On Scheduling Soft Real-Time Tasks with Lock-Free Synchronization for Embedded Devices

Shouwen Lai*, Binoy Ravindran*, Hyeonjoong Cho†

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

†ETRI, RFID/USN Research Group
161 Gajeong-dong, Daejeon 305-350, Korea
raycho@etri.re.kr

ABSTRACT

In this paper, we consider minimizing the system-level energy consumption through dynamic voltage scaling for embedded devices, while a) allowing concurrent access to shared objects through lock-free synchronization b) meeting (m, k)-constraint, and c) completing as many high importance tasks as possible. We present a scheduling algorithm called *Lock-Free Utility accrual* Algorithm (or MK-LfUA) to meet these goals. At offline stage, we statistically determine task execution time, and set the optimal CPU speed that will minimize system-level energy consumption. At run-time, the algorithm dynamically adjusts the CPU speed to compensate for slack time, while taking into account the speed transition overhead. Our simulation studies on the Intel PXA271 processor model illustrate MK-LfUA's superiority over past work by 15-25%.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

Keywords

Lock-free, dynamic voltage scaling, real-time scheduling, (m, k)

1. INTRODUCTION

Embedded wireless sensor networks (WSNs) are increasingly being envisioned for a number of applications such as surveillance, target tracking, environment monitoring etc,. In target tracking applications [8], embedded sensor nodes deployed in a surveillance field periodically generate physical measurements of a target, and continuously report the measurements to a sink node or a local cluster head, which aggregates data to identify/classify the target. When there are multiple targets (intruders), concurrent and periodical aggregation tasks will run on the sink node or the local cluster head. Not all aggregation task instances must meet their deadlines due to the limited CPU and memory resources. Furthermore, to avoid loosing the target during any window of time, it is desirable that the *distribution* of task deadline-misses be bounded.

The (m, k) model [7] allows the specification of such soft real-time requirements. In this model, each periodic real-time task is associated with an (m, k) $(0 \leq m \leq k)$ constraint, which means that at least m out of k consecutive jobs of the task must meet their deadlines. If the (m, k) constraints cannot be met (e.g., due to transient overloads), then the deadlines of as many high importance tasks as possible must be met, where task importance is specified using the utility dimension of time/utility functions (TUFs) [9]. TUFs allow the specification of task importance that is decoupled from task urgency–e.g., the most important task may be the most urgent; the most important may be the least urgent.

We consider minimizing the *system*-level energy consumption of an embedded node, and not just the CPU's energy consumption. We use Martin's empirically-derived system-level energy consumption model [10] to do so. 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 complex relation between energy consumption and core voltage and frequency, to account for components that consume constant energy and for components that consume energy that is only scalable to frequency (i.e., voltage). In addition, most commercial embedded sensor nodes (e.g., Intel iMote) use DVS-enabled processors (e.g., PXA271).

Many embedded real-time softwares involve concurrent accesses to shared data objects, resulting in contention for those objects. For example, two aggregation tasks may operate on same local database table records to trace identical intruders. Thus, correct synchronization of task behaviors is necessary. Mechanisms that resolve such contention can be classified into: (1) lock-based (e.g., [13]) and (2) non-blocking schemes (e.g., [5],[1]). Lock-based schemes have several disadvantages such as reduced concurrency, potential for deadlocks, and need for a-priori knowledge of lock ceilings.

The alternative, non-blocking synchronization, includes lock-free and wait-free synchronization. These methods overcome the disadvantages of locks. For example, for the single-writer/multi-reader problem, wait-free protocols (e.g., [4]) typically use multiple buffers to avoid conflicts for the shared object. In contrast, lock-free protocols (e.g., [1, 5]) allow readers to concurrently read while the writer is writing, but

ACM SAC '2009

readers retry their read if read/write conflicts are detected. Since a reader's worst-case number of retries depends upon the worst-case number of times the reader is preempted by the writer, the additional execution-time overhead incurred for the retries is bounded by the number of preemptions.

Contributions. Given such a model, our goal is to minimize the system-level energy consumption while meeting the (m, k)-constraint as much as possible and allowing concurrent access to shared objects through lock-free synchronization. We present an algorithm called *Lock-Free Utility accrual* Algorithm (or MK-LfUA). We adopt statistical task execution time in this algorithm and derive the optimal frequency that will minimize the system's energy consumption with the (m, k)-constraint and lock-free shared data. We also show how to dynamically adjust the CPU speed when there are slack time and speed transition overheads. Our simulation studies on the PXA271 processor model (used in Intel iMote2 sensor nodes) and using parameters of WSN-target tracking applications illustrate MK-LfUA's superiority over past algorithms (that solve a subset of MK-LfUA's problem space) on energy consumption by 15-25%.

MK-LfUA can be applied to embedded applications, like data aggregation in wireless sensor networks. In a local cluster head or a sink node, there are usually concurrent aggregation tasks with soft real-time requirements and shared objects. MK-LfUA can schedule these tasks with energy-efficiency and with maximized accrued utility (or benefits).

Paper Structure. In Section 2, we outline our models and formulate the scheduling problem. Section 3 describes how to set the optimal CPU speed and the trade-off of lock free in offline stage. We present online scheduling algorithm in Section 4. In section 5, we ensemble both offline scheduling and online scheduling in MK-LfUA algorithm. We report our simulation results in Section 6. The paper concludes in Section 7.

