A User-Customizable Energy-Adaptive Combined Static/ Dynamic Scheduler for Mobile Applications[1]

Trolan C.-L. Ma and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science,
The University of Michigan
1301 Beal Avenue, Ann Arbor, MI 48109-2122
{trolanma, kgshin}@eecs.umich.edu

Abstract

In portable applications, the energy consumed by OS and application tasks primarily comes from DC battery source, which is limited and imposes an upper bound to the amount of time available for execution of tasks. There has been research in the area of Energy-aware Quality-of-Service (EQoS) that focuses on saving energy of mobile applications with an inevitable but graceful degradation of performance. However, most of these EQoS approaches emphasize only the need to conserve as much energy as possible while paying relatively less attention to the intrinsic criticality of the tasks themselves. It is in fact as much important to prioritize the scheduling of more critical tasks over non-critical tasks in the limited-energy environment to improve the overall performance over a bounded duration in which energy is available. Using the Combined Static/Dynamic scheduler (CSD) in the EMERALDS operating system [14, 15] as a basis, we develop an energy-aware scheduling algorithm so that it may execute tasks to achieve effective use of limited energy and ensure critical tasks to be favored in scheduling. The resulting product, called the ***Energy-Adaptive CSD (EA-CSD)****, considers the energy requirement and criticality of each individual task and yields a more energy-aware priority assignment to tasks which is flexibly determined by user-specified energy- and criticality-oriented objectives. Our simulation of the EA-CSD shows that battery life can be extended up to about 100% with varying degrees of performance degradation of up to about 40%, and the actual values of both of these are fully customizable by the user.*

1 Introduction

The Combined Static/Dynamic (CSD) Scheduler proposed by Zuberi *et al.* [14, 15], while ensuring scheduling overheads to be minimized, does not consider the energy requirement and the criticality of tasks in its scheduling decisions. In CSD, all tasks to be scheduled are queued in one of the two queues, namely, *Static-Priority* (SP) queue, which adopts mainly Rate-Monotonic (RM) scheduling, and *Dynamic-Priority* (DP) queue, which utilizes mainly Earliest-Deadline-First (EDF) scheduling policy. The assignment of each task to one of these queues is decided by the relative period length of that task to other tasks, which, in turn, determines the task priority.

1. The work reported in this paper was supported in part by the Office of Naval Research under Grant No. N00014-99-1-0345.

Typically, the DP queue contains tasks whose periods are shorter and therefore priorities are higher than those in the SP queue. Once tasks are in the DP queue, they are scheduled based only on EDF policy. Such a policy prioritizes tasks with earlier deadlines over those with later deadlines at any given instant and does not take into account such factors as energy demand and criticality of individual tasks in calculation of priorities.

There are mainly two undesirable effects from this non-energy-aware version of CSD. First, some of the important tasks with later deadlines may not be able to meet deadlines in a multi-task environment consisting mainly of short-period yet relatively less important tasks. Also, due to the lack of energy-awareness of CSD, it is possible that most energy-demanding tasks dominate the CPU over the less energy-demanding tasks due to their higher priorities (based on the period lengths), thereby reducing the battery life substantially that would otherwise be extended by reversing the order of execution between the two task sets.

In this paper, we propose a scheduling algorithm to achieve the dual goals of maximizing the battery life of portable applications while prioritizing critical tasks over non-critical ones as energy becomes scarce. The proposed solution is implemented on the *Energy-Adaptive CSD* (EA-CSD) which provides both the application developers and end users with a set of parameters for them to customize its behaviors based on the particular application requirements in terms of expected power-on period and expected performance level.

We will first address the energy-adaptive scheduling algorithm and present the semantics and motivation of the parameters in it. We will then incorporate the measured values of energy consumption of various system calls and task runs for the PalmPilot presented by Ellis [3] in simulating a typical example scenario having a mixed workload of tasks with varying levels of energy consumption and task criticality. This will be followed by an analysis of the simulation results obtained from a C++ simulator that models the EA-CSD. The simulation results basically demonstrate two important characteristics of EA-CSD: (1) the EA-CSD outperforms the non-EA-CSD in its ability to extend battery life; (2) the EA-CSD is capable of scheduling and completing more critical tasks before deadlines than non-EA-CSD in a bounded power-on time. Moreover, we will also study how the variation of some of the customizable parameters in the EA-CSD alters the outcomes of simulation and how they compare to their expected behaviors under the design goal of EA-CSD. Finally, we will discuss related work and future work in this area and conclude that the EA-CSD is better suited to conserving energy and critical task prioritization at minimal and acceptable expense of performance.

2 The Energy-Adaptive CSD (EA-CSD)

2.1 Assumptions

Before discussing the design details of EA-CSD, we would like to point out that many of the assumptions that apply to the non-EA-CSD presented in [14, 15] still apply here. **A1 - A5** are the original assumptions of the non-EA-CSD either explicitly stated or implied in [14, 15]. **A6 - A9** are new assumptions made specifically for simulating the EA-CSD.

A1. All tasks are periodic. This is not a limiting assumption since non-periodic tasks can be

handled by introducing periodic servers.

A2. All tasks have deadlines equal to the end of the period in which they are released.

A3. CPU time is the only resource considered for each periodic task.

A4. No precedence constraints exist between tasks. One can relax this assumption as done for non-energy-aware algorithms, but we will instead focus on energy conservation.

A5. Tasks are fully preemptable by other tasks ready to be scheduled and executed.

A6. The scheduler deals with a fixed periodic task set in the entire duration when there is still energy available. No new task is introduced in the middle of task execution.

A7. Tasks consume energy at constant rates during execution in the entire scheduling period.

A8. Task execution is governed by a reward function such that only tasks completed before deadlines have utility to the users. Tasks missing deadlines do not add to total utility.

A9. No energy replenishment is assumed in the middle of scheduling (see Section 2.2.6 for the case of energy replenishment midway through scheduling).

2.2 The Energy-Adaptive Scheduling Algorithm of EA-CSD

2.2.1 Calculation of Energy-Adaptive Dynamic Priority (EADP) for Each Task

Each periodic task under EA-CSD is assigned an EADP to represent its importance relative to other tasks in the system. All periodic tasks currently running, waiting in the queues and even those which still have not been released into the system at any instant are assigned an EADP. The EADP associated with a periodic task basically serves as a “mega-priority” in that it determines the value of another parameter (which, in this case, is the task period) for the task which, in turn, decides the priority of the task in the system as seen by the original non-EA-CSD.

In what follows, we describe the calculation of EADP for each task and define some non-standard parameters used in the calculation.

EADP of a task *i* in a set of *n* periodic tasks is a dimensionless quantity that can be calculated by:

$$EADP_i = (1-r)\varepsilon_i \frac{e}{E} + rI_i\left(\frac{E-e}{E}\right)$$

where r = user-defined ratio between weights of energy-oriented and criticality-oriented objectives $(0 \le r \le 1)$

$$\varepsilon_i = 1 - \frac{U_i w_i}{\sum_{j=1}^{n} U_j w_j} = \text{energy-oriented priority (EOP)} \quad (0 \le \varepsilon_i \le 1)$$

I_i = task-specific criticality-oriented priority (COP) $(0 \le I_i \le 1)$

E = total estimated energy budget available (Unit: Joule)

e = residual (remaining) energy at a certain time (Unit: Joule)

$U_i = \frac{c_i}{P_i}$ = utilization of task i in a period

c_i = execution time of task i in each period (Unit: ms)

P_i = period length of task i (Unit: ms)

w_i = power rating of task i (Unit: mW)

2.2.2 Energy-Oriented Priority (EOP)

EA-CSD can schedule tasks in an energy-efficient manner extending the battery life. This is accomplished by favoring tasks that tend to have a lower energy utilization as compared to other periodic tasks in the system. What EOP does is to evaluate a task's expected proportion in consuming the system energy among all the tasks in the system and assign a task priority based on this proportion. The more energy expected to be consumed by a task, the lower the EOP is, and vice versa. First, each task must have its own power rating (w). The power rating for a task can be measured by a power gauging tool like the PowerScope in [4]. Higher power rating for a task indicates more energy will be consumed within a certain period of time by the task. However, this power rating for the task alone is not enough to represent how power-hungry a task is. We know that the energy consumed by a task with a fixed power rating is proportional to the time that it gets executed. Due to this proportionality, if a task only executes for a negligible amount of time in each period, its contribution to energy consumption of the system would be minimal. So, the utilization (U) of the task is included to reflect its actual share of energy in the system. The product Uw of this task divided by the sum of Uw's of all tasks represents the energy proportion consumed by this particular task among all the tasks in the system. One minus this energy proportion results in higher EOPs for tasks consuming less energy and lower EOPs for tasks expected to have high energy demand.

