19] 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.

[20] 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.

[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. L. Lann. Proof-Based System Engineering and Embedded Systems. In G. Rozenberg and F. Vaandrager, editors, *Proceedings of the European School on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 208–248. Springer-Verlag, October 1998. `http://www-rocq.inria.fr/novaltis/publications/PBSE%20&%20EmS%201998.pdf`.

[23] P. Li and B. Ravindran. Fast Real-Time Scheduling Algorithms. *IEEE Transactions on Computers*, 53(8):1159–1175, September 2004.

[24] P. Li, H. Wu, B. Ravindran, and E. D. Jensen. A Utility Accrual Scheduling Algorithm for Real-Time Activities With Mutual Exclusion Resource Constraints. *IEEE Transactions on Computers*, 55(4):454–469, April 2006.

[25] 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.

[26] 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: June 22, 2006).

[27] 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.

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

[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] 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.

[31] 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.

[32] C. Rusu, R. Melhem, and D. Mosse. Maximizing the System Value while Satisfying Time and Energy Constraints. *IBM Journal of Research and Development*, 47(5/6):689–702, September/November 2003.

[33] 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.

[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] A. Varga. OMNeT++ Discrete Event Simulation System. `http://www.omnetpp.org/` (last accessed: June 22, 2006).

[36] 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.

[37] 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.

[38] H. Wu. *Energy-Efficient, Utility Accrual Real-Time Scheduling.* PhD thesis, Virginia Tech, 2005. `http://scholar.lib.vt.edu/theses/available/etd-08242005-145355/` (last accessed: August 22, 2006).

[39] 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)*, pages 474–479, March 2005.

[40] H. Wu, B. Ravindran, and E. D. Jensen. On Bounding Energy Consumption in Dynamic, Embedded Real-Time Systems. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, April 2006.

[41] 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)*, pages 80–98, August 2004.

[42] 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.

[43] 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 4th ACM International Conference on Embedded Software (EMSOFT)*, pages 64–73, September 2004.

[44] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Energy-Efficient, Utility Accrual Scheduling under Resource Constraints for Mobile Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 5(3), August 2006.

[45] 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.