2. MODELS AND OBJECTIVES

2.1 System and Task Model

The application is assumed to consists of a set of tasks $T = \{T_1, T_2, ..., T_n\}$. Each task contains an infinite sequence of periodically arriving instances called *jobs*. Each task T_i is characterized by the tuple $[D_i, U_i, P_i, \overline{C_i}, Var(C_i), (m_i, k_i, \rho_i)]$. D_i is the deadline of each job of T_i and U_i is the utility achieved when T_i meets its deadline D_i. The relative importance of the task with respect to other tasks can be modeled using this utility parameter. P_i is the task period. $P_i = D_i$.

DVS real-time scheduling is dependent on the prediction of task execution time demands. $\overline{C_i}$ and $Var(C_i)$ represent the mean and variance of T_i's execution time, respectively, at the maximum CPU speed. We estimate the statistical properties (e.g., mean, variance) of the demand rather than the worst-case demand (WCET) for three reasons: (1) many embedded real-time applications exhibit a large variation in their *actual* workload. 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, which results in resource over-supply and exacerbates the power consumption.

We assume that the mean and variance of task execution times are finite and determined through measurement-based (offline) profiling mechanisms. The tuple (m_i, k_i, ρ_i) $(0 < m_i < k_i)$ specifies the requirement that at least m_i job deadlines out of k_i consecutive jobs must be met with ρ_i probability.

Jobs may concurrently read or write shared data objects, causing the multi-writer/multi-reader (MWMR) problem [5].

2.2 Energy Consumption Model

We consider Martin's system-level energy consumption model to derive the energy consumption at a given CPU speed [10]. 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 a 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 potential variations in DC-DC regulator efficiency across the range of output power, CMOS leakage currents, and other second order effects [10].

Summing the power consumption of all components, the system-level energy consumption of a task T_i becomes $E_i = (c_i \cdot f_m/f)(S_3 \times f^3 + S_2 \times f^2 + S_1 \times f + S_0)$, where c_i denotes T_i's expected execution time at the maximum frequency f_m. The normalized CPU speed at frequency f_i is $S_i = \frac{f_i}{f_m}$. Thus, the energy consumption at a given CPU speed S_i is:

$$E(S_i) = c_i \cdot \left(C_3 \cdot S_i^2 + C_2 \cdot S_i + C_1 + \frac{C_0}{S_i} \right) \qquad (1)$$

where $C_3 = S_3 \cdot f_m^3$, $C_2 = S_2 \cdot f_m^2$, $C_1 = S_1 \cdot f_m$, and $C_0 = S_0$ are system-dependent parameters, and $S_{min} \leq S_i \leq S_{max} = 1.0$. We can see that $E(S_i)$ is a convex function. By Descartes' Rule of Signs, there is only one value, denoted S_{low} that minimizes $E(S_i)$ (in Figure 1(a), $S_{low} \approx 0.5$ for energy setting E_1).

2.3 Scheduling Objectives

Given these models, our scheduling objectives are to minimize the system-level energy consumption (through CPU speed scaling) and maximize the total utility that is accrued from task completions, under the constraints of (a) allowing concurrent shared object accesses through lock-free synchronization; (b) meeting the (m, k) constraints with at least the probability ρ_i; and (c) ensuring that the total CPU utilization due to all scheduled tasks is less than 1.0.

This problem is NP-hard because it subsumes the problem of scheduling to maximize total task accrued utility without (m, k) constraints, which has been shown to be NP-hard in [6]. Thus, MK-LfUA is a heuristic algorithm.

3. OFF-LINE SCHEDULING

3.1 Bounding Task Execution Time

For a random variable X with mean μ and finite non-zero variance σ^2, for any $\lambda > \sqrt{3/8}$, by Vysochanski-Petunin:

$$P(|X - \mu| \geq \lambda\sigma) \leq \frac{4}{9\lambda^2} \qquad (2)$$

Thus, we can construct 3σ limits to bound X with a probability 95% ($\lambda = 3$) and 4σ limits to bound X with a probability 99% ($\lambda = 4$).

We now probabilistically estimate T_i's actual execution time, denoted c_i, using the desired probability ρ_i for meeting (m_i, k_i). c_i is thus *bounded* as:

$$P[|c_i - \overline{C_i}| < \lambda \cdot Var(C_i)] \geq \rho_i \quad (3)$$

where $\lambda = \left\lceil \sqrt{\frac{4}{9 \cdot (1-\rho_i)}} \right\rceil$. For example, for $\rho_i = 95\%$ ($\lambda = 3$), c_i will be bounded within $[\overline{C_i} - 3Var(C_i), \overline{C_i} + 3Var(C_i)]$ with a probability of at least 95%.

Given the tuple (m_i, k_i, ρ_i) $(0 < m_i < k_i)$ for a task T_i, its estimated execution time demand is computed as $c_i = \overline{C_i} + \left\lceil \sqrt{\frac{4}{9 \cdot (1-\rho_i)}} \right\rceil \cdot Var(C_i)$. For example, if $\rho_i = 95\%$, $c_i = \overline{C_i} + 3Var(C_i)$. By doing this, we can satisfy the requirement regarding ρ_i probability .