2.2.3 Criticality-Oriented Priority (COP)

COP associates each task with a priority between 0 and 1 indicating the importance of meeting its deadline. Tasks that are highly critical and whose deadline misses may result in catastrophe are

assigned COPs close to 1. Tasks that are optional and do not result in visible degradation of performance if they execute less frequently or even miss deadlines are given COPs close to 0. For an application, COP of a task at any given time can be determined by dynamic computation which takes into account all the periodic tasks presently in the system and evaluates the relative importance of each of them. The dynamic computation is performed using a preprogrammed formulae that considers all the possible combinations of tasks executing in parallel in the system and decides on a COP for each task in each combination. If a new task is introduced into the system in the middle of a scheduling session, an interrupt is issued to the OS kernel and the COPs of all tasks need to be re-computed. However, this is assumed not to occur frequently as stated above. Besides, another way to determine COP is to have the users specify the COP of each task currently running themselves. However, this is more difficult to implement and any erroneous criticality specification by the users may result in a disastrous consequence.

2.2.4 User-Defined Ratio

The user-defined ratio r provides the means of communication between the user and the EA-CSD. Through this ratio, the user can instruct the EA-CSD whether it should place more emphasis in energy consumption or criticality of a task in deciding on the EADP of the task at any given instant. When $r = 0$, criticality is not a concern for the user and the EA-CSD does not take that into account when computing EADPs for all the tasks. The resulting schedule will purely favor tasks with low energy consumption, but not those with high criticality, generally leading to long battery life with low performance. When $r = 1$, energy is not so much an important issue to the user and so it is disregarded by the EA-CSD. The resulting schedule will have more critical tasks running than any other tasks which are not as critical, even if they have expected low energy consumption. This typically leads to shorter battery life, but performance is enhanced since more critical tasks are executed.

One can adjust r to an intermediate value between 0 and 1. For r in the range [0, 0.5), the energy consumption of a task plays a more crucial part in calculation of EADP and we say the energy-oriented objective is favored as tasks with low energy demand are preferred by EA-CSD. But, for the range (0.5, 1], the criticality of a task dominates in the resulting EADP, and task criticality is favored as EA-CSD tends to schedule more critical tasks than low-energy tasks.

2.2.5 Weights of EOP and COP in EADP as Energy Wanes

The formulae for computing EADP is composed of the sum of two terms. The *energy consumption term* $(1-r)\varepsilon_i \frac{e}{E}$ encapsulates the EOP ε_i and the *criticality term* $rI_i\left(\frac{E-e}{E}\right)$ includes the COP I_i. These two terms are proportional to $\frac{e}{E}$ and $\frac{E-e}{E}$ respectively, and the EOP and COP are therefore weighted differently as e changes. At the begining when the energy level is E, the criticality term vanishes and the energy consumption term dominates in the calculation of EADP. Since e decreases as time elapses, one can expect that the weight of the EOP is gradually decreasing while COP is weighted more and more heavily. This demonstrates that as the energy level wanes, EA-CSD places a heavier weight on the criticality of tasks in its schedule and tends

to schedule them more often, taking less into account the tasks' energy consumption in its scheduling decisions. So, generally, EA-CSD biases more toward critical tasks as energy becomes scarce, because it ensures more critical tasks' deadlines to be met, and reduces the dependency between meeting critical tasks' deadlines and the residual energy level in the application.

2.2.6 When the EADPs are Computed and Updated

When the mobile application begins executing a task set, the EADPs of all tasks are computed using information about energy consumption and criticality for individual tasks. Then, during the scheduling process of tasks in EA-CSD, the EADPs of tasks are regularly updated and the interval between two successive updates is directly controlled by the user through the adjustment of update factor f ($f \geq 1$). Suppose an update occurs now and the remaining energy in the system is e. Then, when the residual energy reaches the next update energy e_u, the EADPs will be updated for all tasks, where e_u is computed by:

$$e_u = e - \frac{e}{f}$$

The updates will contine with the next update energy calculated using the above formulae at each update until the remaining energy e is lower than a threshold energy e_t calculated by:

$$e_t = EF_t$$

where E is the total energy estimated in the system initially and F_t is the low threshold factor, which is usually decided by the application developer based on the usual characteristics of tasks in the system. This threshold prevents the system from infinitely updating the EADPs of tasks since the formulae for calculating the update energy will be executed without any stopping condition.

For example, if E = 2500, f = 2, F_t = 0.1, the updates will occur at e = 2500, 1250, 625, 312.5. The next update point after 312.5 (156.25) is smaller than $2500 \times 0.1 = 250$ and therefore EADPs will not get updated, starting from e = 156.25 and on.

The update factor specifies how often the EADPs of tasks need to be updated. The more often the update, the larger the update overhead is. A larger number of updates reflects more accurately the current priorities of tasks in the system. However, if the number of updates is far too large, it can use up large portions of CPU cycles and energy, causing a significant degradation on system performance and shortening the battery life.

If the energy is replenished in the middle of scheduling, an interrupt is issued to the OS and the total amount of energy after replenishment is assigned to E. The EADPs for all tasks are re-computed with the system's remaining energy e now reset back to E. Using the update factor f and low threshold factor F_t, e_u and e_t are re-computed and the scheduling session progresses in the usual way using the updated values of the above parameters after returning from interrupt routine,

starting from energy level *E* and original task periods (or modified task periods based on amount of energy replenishment). This kind of replenishment is assumed to occur infrequently during scheduling (once or twice during the entire power-on period) since frequent interrupts involve costly context switches undesirable for real-time applications.

2.2.7 How the Updated EADPs are Used - Stretching of Task Period

At any time during the scheduling of tasks in EA-CSD, the most current EADP of a certain task in comparison of other tasks' EADPs translates to a priority position of that task as seen by the scheduler. Theoretically, EA-CSD will prefer running tasks with higher EADPs since they are the tasks regarded as most important from both the energy- and criticality- perspectives by the user. However, EADP alone does not specify the strict priority of tasks in terms of order of execution for the scheduler. What still remains as the decisive factor in EA-CSD is the task period. As stated earlier, in CSD, short-period tasks usually are in the *Dynamic-Priority* (DP) queue and are scheduled based on EDF. They have higher priorities than all longer-period tasks in the *Static-Priority* (SP) queue scheduled by the RM policy.

Bearing this in mind, the EADP of a task is utilized to determine whether a task's period should be stretched (less frequent execution), and if so, how much it needs to be stretched. A task which has an EADP lower than the maximum EADP among all tasks at any given time is considered to be less eligible to run than the maximum-EADP task. Its period needs to be stretched so that its actual priority is lowered than before (assuming it is in the SP queue). If the stretch is sufficiently large enough for a task originally in the DP queue, a task can be demoted from the DP queue to the SP queue and be scheduled based on their periods only after all the tasks in DP queue complete execution. By stretching the correct tasks, EA-CSD attempts to dynamically adjust the actual relative priorities of tasks in the system reflective of the user-specified goals, be it the energy-oriented objective or the criticality-oriented objective.

The amount of period stretch for a task at any update point is directly proportional to the difference between the task's EADP and the maximum EADP among all tasks at that time. For example, suppose at the updating time of EADP, task *i* has EADP equal to $EADP_i$ and the maximum EADP among all the tasks is $EADP_{max}$. Then, the period stretched S_i for task *i* is calculated by:

$$S_i = \frac{EADP_{max} - EADP_i}{EADP_i} \times k$$

where *k* is called the *stretch constant* and is application-dependent, and the new period for task *i* ($P_{i,new}$) is:

$$P_{i,\,new} = S_i + P_{i,\,old}$$

However, tasks cannot be infinitely stretched in their periods. An upper bound on task period as compared to the initial unstretched period is necessary because if tasks are infinitely stretched in

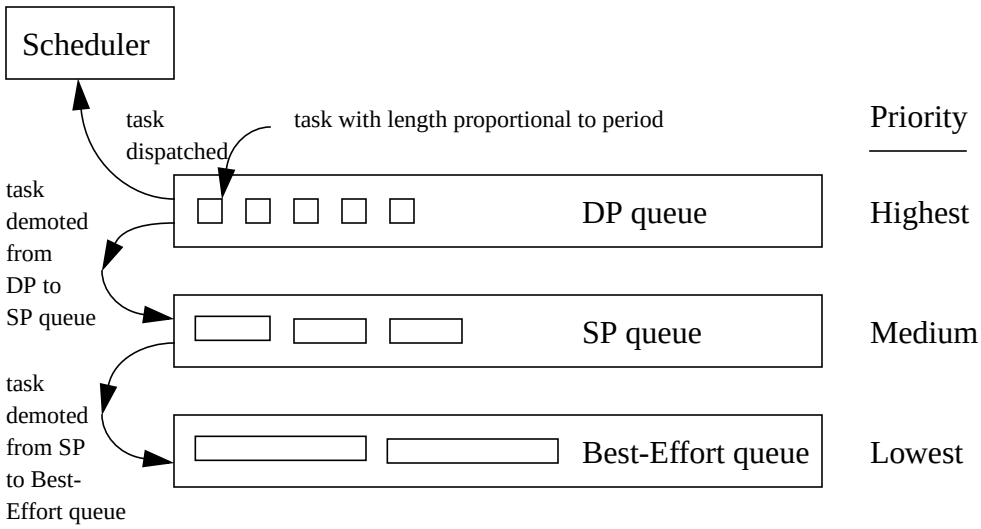

Figure 1. Task queues in EA-CSD including the added Best-Effort queue containing all the tasks with periods sufficiently stretched beyond those in SP queue.

their periods, they are invoked less frequently to an extent that causes the tasks to lose their functionality in performing useful work that is most often real-time in nature. Thus, every task has an upper bound in task period calculated based on a maximum stretch factor F_s for the application. This factor can be individually assigned on a per-task basis or one factor can be applied to all tasks in a certain application. The resulting upper-bound period of a task P_{max} having initial period P_o scaled by maximum stretch factor F_s is calculated as:

$$P_{max} = P_o F_s$$

2.2.8 Improved Schedulability of Resulting Task Set

When a task's period is stretched, the total processor utilization $U = \sum_i \frac{c_i}{P_i}$ for all tasks decreases, improving schedulability. Therefore, without any energy constraints, the resulting task set after any deadline-/period- stretching operations is still schedulable if it was already so before the energy-aware operations are applied to the tasks.

2.2.9 Addition of "Best-Effort Queue"

We added a third queue called "Best-Effort queue" as shown in Figure 1, whose tasks have lower priorities than those in SP queue, to EA-CSD in order to accommodate the tasks whose periods have been significantly stretched beyond the maximum-period task in the SP queue. These tasks are among the least important tasks in the task set and are not executed unless all the tasks in both the DP queue and SP queue have completely finished. Besides, there is no guarantee that these

tasks in Best-Effort queue will ever be executed at all before the battery runs out of energy (hence the name "Best-Effort" due to its lack of guarantee). If this happened, the non-executed tasks in the Best-Effort queue would represent only acceptable degradation of performance for the portable application. They have virtually been avoided to be executed by EA-CSD because their characteristics do not conform to the user's currently specified objective and whether they get CPU cycles does not materially affect the functional integrity of the system. They may be too power-hungry if the user asks for longer battery life or too trivial if the user specifies more critical tasks to be executed.

3 A Case Study - The PalmPilot

3.1 Selection of Periodic Task Set for Simulation

We developed an EA-CSD simulator (written in C++) that fully models the EMERALDS OS environment and performs all the scheduling activities starting from the time when energy source is renewed to the point when energy is completely exhausted. It also displays statistics about the number of invocations and timely completions for each task at the end of the run, as well as the final battery life and the quality index as a measurement of overall average performance of the system in the entire power-on period. Interesting conclusions are then drawn from these results, including how EA-CSD outperforms non-EA-CSD and how variation of parameters affects the performance of EA-CSD.

The simulation of the EA-CSD is run on a typical task set consisting of a mixed workload with varying degrees of energy consumption rate and criticality. To this end, we have chosen several tasks in a PalmPilot Professional running PalmOS version 2.0.4 with energy consumption rates measured by Ellis [3]. The task set is selected from the application Hiker's Buddy, which takes NMEA protocol strings sent by a Global Positioning System (GPS) receiver and plots the hiker's current location on a map downloaded from the server. Hiker's Buddy searches the downloaded map to spot the exact point where the hiker stands in the map. Since it is logical to think that the downloaded map cannot be infinitely large to cover everywhere in minute details in the world and only a portion surrounding the current location on the map is needed at any time, we assume that as the hiker (user) changes location constantly as he/she moves, the map needs to be downloaded again in a roughly periodic manner, making it a periodic task candidate in our simulation.

The task set running under EA-CSD consists of a total of 7 periodic tasks. The first 4 tasks include primarily critical tasks integral to the running of the Hiker's Buddy. The other 3 tasks are comparatively less critical, with the first two tasks being some background batch jobs consisting of register-based computation loops and the last one being the periodic task for keeping the LCD display of the PalmPilot on.

Table 1 shows the profiles of the 7 selected periodic tasks in different aspects including task execution time and period, as well as power rating and criticality. All the power ratings are directly extracted from the corresponding measurements in [3] for higher accuracy. All the remaining parameters are values closely approximating the actual situation while the Hiker's Buddy is concurrently running with some background jobs.

.

Task ID#	Description	Execution Time (ms)	Period (ms)	Criticality (COP)	Power Rating (mW)
1	Parsing NMEA sentences + receiving from GPS	5	100	0.8	90
2	Keeping serial line open to GPS	7	40	0.95	60
3	Downloading map from serial line	10	100	0.9	150
4	Searching map with memory-intensive computation	10	30	0.8	140
5	Background looping register-based computation job #1	6	50	0.3	125
6	Background looping register-based computation job #2	3	20	0.2	125
7	Keeping LCD display on	10	150	0.1	40

Table 1. Profiles of the 7 selected periodic tasks in simulation of EA-CSD running on PalmPilot.

3.2 Time Trace of Task Set

Figure 2 shows the time trace of the 7 periodic tasks running under EA-CSD in PalmPilot under the conditions: $r = 0.9$, $f = 10$, $F_t = 0.1$ and $k = 10$. The task queues are polled every 5000s and the length of the numbered bar is proportional to the period of the task with that ID number. We see that initially at time t = 0, EADP is first computed and the task periods begin to adjust depending on the relative magnitudes of their EADPs. At t = 500 (5000s), we can already see a conspicuous change in the task period and priority position (assuming all tasks are in SP queue, which in reality is not necessarily true) for Task 6. Task 6 (COP = 0.2, $w = 125$) can be classified as a non-critical task with high energy consumption and is the kind of task biased against most by EA-CSD. Therefore, it is the first task which has its period stretched substantially from the outset. As a result of the stretch, its priority position changes to 3rd from 1st in a period of 5000s. At t = 1500 (15000s), while Task 6 continues to have its period stretched as it becomes less eligible for scheduling, the period of another less critical and power-hungry Task 5 (COP = 0.3, $w = 125$) also gets stretched enough that it is demoted from priority position 4th to 6th. At t = 2000 (20000s), Task 6 declines to 4th position in priority. So, up to this point, the effect of EA-CSD in favoring tasks with high criticality and low energy consumption speaks for its ability to schedule tasks to satisfy both energy- and criticality-oriented objectives.

With $r = 0.9$, the criticality of tasks is of utmost importance in making scheduling decisions. Task 2 (COP = 0.95) and Task 3 (COP = 0.9) are the 2 most important tasks in the system as judged by the user. As a result, Task 2 and Task 3 do not suffer from any period stretch at all and remain two of the tasks cared most by EA-CSD. As Task 5 and Task 6 have their periods stretched past those of Task 2 and Task 3, Task 2 and Task 3 climb to priority positions 1st and 3rd from 3rd and 5th,