For the requirement of (m_i, k_i) in the tuple, it is satisfied by the online scheduling algorithm in Section 4.

3.2 Optimal Offline CPU Speed

We first discuss how to set the optimal CPU speed, denoted S_{opt}, to minimize the system-level energy consumption in offline stage.

Since determining the sufficient schedulability condition of tasks with (m, k)-constraint is NP-hard [12], it is difficult to optimally decide the CPU speed to meet (m, k)-constraint. So we consider the offline CPU speed for scheduling all jobs in the system with lock-free synchronization.

Similar to [5, 1], we assume that all accesses to lock-free objects require the same time, denoted $\underline{s}$ time units.

Recall the definition of S_{low} in Section 2.2. Now,

PROPOSITION 1. *If $\sum_{j=1}^{n} \frac{(c_j+s)}{P_j \cdot S_{low}} \leq 1$, then all jobs are schedulable and $S_{opt} = S_{low}$.*

PROOF. From [1], the sufficient schedulability condition under EDF with lock-free is $\sum_{j=1}^{n} \frac{(c_j+s)}{P_j} \leq 1$ for maximal CPU speed (1.0). When CPU speed is S_{low}, the execution time for a job becomes T_j is $(c_j + s)/S_{low}$. Considering the convex feature of energy function, the proposition holds. □

THEOREM 1. *When $\sum_{j=1}^{n} \frac{c_j+s}{P_j} < 1$, all jobs are schedulable, and all tasks have the same optimal speed $S_{opt} = \sum_{i=1}^{n} \frac{c_i+s}{P_i}$ if $S_{opt} \geq S_{min}$.*

PROOF. Since all task power consumption functions from Section 2.2 are convex and identical, the optimal speeds to minimize system-level energy consumption for all tasks are same. When $\sum_{j=1}^{n} \frac{c_j+s}{P_j} < 1$, the total CPU Utilization ≤ 1.0 for maximum CPU Speed 1.0, so all tasks are schedulable. When we set $S_{opt} = \sum_{i=1}^{n} \frac{c_i+s}{P_i} \geq S_{min}$, the total CPU utilization becomes 1.0 and the system-level energy consumption is minimized due to convex energy consumption functions from Section 2.2. □

Note: if $\sum_{j=1}^{n} \frac{c_j+s}{P_j} < S_{min}$, then one needs to use S_{min}.

PROPOSITION 2. *If $\sum_{j=1}^{n} \frac{c_j+s}{P_j} > 1$, then all tasks need to be scheduled with a maximum CPU speed $S_{opt} = 1.0$.*

Thus, we determine the CPU speed to be set offline as either S_{min}, or $\sum_{j=1}^{n} \frac{c_j+s}{P_j}$, or the maximum speed 1.0 (per Proposition 2).

We define the function of obtaining the optimal CPU speed as *Get_OptimalSpeed*() in Algorithm 1.

Algorithm 1: `Get_OptimalSpeed()`

```
1: input: Task set T, access time s for lock-free object;
   output: S_opt;
2: U = Σ_{j=1}^{n} (C_j+s)/P_j;
3: for ∀ CPU Speed S_i do
5:   S_low = min{E(S_i)};
7: if U ≤ S_low then
8:   S_opt = S_low;
9: else if U < 1 then
10:  S_opt = U;
11: else if U > 1 then
13:  S_opt = 1.0;
14: return S_opt;
```

3.3 Tradeoff of Lock-Free

Lock-free is blocking-free but suffers from retries. Lock-based doesn't have retries but suffers from blocking, besides deadlocks and priority inversion problems. To understand the tradeoffs, we compare lock-free and lock-based object sharing by comparing task sojourn times, similar to [5]. A task's sojourn time is the time between the task's arrival and its completion, and thus includes the task's execution time, retry time, and interference.

Since a job's worst-case number of retries depends upon the worst-case number of times the job is preempted, the additional execution-time overhead incurred for the retries is bounded by the number of preemptions.

THEOREM 2. *(Lock-Free Retry Bound under (m, k)). With MK-LfUA, the total number of retries for mandatory job $\mathcal{J}_i$, is at most:*

$$\sum_{j=1, j\neq i}^{N} \left\lceil \left\lceil \frac{P_i}{P_j} \right\rceil \cdot \frac{m_j}{k_j} \right\rceil \quad (4)$$

PROOF. Retries for mandatory job $\mathcal{J}_i$ is bounded by the number of preemptions. With MK-LfUA algorithm (described in Section 5), the number of retries for mandatory job $\mathcal{J}_i$ in the interval $[t, t + P_i]$ (assume t is the release time of $\mathcal{J}_i$) is bounded by the number of other mandatory jobs with higher execution eligibility than $\mathcal{J}_i$ in this time interval. The maximum number of releases of task $T_j (j \neq i)$ within $[t, t + P_i]$ is $\lceil P_i/P_j \rceil$, and the mandatory jobs of T_j is $\left\lceil \left\lceil \frac{P_i}{P_j} \right\rceil \cdot \frac{m_j}{k_j} \right\rceil$ in this interval. Thus, the total number of retries is bounded by $\sum_{j=1, j\neq i}^{N} \left\lceil \left\lceil \frac{P_i}{P_j} \right\rceil \cdot \frac{m_j}{k_j} \right\rceil$. □

Let u_i denote the computation time for accessing non-shared objects and I denote the worst-case interference time for a job. The worst-case sojourn time of a job under lock-based is $u_i + I + r \cdot m_i + r \cdot min(m_i; n_i)$ [14], where m_i is the number of shared objects accessed by job $\mathcal{J}_i$ and n_i is the number of jobs that could block $\mathcal{J}_i$. On the other hand, the worst-case sojourn time of a job under lock-free is $u_i + I + s \cdot m_i + s \cdot f$, where $f = \sum_{j=1, j\neq i}^{N} \left\lceil \left\lceil \frac{P_i}{P_j} \right\rceil \cdot \frac{m_j}{k_j} \right\rceil$.

THEOREM 3. *Let jobs be scheduled by MK-LfUA. Now, if*

$$\begin{cases} \frac{s}{r} < \frac{2 \cdot m_i}{f + m_i}, m_i \leq n_i \\ \frac{s}{r} < \frac{m_i + f}{m_i + n_i}, m_i > n_i \end{cases}$$

then the sojourn time of job J_i with lock-free objects is shorter than that with lock-based objects.

PROOF. Let A denote $r \cdot m_i + r \cdot min(m_i, n_i)$ and B denote $s \cdot m_i + s \cdot f$. We now derive the condition under which lock-free sharing yields shorter sojourn times than lock-based, which means $A > B$. There are two cases:
Case 1: When $m_i \leq n_i$,$A = 2 \cdot r \cdot m_i$. So when $2 \cdot r \cdot m_i > s \cdot m_i + s \cdot f$, we can get:

$$\frac{s}{r} < \frac{2 \cdot m_i}{f + m_i}$$

Case 2: When $m_i > n_i$, A becomes $r \cdot (m_i + n_i)$. So when $r \cdot (m_i + n_i) > s \cdot m_i + s \cdot f$, we can get $\frac{s}{r} < \frac{m_i + f}{m_i + n_i}$. □

Theorem 3 identifies the condition for jobs to obtain a shorter sojourn time under lock-free. In [1], the authors show that s is typically much smaller than r in comparison with various lock-free objects. For example, for simple lock-free objects such as buffers and stacks, s varies from $2ms$ to $10ms$ with hardware support, while $r > 100ms$ [1].

4. ON-LINE SCHEDULING

4.1 Partitioning Jobs to Meet (m, k) Constraints

To schedule tasks to meet (m, k)-constraints, we partition the jobs of a task as mandatory (i.e., those which must meet their deadlines) and optional (i.e., those which may miss their deadlines). To decide which jobs are mandatory and which are optional, we use the *k-sequence* concept in [7] to record the recent execution history of a task T_i. The *k-sequence* is a word of k bits ordered from the most recent to the oldest job of the task, in which each bit records whether the job deadline was missed (bit = 0) or met (bit =1). Each new job arrival causes a shift of all the bits toward the left.

For the j^{th} job of T_i, we denote the previous *k-sequence* as $s_j = (\delta^i_{j-k_i+1}, ..., \delta^i_j)$, and denote the position (from right) of the n^{th} deadline-meet (or 1) in s_j as $l^i_j(n, s)$. Then, the *failure distance* (i.e., the distance to task failure state) [7], denoted FD^i_j is given by:

$$FD^i_j = k_i - l^i_j(m_i, s_j) + 1. \quad (5)$$

For a newly arriving job, it is a *mandatory* job if $FD^i_j = 1$. The *optional* jobs have $FD^i_j > 1$. For example, let the (m, k) constraint of a task be $(3, 5)$ and the current *k-sequence* is 11011, then $FD^i_j = 2$, and the next job is an optional job. If the current *k-sequence* is 11010, then $FD^i_j = 1$, and the next job is a mandatory job.

The procedure of partitioning jobs into mandatory and optional at run-time is depicted in Algorithm 2. The time complexity for Algorithm 2 is $\mathcal{O}(n)$.

Algorithm 2: `Job_Partition(`σ`)`

```
1:  input: Current online Job set J_r = {J_1, ..., J_n'};
    output: Exe_Queue ;
2:  for ∀J_k ∈ J_r do
3:      T_i = Task from which J_r comes;
4:      M = m_i;  /*m_i is the (m,k) parameter of T_i*/
5:      K = k_i;  /*k_i is the (m,k) parameter of T_i*/
7:      pos = Find_Right(k-sequence for T_i, M, 1);
9:      FD(J_k) = M − pos + 1;
10:     index = FD(J_k);
11:     EnQueue(Queue_FD[index],J_k);
13: if Queue_FD[1] ≠ ∅ then
14:     Exe_Queue = Queue_FD[1];
15: else
16:     /*put optional jobs into the queue for execution*/
17:     for j = 2 to max(k_i − m_i) do
18:         if Queue_FD[j] ≠ ∅ then
19:             Exe_Queue = Queue_FD[j];
21:             break;
22: return Exe_Queue;
```

4.2 Job Scheduling to Maximize Total Utility

Having partitioned jobs into mandatory and optional to meet (m, k) constraints, the next questions that we need to answer include a) which jobs must be selected for execution (note that due to potential overloads, not all jobs can be feasibly executed) and b) the order in which the selected jobs must be dispatched for execution, to maximize the total accrued utility. This is done as follows.