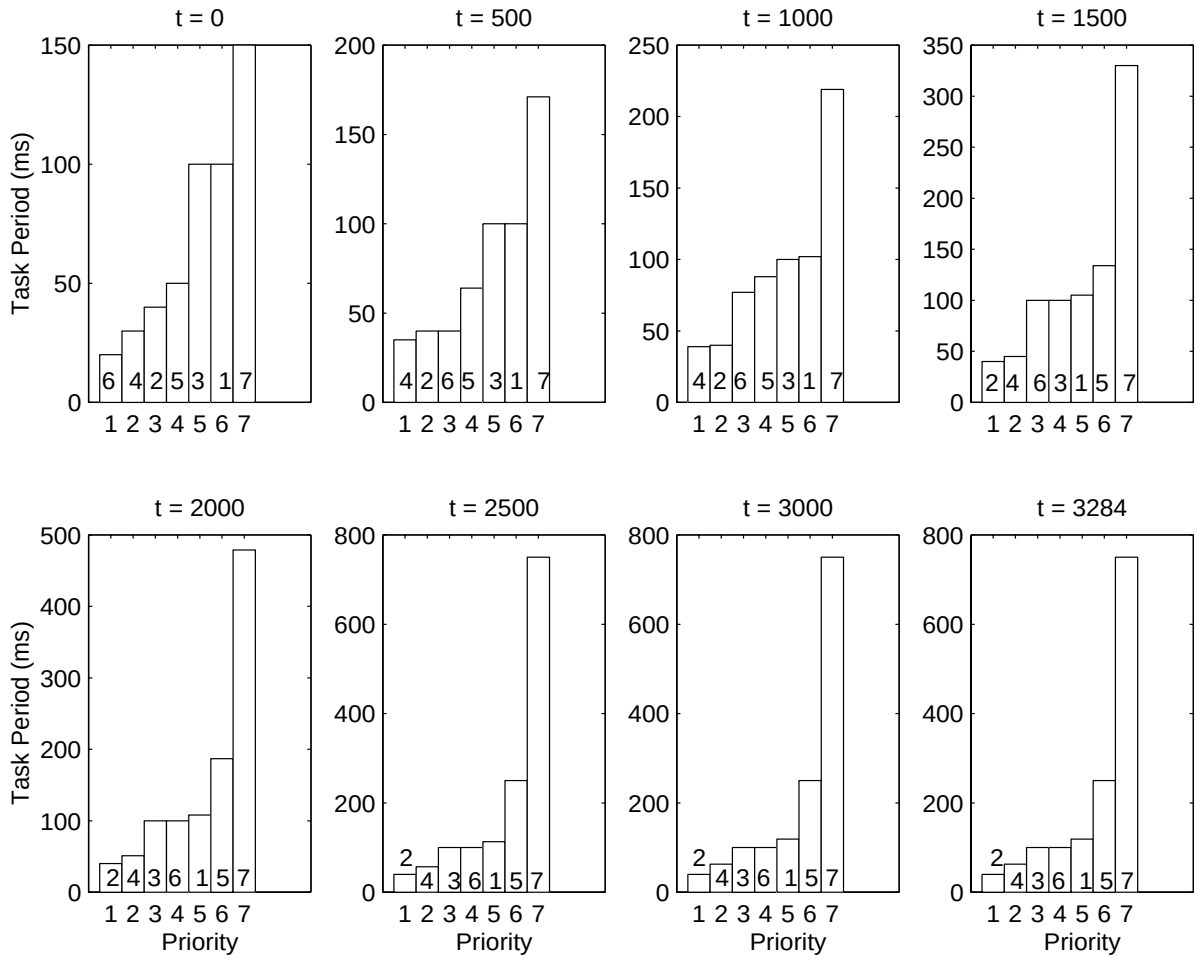

Figure 2. Time trace of task periods for the 7 periodic tasks running under EA-CSD in PalmPilot. Numbers inside bars indicate task ID# and tasks are arranged in ascending order of task period. The numbers under each diagram reflect task priorities if all tasks are in SP queue. Generally, tasks having low criticality and high energy consumption get their periods stretched more often than tasks with high criticality and low energy consumption. All times t are in unit (X 10s).

respectively.

As energy becomes seriously scarce at the end of the power-on period, Task 5 and Task 6 basically have their periods constant and their priority positions fixed since they have reached the upper bound of the period as explained above. However, at this point, the periods of all tasks are already very indicative of the actual priority that each task deserves under energy-adaptive and criticality-aware scheduling. Least critical Task 7 (COP = 0.1) remains the task having the longest period all the way, which is just another manifestation that low criticality asks for more period stretching.

3.3 Task Invocation and Execution Profiles

The bar chart in Figure 3 compares the numbers of tasks invoked and completed within deadlines over the entire duration when energy is not exhausted on the PalmPilot, using the same parameters

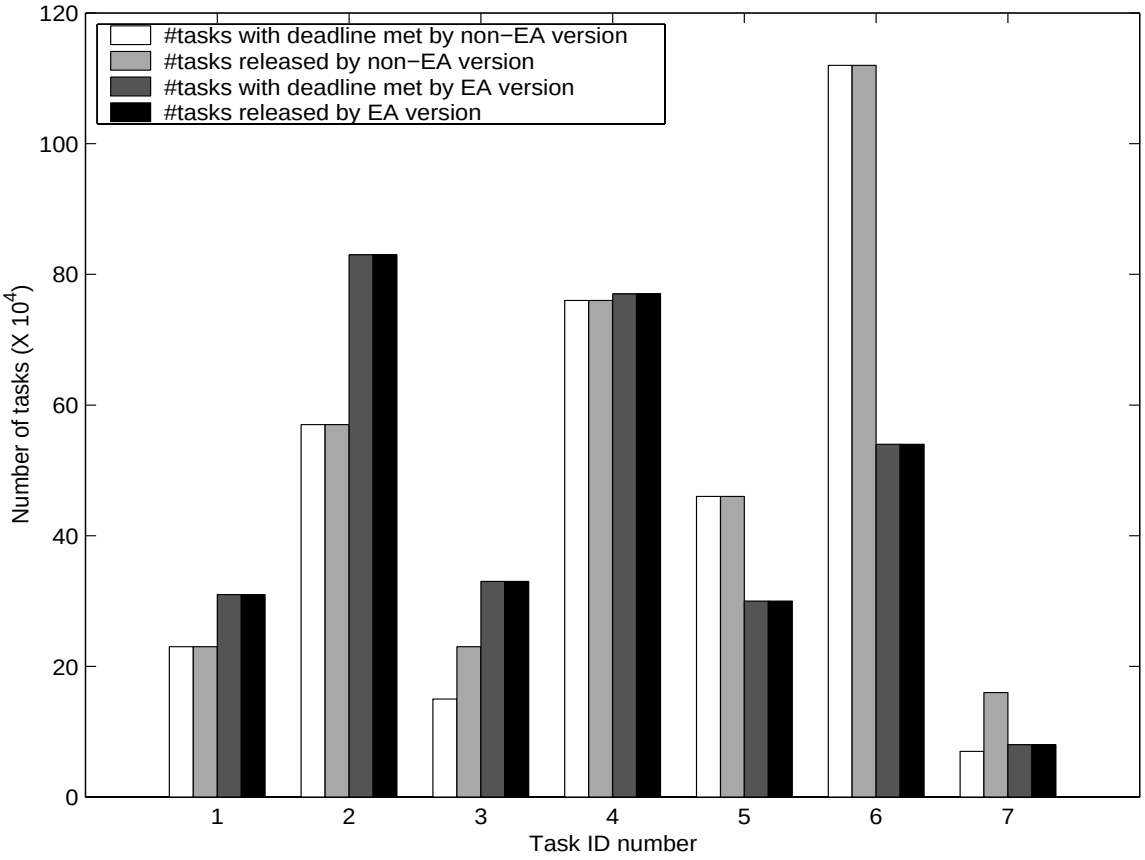

Figure 3. Number of tasks released and meeting deadlines during the entire power-on duration of the PalmPilot for non-EA- and EA-CSD.

as that in Section 3.2. As we can see, the criticality dominates in the EA-CSD's decision to stretch task periods. So, two of the least critical tasks, Task 5 and Task 6, have a lower number of tasks invoked and completed in a timely manner in the EA version than in the non-EA version. The difference between the non-EA and EA versions is particularly pronounced for Task 6, which has a marginally lower COP than Task 5 and has dropped more than 50% in the number of invocations and timely completions. Task 7 has more invocations in the non-EA version, but its invocation is stifled in the EA version since it is the least critical task among all 7 tasks and its period gets stretched enough that it remains at the bottom of the priority throughout. On the other hand, the two most important tasks in the system, Task 2 and Task 3, both have substantially larger number of invocations and completions before deadlines in EA version than in non-EA version. Particularly, Task 2 has the highest COP and lowest energy consumption among all tasks and is therefore heavily favored by EA-CSD. Its numbers of invocations and timely completions both have jumped up by more than 45%. Task 1 (COP = 0.8) and Task 4 (COP = 0.8) have slightly lower COPs and are not as heavily favored by EA-CSD as Task 2 and Task 3. Both Task 1 and Task 4 demonstrate a slight increase in both EA-version numbers over their non-EA version counterparts, but not as much as in the cases for Task 2 and Task 3. Comparing Task 1 and Task 4, Task 4 has less improvement using EA version. This is because even though Task 4 is critical, the energy consumption rate for Task 4 is much higher than Task 1 and the energy adaptability of the

EA version is not inclined to schedule more power-hungry tasks.