To determine which jobs must be selected for execution, the algorithm follows a greedy approach and examines all jobs in the non-increasing order of their "return on investment." We measure a job's return on investment as the amount of utility that can be obtained per unit execution time i.e., its utility density (UD). A job i's UD is given by $UD_i = U_i/c^r_i$, where c^r_i is the remaining job execution time.

Algorithm 3: `Job_Scheduling(`*queue*`)`

```
2:  input : Ready Jobs Queue: queue
4:  output: the selected job J_exe
6:  σ_tmp := UD_Sorting(queue);
7:  for ∀J_k ∈ σ_tmp do
9:      if !Feasibility_Check(J_k,σ) then
11:         abort(J_k);
12:     else
14:         σ:=EDF_Insert(σ,J_k);
15: J_exe := headOf(σ);
```

Each examined job is inserted into a deadline-ordered schedule, as deadline-ordering is optimal with respect to meeting all deadlines [14]. The schedule is then tested for its feasibility, and if infeasible, the inserted job is rejected. This process is repeated until all jobs are examined, while preserving the invariant of schedule feasibility. The job at the head of the schedule is then dispatched for execution.

We describe the procedure of job scheduling to maximize total utility in Algorithm 3. In line 6, we sort all jobs in decreasing order of utility density (UD). In line 11, abortion is done only when termination time is missed. The sub-procedure of inserting jobs into a deadline-ordered schedule is presented in Algorithm 4.

The time complexity for Algorithm 3 is $\mathcal{O}(n^2)$ since either $Feasibility\_Check()$ and $EDF\_Insert()$ has a time complexity of $\mathcal{O}(n)$.

Note that jobs synchronize through lock-free synchronization. Thus, no dependencies arise between jobs and no deadlocks occur, saving MK-LfUA from costly dependency-chain and deadlock resolution computations, unlike in [6, 13].

This process will ensure that the output schedule is a feasible schedule. Furthermore, during underloads, no jobs will be rejected, and this schedule will be EDF. In addition, by

Algorithm 4: `EDF_Insert(σ, J_k))`

input : J_k and an ordered job list σ
output: the updated list σ
if $J_k \notin \sigma$ **then**
  $\sigma_{tent} = \sigma$;
  Insert(J_k, σ_{tent}, $J_k.deadline$);
  if *feasible*(σ_{tent}) **then**
    $\sigma := \sigma_{tent}$;
return σ;

virtue of the greedy UD-order by which jobs are examined, the output schedule will likely yield a high total utility.

4.3 CPU Speed Adjustment

The actual task execution time may differ from the expected execution time demand. Thus, the CPU speed must be dynamically adjusted. We need to dynamically reclaim the slack time if a job is completed earlier than expected, and increase the next tasks' speed to accommodate all jobs if a job is completed later than expected.

Since there is a CPU speed transition overhead, we must take into account the potential payoff from changing the speed to determine CPU speed. From [3], the energy and time transition overheads are $E_{TRAN} = (1-\eta)\cdot c\cdot|V_{DD_2}^2 - V_{DD_1}^2|$ and $t_{TRAN} \approx \frac{2\cdot C}{I_{MAX}}|V_{DD_2} - V_{DD_1}|$, respectively, where η, C, and I_{MAX} are system parameters. Since $f \propto V_{DD}$ and $S_i = f_i/f_m$, the transition energy and time overhead from CPU speed S_2 to S_1 are:

$$E_{TRAN} \approx R_1 \cdot |S_2^2 - S_1^2| \tag{6}$$

$$t_{TRAN} \approx R_2 \cdot |S_2 - S_1| \tag{7}$$

where R_1 and R_2 are constants related with the system.

Suppose the *slack time* for a job J_i is $\Delta t_i = e_{est} - e_{act}$, where e_{est} is the estimated execution time for J_i and e_{act} is the actual execution time for J_i, and the *aggregate slack time* for the previous k executed jobs is $\sum_1^k \Delta t_k$. Let the remaining execution time for the next job be c^r, the potential new CPU speed for reclaiming *aggregate slack time* be S_{adj}, and let the CPU has discrete speeds: $\{S_{min}, ..., S_{opt-1}, S_{opt}, ..., S_{max}\}$. Now,

PROPOSITION 3. *If the* aggregate slack time

$$\sum_1^k \Delta t_k > \left(\frac{1}{S_{opt-1}} - \frac{1}{S_{opt}}\right)\cdot[c^r + 2\cdot R_2\cdot(S_{opt} - S_{opt-1})],$$

then the adjusted speed can be set as:

$$S_{adj} = max\left\{S_{low}, \left\lceil \frac{c^r \cdot S_{opt}}{c^r + \left(\sum_1^k \Delta t_k - 2\cdot t_{TRAN}\right)\cdot S_{opt}} \right\rceil\right\};$$

otherwise, the CPU speed can be unchanged.

PROOF. When the *aggregate slack time* > 0, we should first check whether the slack time can be allocated to the next job, considering the discrete CPU speed and transition overheads. The sufficient condition for allocation is $\frac{c^r}{S_{adj}} < \frac{c^r}{S_{opt}} + \sum_1^k \Delta t_k - 2\cdot t_{TRAN}$. Now, we can derive that $c^r < \frac{S_{opt}\cdot S_{adj}}{S_{opt}-S_{adj}}\cdot\left(\sum_1^k \Delta t_k - 2\cdot t_{TRAN}\right) \le \frac{S_{opt}\cdot S_{(opt-1))}}{S_{opt}-S_{(opt-1)}}\cdot\left(\sum_1^k \Delta t_k - 2\cdot t_{TRAN}\right)$.

Thus, the threshold for CPU speed adjustment is $\sum_1^k \Delta t_k > \left(\frac{1}{S_{opt-1}} - \frac{1}{S_{opt}}\right)\cdot[c^r + 2\cdot R_2\cdot(S_{opt} - S_{opt-1})]$. Combining with the transition overhead model, we can obtain the adjusted CPU speeds. □

PROPOSITION 4. *If $\sum_1^k \Delta t_k < 0$, we need to increase the CPU speed for the next job. The new adjusted speed is:*

$$S_{adj} = argmin\left\{S_i | S_i > \frac{c^r\cdot S_{opt}}{c^r + (\Delta t - 2t_{TRAN})\cdot S_{opt}}\right\}$$

where $S_{opt} < S_i < S_{max}$.

PROOF. Similar to Proposition 3's proof, we can obtain $\frac{c^r}{S_{adj}} < \frac{c^r}{S_{opt}} + \sum_1^k \Delta t_k - 2\cdot t_{TRAN}$, considering the transition overhead. It indicates that $S_{adj} > \frac{c^r\cdot S_{opt}}{c^r + (\Delta t - 2t_{TRAN})\cdot S_{opt}}$. We conclude the result from the convex feature of $E(S_i)$. □

To dynamically adjust the CPU speed, MK-LfUA first calculates the *aggregate slack time* by adding up the slack time from previously executed jobs. After obtaining the aggregate slack time $\sum_1^k \Delta t_k$, the algorithm adjusts the CPU speed according to Propositions 3 and 4.

Algorithm 5: `Speed_Adjust(S_opt, △t, c^r)`

input: S_{opt}, $\triangle t$, c^r; **output**: S_{adj} ;
constant set = enum$\{S_{min}, ..., S_{opt-1}, S_{opt}, ..., 1.0\}$;
$Thresh = \left(\frac{1}{S_{opt-1}} - \frac{1}{S_{opt}}\right)\cdot[c^r + 2\cdot R_2\cdot(S_{opt} - S_{opt-1})]$;
if $\triangle t > Thresh$ **then**
  $S_{new} = \left\lceil \frac{c^r\cdot S_{opt}}{c^r + (\triangle t - 2\cdot t_{TRAN})\cdot S_{opt}} \right\rceil$;
  $S_{adj} = max\{S_{low}, S_{new}\}$;
  $\triangle t = 0$;
else if $0 < \triangle t < Thresh$ **then**
  $S_{adj} = S_{opt}$;
if $\triangle t < 0$ **then**
  $S_{adj} = argmin\left\lceil \frac{c^r\cdot S_{opt}}{c^r + (\triangle t - 2\cdot t_{TRAN})\cdot S_{opt}} \right\rceil$;
  $\triangle t = 0$;
return S_{adj};

Algorithm 5 shows dynamically adjusting the CPU speed.

5. ALGORITHM DESCRIPTION

We now provide a high-level procedural description of MK-LfUA in Algorithm 6, combining the offline scheduling and online scheduling. We define the following auxiliary functions:

- `Offline_Computing()` is invoked at time $t = 0$. For a task T_i, it computes the estimated execution time demand $c_i = \overline{C_i} + 3Var(C_i) + s$ (Section 3.1).
- `Get_OptimalSpeed()` obtains the optimal CPU speed for the task set, offline (Algorithm 1).
- `Job_Partition(σ)` partitions jobs into mandatory and optional at run-time, per their *failure distance* (Algorithm 2).
- `Job_Scheduling(queue)` schedules jobs in *queue* (Algorithm 3).
- `SlackTime_Calc()` calculates the *aggregate slack time* after executing previous jobs.
- `Speed_Adjust(S_opt, △t, c^r)` adjusts the optimal CPU speed for job J_{exe} according to *aggregate slack time* via *SlackTime_Calc()* (Algorithm 5).

We present the pseudocode for most of these functions in previous sections. We omit *Offline_Computing()* and *SlackTime_Calc()* as they are straightforward.

Algorithm 6: MK-LfUA: High Level Description

```
1: input : T = {T_1, ..., T_n}, J_r = {J_1, ..., J_n'}
2: output: selected job J_exe and CPU speed S_exe
3: Offline_Computing();
4: S_opt = Get_OptimalSpeed();
5: Initialization: t := t_cur, σ := ∅;
7: Initialization: static △t = 0;
8: switch triggering event do
9:     case task_release(T_i)        c_i^r = c_i;
10:    case task_completion(T_i)     c_i^r = 0;
11:    otherwise                     Update c_i^r;
12: EXE_Queue = Job_Partition(J_r);
13: J_exe = Job_Scheduling(EXE_Queue);
14: △t = SlackTime_Calc();
15: S_exe = Speed_Adjust(S_opt, △t, c_{J_exe}^r);
```

6. SIMULATION RESULTS

We extensively evaluated MK-LfUA's performance through simulation. The goal of our simulation study is to understand how MK-LfUA's timing and energy consumption properties compare with other algorithms that address subsets of MK-LfUA's problem space. We used the Intel PXA271 processor model, which supports five discrete frequencies {13, 104, 208, 312, 416 MHz}. The energy consumption at a given frequency is calculated using Equation 1. In practice, the C_3, C_2, C_1, and C_0 terms in $E(S_i)$ depend upon the power management state of the system and its subsystems [10]. We used the three energy settings E_0, E_1, and E_2 shown in Table 1 from [10] (these settings are also used in [14]).