3.4 Non-EA-CSD vs EA-CSD - Which One Performs Better?

In order to compare the performances of non-EA-CSD and EA-CSD and see the value of incorporating energy-adaptive capability in the latter, it is necessary to define certain performance metrics for comparison. We will use several performance metrics (**M1 - M3**) to compare the difference between the two versions of CSD as follows.

M1. *Battery life* provides a concrete measurement of how capable the two versions of CSD are in conserving energy by executing tasks that have lower energy consumption first.

M2. *The total sum of COPs of all tasks that have completed before their deadlines in the whole power-on period* provides an idea of the level of overall criticality of tasks executed and how the two versions fare against each other in favoring more critical tasks as energy wanes. This measures the absolute total level of criticality achieved using a fixed budget of energy.

M3. *The total sum of COPs (**M2**) divided by the battery life (**M1**)* provides a quality index which is decided by the amount of COP obtained per time unit in the entire power-on period. This is necessary because the total sum of COPs in **M2** does not take into account the battery life over which the COPs are accumulated. Total COP can go up just because more tasks are invoked and executed in a longer period of battery life. As a result, the quality index is a metric for measuring the average criticality achieved over time and is indicative of the overall performance for the application.

In all of the following simulation experiments, the effect of varying one parameter at a time in the EADP equation is analyzed to see the sensitivity of the above performance metrics.

3.4.1 Effect of Variation of User-Defined Ratio *r*

This part of simulation involves varying the user-defined ratio r to see if it can achieve the effect that we expect. The simulation uses the following values of parameters: $F_t = 0.1$, $k = 10$. In previous sections, we expect that when r is close to 0, the energy consumption of a task takes on more importance than the criticality of the task in scheduling decisions. On the contrary, as r gets closer to 1, the criticality will gradually outweigh the energy consumption of a task in priority determination, even though initially when there is enough energy, energy consumption still has a lingering effect in extending battery life.

Figure 4 gives a comprehensive picture that is very close to our expectation. For the curves labelled with $f = 10$, $f = 50$, $f = 100$ in Figure 4(a), we see a general trend that the curves remain pretty much constant (or go up somewhat) and then undergo an abrupt slump as the user-defined ratio increases from 0 to 1. When the user-defined ratio is near 0, the energy consumption term dominates in deciding period stretch, and more energy is conserved than when the user-defined ratio is large, where the energy consumption term is pretty much ignored. Therefore, more critical tasks are scheduled to meet deadlines (as demonstrated in Figure 3 with $r = 0.9$) at the expense of energy-ignorant scheduling by EA-CSD as r approaches 1. Energy-ignorant

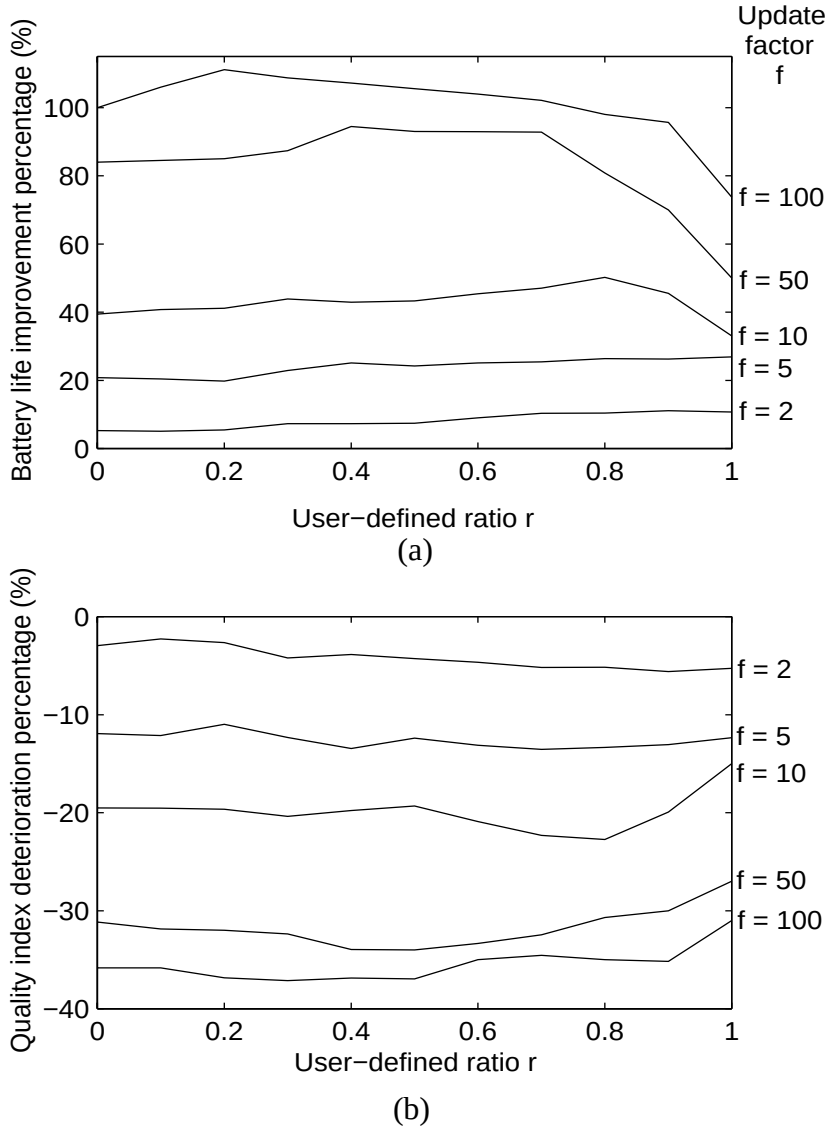

Figure 4. Battery life improvement and quality index deterioration with varying user-defined ratio *r* for EA-CSD. Curves only begin to look concave in both areas when *f* is greater than or equal to 10. Battery life improvement tops out 100% with 40% deterioration of performance when *f* = 100.

scheduling simply leads to shorter battery life since some power-hungry tasks can then have the chance to occupy more CPU cycles.

The near-constant slope in the intermediate range of these curves can be explained by the lingering dominance of the energy consumption term over the criticality term in EADP computation in the early stage of scheduling, as well as the peculiarity of the task set chosen. In this range of *r*, since the weight of energy consumption term is more important than the criticality term in calculating EADP when energy is still near full (i.e., $e \approx E$), some high-energy tasks get their periods stretched early, saving much of the energy. The period stretching of primarily less critical tasks when energy is near exhausted (i.e., $e \approx 0$) occurs late or not frequently enough

(due to relatively low update factor) so that the energy sacrificed in energy-ignorant scheduling is not sufficient to offset the energy savings earlier. Combining with the particular values of power rating and criticality of the task set chosen, this explains why the battery life improvement remains at a near-constant level in the intermediate range of *r*.

For the curves labelled with $f = 10$, $f = 50$, $f = 100$ in Figure 4(b), we see the reverse trend of the curves as compared to Figure 4(a) which reveals the complementary relationship between battery life and quality index. It also demonstrates the tradeoff between them and one cannot easily achieve both with a fixed amount of energy resources. As *r* nears 1, more critical tasks get completed before deadlines, adding to the total COP which translates to a higher quality index (less deterioration) within a shorter battery life.

In both Figures 4(a) and 4(b), for the curves with *f* lower than 10, the relatively low update factors do not provide sufficient number of updates of EADPs of tasks and therefore task periods are stretched less often. Even if *r* is close to 1, critical tasks cannot overtake less critical tasks in terms of priority position nearer toward energy exhaustion with that limited number of period stretch. As a result, the energy consumption term still dominates in scheduling decisions. This explains why the curves are not as concave as those with higher update factors, and battery life does not go back down as *r* approaches 1 in Figure 4(a).

3.4.2 EA-CSD Extends Battery Life at Expense of Performance Degradation

From Figure 4(a), it is easy to see that when *f* is greater than 1, the battery life is extended over the non-EA version no matter what the user-defined ratio is. However, an accompanying undesirable effect is the corresponding deterioration (negative percentage indicates deterioration in Figure 4(b)) of quality index, which represents a kind of performance degradation over the non-EA version. As we will see later, this tradeoff between battery life and performance degradation is an intrinsic one, just as it is similarly claimed in [5]. The key point for the design of EA-CSD is to be able to let the users to determine the particular level of battery life and performance desired themselves through a set of user-adjustable parameters.

One conclusion that we can immediately draw from Figure 4 is that unless the update factor in EA-CSD is equal to 1, in which case the EA-CSD degenerates to the non-EA-CSD since tasks never get updated in period, battery life is extended and performance is deteriorated in general. This conclusion is applicable to all cases that we will consider from this point on.