Table 1: Energy Models

Models	C_0	C_1	C_2	C_3
E_0	0	0	0	f_m^3
E_1	$0.25f_m^3$	0	0	$0.75f_m^3$
E_2	$0.5f_m^3$	0	0	$0.5f_m^3$

We adopted a real-time target tracking WSN application [8] in our simulation study. We considered a maximum of 6 targets and thus 6 periodic tasks that are concurrently fired on a local cluster-head or a sink node due to them. The task parameters are shown in Table 2. We set two targets, associated with tasks T_1 and T_2, as animals moving at slow speeds, and hence resulting in *long* data generating (and thus task) periods. We set other two targets, associated with tasks T_3 and T_4, as human beings moving at moderate speeds and thus *moderate* data generating periods, and the last two targets associated with tasks T_5 and T_6 as vehicles moving at high speeds and *short* data generating periods. We assume that the physical measurements for each target is periodically reported to an embedded sink node/local cluster head. We also assume that the underlying communication has negligible overheads (not the focus of our work).

The statistical task execution time demand (c_i) was determined through numerous experiments to obtain the bounded value, according to Section 3.1. $\rho = 95\%$ for all tasks.

We compared MK-LfUA against algorithms that target different subsets of its problem space including DBP [7], EDF*+DRA [2], MK_E{R} [11], ReUA [14], and BaseEDF. DBP is the distance-based priority algorithm for (m, k) constraints, but it does not consider DVS. EDF*+DRA is a DVS real-time scheduling algorithm, but it does not consider (m, k) constraints. MK_E{R} is a DVS real-time scheduling algorithm for (m, k) constraints, but it does not consider overloads or synchronization. ReUA is a DVS real-time scheduling algorithm that considers overloads and lock-based synchronization, but it does not consider (m, k) constraints. BaseEDF is EDF with maximum CPU speed.

6.1 Energy Consumption

We compared the system-level energy consumption of the algorithms under different power models. Figure 1(a) shows the convex characteristic of our power model. For energy setting E_1, the CPU speed S_{low} that minimizes system-level energy consumption is 0.55 (approximately), and the corresponding CPU frequency is 208Mhz for Intel PXA271. For energy model of E_0, $S_{low} = 0$, and the system-level energy consumption increases monotonically with the CPU speed.

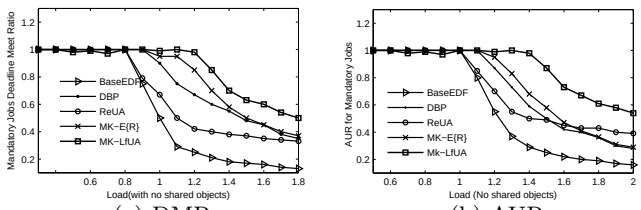
Figures 1(b) and 1(c) show the normalized energy consumption of different algorithms under energy settings E_0 and E_1, respectively. From Figure 1(b), we observe that during underloads, most algorithms consume the same energy. This is because, the E_0 model only considers the CPU's energy consumption and all algorithms seek to set as low CPU speed as possible. During overloads, all algorithms consume the same energy, as they all set the CPU speed as $S_{max} = 1.0$.

From Figure 1(c), we can observe that MK-LfUA saves at least 10% more energy than other schemes during underloads (i.e., total CPU load < 1). This is because, MK-LfUA statistically estimates task execution time demands, which gives a tighter bound than WCET. This results in reclaiming less slack time at run-time and less speed transition overheads under MK-LfUA than other algorithms. Furthermore, MK-LfUA sets an optimal CPU speed for each job (offline), yielding the minimum possible initial energy consumption.

During overloads, almost all schemes consume the same energy as they set the maximum CPU speed ($S_{max} = 1.0$).

6.2 Timeliness Assurance

In our first set of experiments, we evaluated the statistical timeliness assurances of the algorithms. Figure 2(a) shows the Deadline Meet Ratio (DMR) and Figure 2(b) shows the Accrued Utility Ratio (AUR) of the mandatory jobs under no shared objects. DMR is the ratio of the jobs meeting their deadlines to the total job releases of all tasks, and AUR is the ratio of accrued aggregate utility to the maximum possible utility. Since the AUR is strongly affected by the DMR, they have similar trends. Compared to other algorithms, MK-LfUA achieves at least 10-30% DMR and AUR during overloads, since it can distinguish mandatory jobs from optional jobs and favors jobs with higher utility.



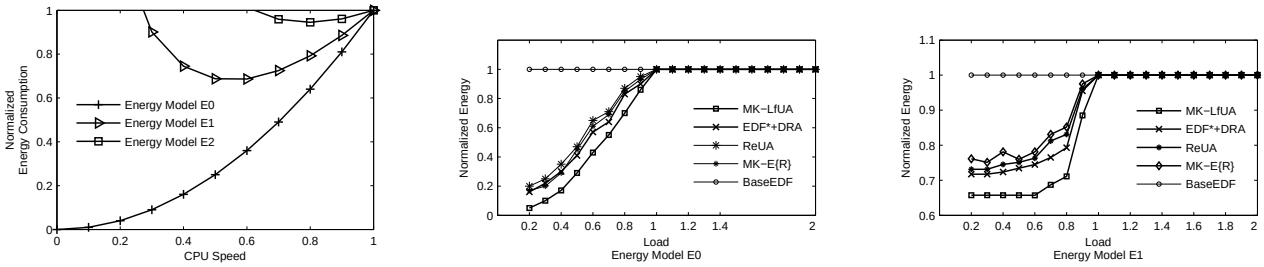
(a) DMR (b) AUR

Figure 2: DMR and AUR with no shared objects

To measure DMR and AUR under concurrent object sharing, we only compare MK-LfUA and ReUA, since only these

Table 2: Experimental Application Parameters

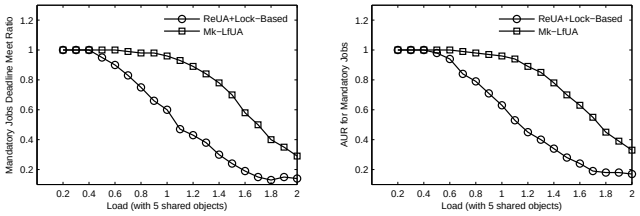
Task	Target Type	Jobs	Execution Time, c_i (ms)	Period (ms) Range	Utility	(m, k)
T_1	Animal	120	82	≥ 1000	20	$(3, 5)$
T_2	Animal	124	82	≥ 1000	20	$(3, 5)$
T_3	Human Being	124	110	≥ 500	40	$(4, 5)$
T_4	Human Being	109	110	≥ 500	40	$(4, 5)$
T_5	Vehicle	109	135	≥ 200	80	$(4, 5)$
T_6	Vehicle	124	135	≥ 200	80	$(4, 5)$



(a) Convex Feature of Energy Models (b) Energy Consumption for E_0 (c) Energy Consumption for E_1

Figure 1: Normalized Energy Consumption under Different Energy Settings

two consider synchronization, utility accrual scheduling, and DVS. We use five shared objects for all tasks. Figures 3(a) and 3(b) show the DMR and AUR of these algorithms, respectively, under different CPU loads. We observe that MK-LfUA outperforms ReUA by at least 20% in terms of DMR and AUR when the CPU load exceeds 1.0.



(a) DMR (b) AUR

Figure 3: DMR and AUR with shared objects

7. CONCLUSIONS

In this paper, we presented an algorithm called MK-LfUA that minimizes the system-level energy consumption through DVS, while meeting (m, k)-constraint with lock-free synchronization. The algorithm can be applied to embedded applications such as data aggregations in wireless sensor networks. During overloads, when the (m, k)-constraint cannot be met, the algorithm maximizes the accrued task utility. We determine the optimal CPU speeds which minimize energy consumption and derive tradeoffs of lock-free in offline stage. We also illustrated how to dynamically adjust the CPU speed considering the frequency transition overhead. Our simulation results illustrate that MK-LfUA can save about $15 - 20\%$ more energy compared to past algorithms, and that MK-LfUA meets more deadlines and accrues more utility for mandatory jobs than past algorithms.

8. ACKNOWLEDGMENT

This work was supported by the IT R&D program of MKE/IITA, South Korea.

9. REFERENCES

[1] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *RTSS*, pages 28–37, 1995.

[2] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *RTSS*, pages 95 –105, 2001.

[3] T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *ISLPED*, pages 9–14, 2000.

[4] J. Chen and A. Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus. In *Euromicro Workshop on Real-Time Systems*, pages 1–8, 1998.

[5] H. Cho, B. Ravindran, and E. D. Jensen. Lock-free synchronization for dynamic embedded real-time systems. In *ACM DATE*, pages 438–443, 2006.

[6] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, CMU, 1990. CMU-CS-90-155.

[7] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Trans. Computers*, 44(12):1443–1451, 1995.

[8] T. He, J. Stankovic, and T. Abdelzaher. Achieving real-time target tracking using wireless sensor networks. In *IEEE RTAS*, pages 37 – 48, 2006.

[9] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *RTSS*, December 1985.

[10] T. Martin and D. Siewiorek. Non-ideal battery and main memory effects on cpu speed-setting for low power. *IEEE Trans. VLSI Systems*, 9(1):29–34, 2001.

[11] L. Niu and G. Quan. Energy minimization for real-time systems with (m,k)-guarantee. *IEEE Trans. on VLSI Systems*, 14:717–729, 1997.

[12] G. Quan and X. Hu. Enhanced fixed-priority scheduling with (m,k)-firm guarantee. In *RTSS*, pages 79 – 88, 2000.

[13] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

[14] H. Wu, B. Ravindran, and E. D. Jensen. Energy-efficient, utility accrual scheduling under resource constraints for mobile embedded systems. In *ACM EMSOFT*, pages 64–73, 2004.