3.4.3 Effect of Variation of Update Factor *f*

3.4.3.1 Battery Life Extension and Performance Deterioration

Next we will see the effect of changing *f* on battery life and quality index. When $f = 1$, from the formulae of calculating the next update energy e_u, the EA-CSD does not update the EADPs of individual tasks in the system at all and hence tasks are scheduled in the same way as non-EA-CSD without energy-adaptive updates. Therefore, as *f* gets closer to 1, EA-CSD does not provide any significant extension of battery life and it basically degenerates to non-EA-CSD. This is demonstrated in Figure 5, where there is 0% improvement on battery life and 0% decline in

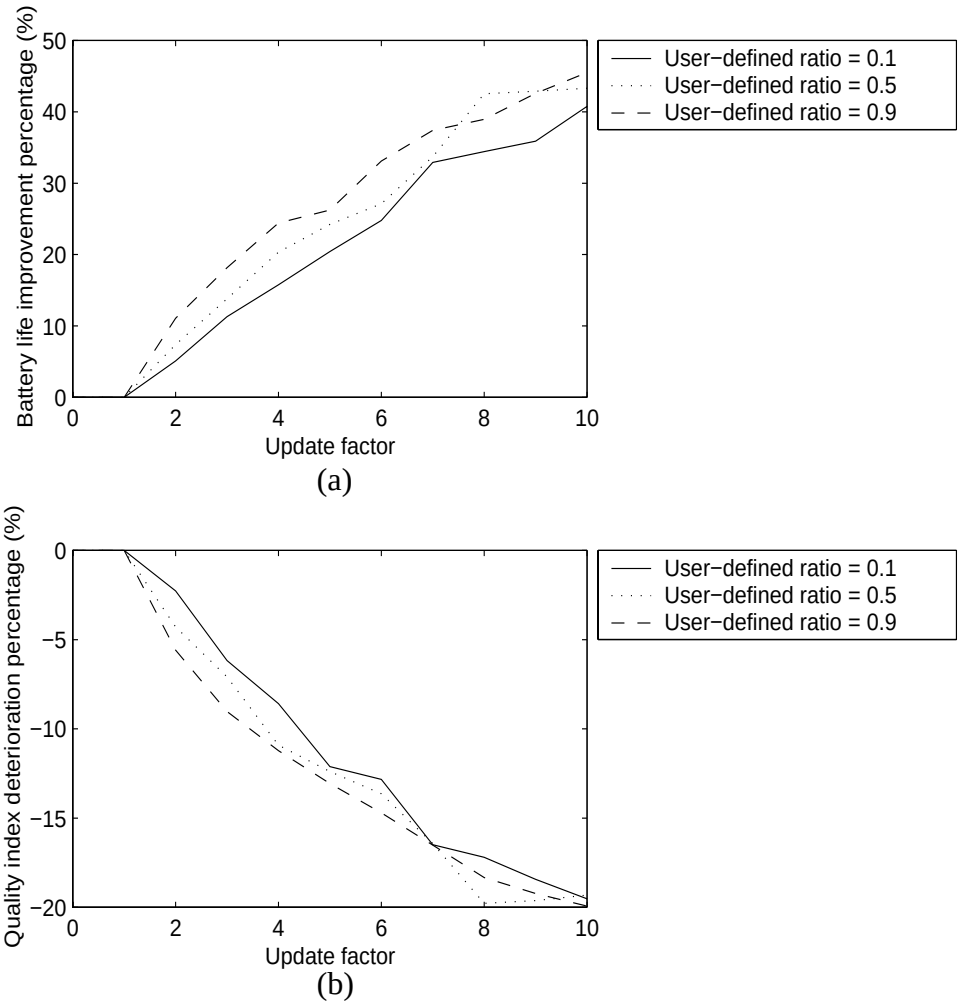

Figure 5. Battery life improvement and quality index deterioration with varying update factor *f* for EA-CSD. Battery life extends more and quality index declines as update factor increases starting from 1.

quality index as compared to non-EA-CSD when $f = 1$.

As *f* increases from 1 to 10, it is easy to see that battery life gets longer with a corresponding deterioration in performance. This can basically be attributed to the fact that more updates mean periods of tasks of high energy consumption are more often stretched, allowing those less power-hungry tasks to take over the CPU more frequently than tasks consuming more energy. As a result, battery life can be extended by as much as 40% with a moderate update factor of 10. The corresponding decline in quality index can be explained by the fact that the total amount of COP earned during the power-on period does not grow as fast as the rate of extension in battery life. A smaller amount of COP is achieved per time unit. Quality level declines by around 20% as *f* increases to 10.

In general, for any fixed *f*, there is a slight increase in battery life extension and minimal decrease in quality index as one goes from the user-defined ratio *r* of 0.1 to 0.9. As mentioned in Section

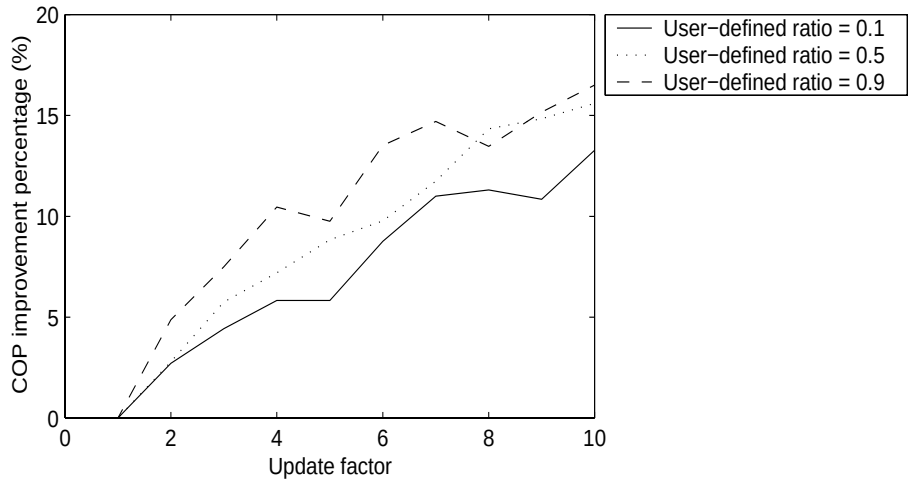

Figure 6. Total amount of COP earned during a power-on period, indicative of the total level of task criticality achieved.

3.4.1, the update factor ranging from 1 to 10 is moderate enough that power-hungry and less critical tasks cannot be sufficiently demoted to lower priority by having their periods extensively stretched. As a result, the energy consumption term in EADP calculation still has lingering dominance in determining relative priorities of tasks even as r tends to 1. But comparing the r values in the range from 0 to 1, as r increases, high-energy, less critical tasks get their periods stretched to a larger extent, creating relatively fewer periods with high-energy workloads. This increases the battery life by a minimal amount as r goes from 0.1 to 0.9.

3.4.3.2 Total amount of COP

The absolute total amount of COP earned during the entire power-on period also rises as the update factor increases because less critical tasks have periods stretched more often as number of updates increases. An immediate result of that is the higher probability for more critical tasks to overtake less critical tasks in priority and execute before them. The change of total amount of COP earned as a measurement of the total level of task criticality achieved in the power-on period is shown in Figure 6.

3.4.3.3 Update Overhead Analysis

One may wonder from Figure 5 whether f can be *infinitely* increased to extend the battery life without bounds at the expense of a relatively less significant decline in performance. The answer to this question is "No". As exhibited in Figure 7(a) with $r = 0.9$, $F_t = 0.1$ and $k = 10$, as the update factor gets larger and larger to around 100, the battery life extension percentage starts to "tail off" and grows at a slower rate until it more or less becomes flat or even winds back down. This is due to the substantially larger overheads involved in calculating the EADP and updating task periods as the update factor rises. Similar situations also happen for performance deterioration in Figure 7(a) and COP improvement percentage in Figure 7(b).

The action of calculating the task EADPs requires the parsing of all the tasks in all queues in the

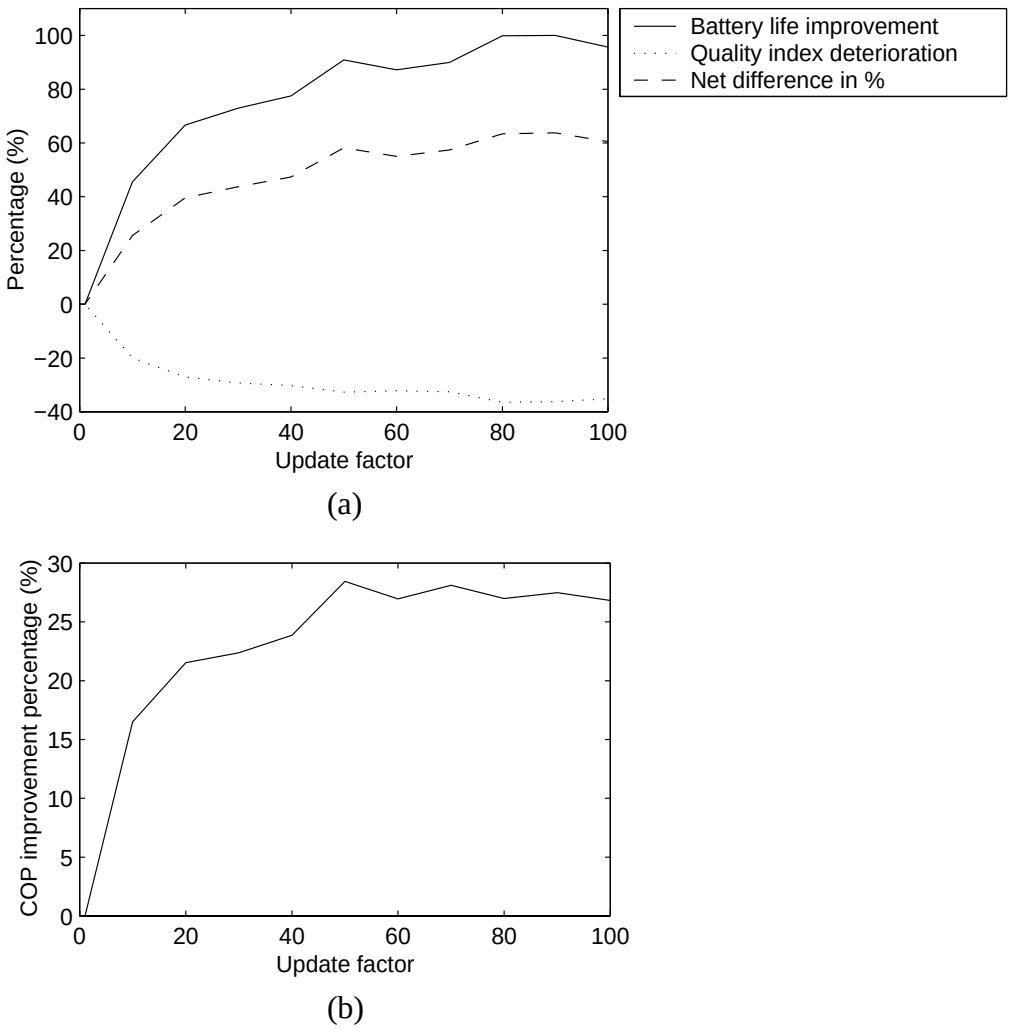

Figure 7. Battery life improvement, quality index deterioration and total COP improvement with varying update factor *f* for EA-CSD. Notice that all the curves tail off and even rebound back as update becomes more frequent due to large overheads involved in updating.

system and updating EADPs, which typically incur O(*n*) overheads for a total of *n* tasks in system. The check for maximum EADP among all *n* tasks can be done simultaneously by keeping track of the maximum EADP up to a point as EA-CSD calculates each EADP and only updates the maximum EADP in case there is a higher-EADP task. Then the EA-CSD parses through each task again in all queues and stretches the periods of tasks based on the maximum EADP just found, which incurs O(*n*) overheads again. The dominating factor in processing overheads should come from the final step in each update - sorting tasks based on the updated periods and demoting tasks from one queue to another if necessary. Assuming there are *m* tasks in the unsorted DP queue, there are (*n* - *m*) tasks in the sorted SP queue and Best-Effort queue. Using a simple sorting algorithm like an insertion sort, the average case sorting overhead is in the order of $O((n-m)^2)$. The demotion of tasks takes only negligible overheads compared to the sorting since it happens infrequently and not for each task in the queues. Any costs associated with demotion will also be amortized over a long period between successive updates. Table 2

summarizes all the essential steps in updating of tasks and their associated overhead costs.

Update Step Number	Description	Order of Overhead Costs
1	Calculation of EADPs and keeping track of maximum EADP thus far as EA-CSD goes through each task in queues	O(n) *
2	Update (stretching) of periods for all tasks in queues based on maximum EADP found in Step 1	O(n) *
3	Sorting of tasks based on updated periods in SP queue and Best-Effort queue and demotion of tasks between queues	O((n-m)^2) **

* Assuming there are totally n tasks in the system.
** Assuming there are m tasks in the DP queue and totally n tasks in the system.
Note: Total overhead is in order of O((m-n)^2) overall since Step 3 dominates over Step 1, 2.

Table 2. Steps in updating EADPs and periods of tasks and the ensuing sorting along with their associated orders of overhead costs.

For an update factor of lower than 100, total update overhead is small compared to the sum of the execution times of tasks and so, the overheads generated typically can be *amortized* over a long period of time between successive updates. However, as the update factor begins to exceed 100, the update overheads start to become comparable to the execution times of the tasks themselves, virtually overloading the system with extra energy-consuming tasks with useless values (of zero criticality). Therefore, the battery life extension percentage, performance deterioration percentage and COP increase percentage all gradually tail off and become flat or even rebound back as update factor increases beyond a certain threshold.

3.4.3.4 Update Factor *f* vs. User-Defined Ratio *r* for Customization

As introduced in Sections 2.2.4 and 2.2.6, both the update factor f and user-defined ratio r are user-adjustable parameters of the EA-CSD. The question is how these adjustable parameters should be used to suit a particular application's needs. We now know that changes in update factor and user-defined ratio can directly affect battery life, performance and total level of task criticality achieved in a bounded power-on period of the mobile application. Therefore, based on user preferences and the requirements for individual tasks, one can easily adjust the parameters either before or in the middle of the running of EA-CSD to achieve the desired goals.

A close examination of Figures 4 and 5 reveals that the user-defined ratio is coupled with the update factor and the former is dependent on the latter in order to carry out its desired functionality. For instance, user-defined ratio can only make significant impact on battery life and performance if the update factor is larger than a certain threshold, and in the above simulation, the update factor should be larger than 10. If the update factor is sufficiently large, the user-defined ratio can then be adjusted to either achieve a longer battery life with diminished performance or the opposite with a larger proportion of critical tasks timely executed in an energy-unaware manner.

Comparing Figures 4 and 5, we can also easily detect the difference of magnitudes of impact on

battery life and performance by the update factor and user-defined ratio. Generally speaking, the update factor is a more sensitive user-adjustable parameter which produces relatively drastic changes in battery life and performance while the user-defined ratio impacts slightly on those two, usually within only a limited range of values given a fixed update factor. For example, with r = 0.9, battery life improvement jumps surprisingly fast from 0% to about 45% as f increases from 1 to 10. On the other hand, with f = 10, the battery life improvement is 40% at r = 0 and drops to 30% at r = 1. The update factor is more sensitive than user-defined ratio in effecting changes in battery life and performance. It is therefore reasonable to use the update factor as a coarse-grained adjustment parameter. The user-defined ratio can be utilized as a fine-grained parameter for fine-tuning the metrics given a coarse level of them achieved by update factor. After all, the user-defined ratio still has the additional capability over the update factor to decide the bias toward either energy-oriented or criticality-oriented objective.

3.4.4 Effect of Variation of Stretch Constant *k*

Another customizable parameter for application developers, but not users, is the stretch constant k. k can be used to determine the extent to which the task periods are stretched during each update for those tasks that have lower EADPs than the maximum EADP. The larger the value of k, the more the periods of tasks get stretched, and vice versa. In general, when k gets larger, it will take less time for the EA-CSD to figure out the final convergent priority order (order that does not change from a certain point on) of tasks based on the energy consumption and criticality of tasks in the system. This is because tasks are ordered in SP queue and Best-Effort queue based on periods. If periods get stretched to a larger extent, it takes a fewer number of updates for low-energy and more critical tasks to get to the front line of EA-CSD to be scheduled more often than the high-energy and less critical tasks. Usually this implies a more substantial battery life improvement as k is increased. Since the increase in total amount of COP cannot keep pace with the rapidly improving battery life, there is a corresponding degradation of performance as k gets larger. These two phenomena are demonstrated in Figure 8(a), where the simulation is performed with r = 0.9, f = 10, F_t = 0.1.

In Figure 8(a), battery life lengthens and performance declines as k continues to increase from 0 to 90. From 90 to 100, the curves tail off and wind back in the reverse direction since each task is assocated with an upper-bound period length P_{max} as described in Section 2.2.7. Further increasing k does not effectively change the task periods as most long-period tasks have reached their maximum periods already. Also, increasing k means tasks take fewer updates to converge to the final target priority order. Once this target convergent order has been reached, the value of k does not matter as much as it is before the order is reached. Convergent orders are reached with essentially the same number of updates for large k's. All these explain why the curves become flatter as k approaches a certain large threshold. Figure 8(b) shows the percentage improvement in the total amount of COP earned during the power-on period as k varies. It can be seen that it follows a similar trend as battery life improvement for the same reasons.

4 Related Work

There have been substantial studies on power management for mobile applications. However,

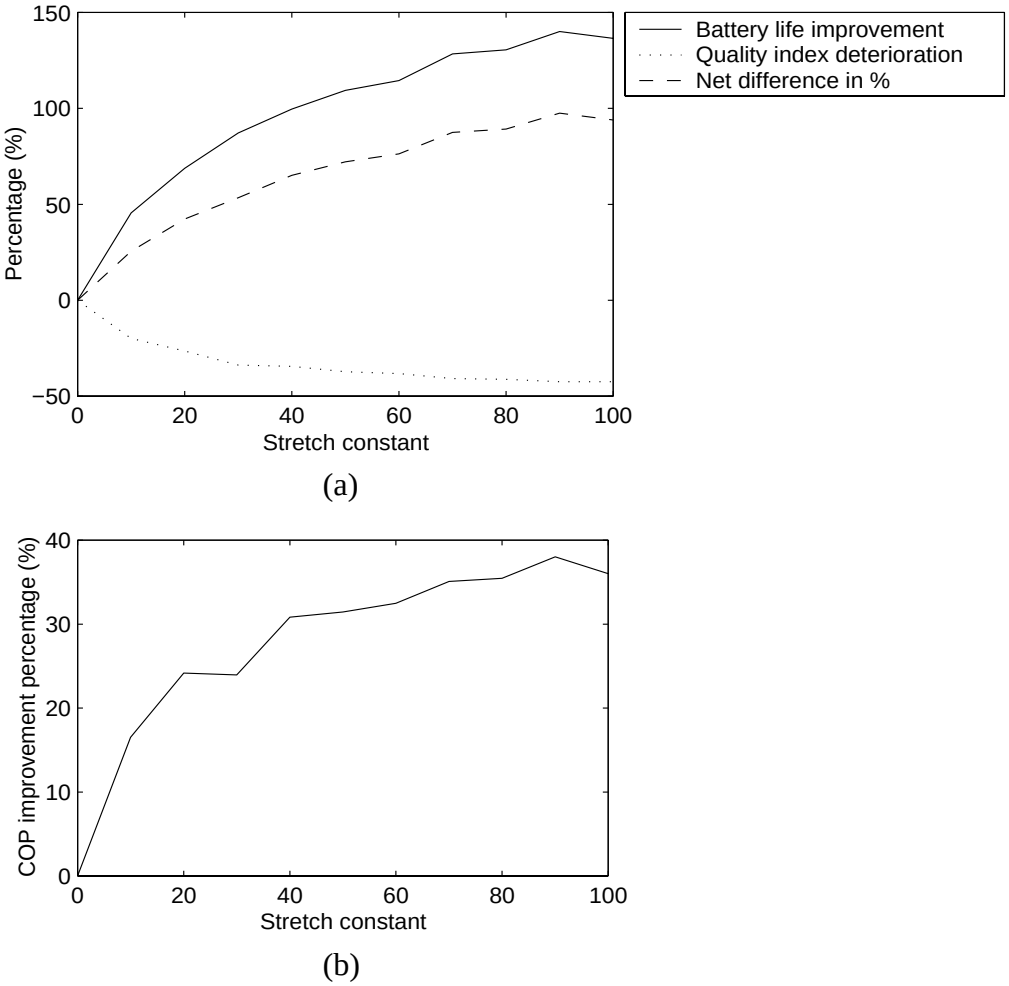

Figure 8. Battery life improvement, quality index deterioration and total COP improvement with varying stretch constant *k* for EA-CSD. Note that the curves tail off and even rebound back as *k* gets larger due to the upper bound imposed on task period length.

unlike EA-CSD, most of these do not consider individual tasks' characteristics and therefore rarely consider the relative importance of energy consumption and criticality of tasks.

Some researchers focus on low-power architectural designs in processor circuits in saving energy [1, 6]. The need for hardware-level energy savings is reiterated by Lorch *et al.* [9] with the analysis of power-saving features in Apple Macintosh computer. They advocate the design of lower-power versions of hardware components including the CPU, hard drive and display. Most of these hardware energy-conservation methods are implemented system-wide regardless of the varying application task requirements and specifications. Other researchers analyze the reduction in energy consumption by powering down unused components [2, 8, 11] or switching the system into sleeping mode in predicted idle periods [7, 12]. However, the former saves energy only on a few specific power-hungry hardware components while the latter relies too much on predictions in order to decide the mode of operation, which decreases the system's predictability and reliability in handling hard real-time tasks.

Weiser *et al.* [13] proposes dynamically adjusting clock speed to save energy resources, primarily by allowing the system to use a lower voltage level at a slower clock speed which does not jeopardize the system's timeliness in meeting task deadlines. In contrast to tasks scheduled by EA-CSD, applications are basically equally treated in their shares of energy resources and the system hardware takes up the responsibility of saving system-wide energy resources by reducing clock speed, which is decided and controlled by OS. There are disadvantages for this approach in that it is relatively difficult to strike a compromise between energy savings and interactive responses of the applications with the selection of clock speed adjustment interval. Critical tasks may easily miss deadlines if the response time is too long due to coarse-grained clock speed adjustment interval. Comparing EA-CSD to this approach, EA-CSD renders a much higher probability for critical tasks to meet deadlines.

Ellis [3] emphasizes the importance of developing an energy-saving API between individual applications and the system. Using the Hiker's Buddy running on the PalmPilot as an example, she explores how architectural designs can provide power management facilities that cooperate with software-based applications in reducing energy usage.

One of the more closely related work is Odyssey [5], which provides an example API introduced by Ellis that supports energy conservation over multiple concurrent applications. It is part of the operating system architecture that monitors dynamically the usage of energy by individual applications based on the total amount of energy available in the system. Energy usage is managed on the application level. Applications that exceed energy usage expectations are notified, which then adjust their fidelity to cope with the latest system energy resource level. The lowering in fidelity corresponds exactly to the performance degradation that has been addressed in this paper for EA-CSD. As expected, this is accompanied by a reduction in energy consumption, which, in turn, extends the battery life just like EA-CSD.

Another relevant work by Lorch *et al.* [10] involves delaying scheduling of tasks without useful activity, which is analogous to postponing the execution of less critical tasks by EA-CSD. Nonetheless, it makes scheduling decisions based on whether tasks are actively computing at run-time, not based on their intrinsic qualities such as the energy consumption and criticality.

Last but not least, the primary purpose of providing a highly user-customizable interface in designing EA-CSD receives solid support from [9] in its claim that an energy-saving strategy depends strongly on the user's environment and that user variability must be taken into account.

5 Problems Encountered and Future Work

Our EA-CSD simulator is intended to emulate closely the real OS environment in mobile applications including dispatching and queueing mechanisms, overhead predictions in blocking/unblocking as well as updating EADPs and the ensuing period stretching of tasks. Even though it is virtually impossible to have an exact clone of the real OS environment in the simulator, the general trend found from the simulation results should still serve as a reliable indication about the merits of the EA-CSD over the non-EA-CSD in terms of meeting the energy- and criticality-oriented objectives that have been discussed throughout the paper. In the future, we would like to simulate the running of the EA-CSD on EMERALDS RTOS with embedded microcontrollers

such as Motorola 68332, Intel i960 and Hitachi SH-2, which will provide the full and accurate set of RTOS conditions.

There seems to be a little deviation from what is expected in the simulation of varying the user-defined ratio and examining its effect on battery life improvement and performance degradation. More specifically, as one goes from $r = 0$ to 1 through the intermediate range, the energy consumption term in calculation of EADP still has lingering dominance, and high-energy, less critical tasks get their periods stretched more often and more extensively than are actually necessary. This creates quite a number of less energy-consuming idle periods during the running of EA-CSD, leading to low CPU utilization and the slightly upward slope of the battery life curves in Figure 4(a) in the intermediate range. To prevent this slightly upward slope from causing unpredictable application-level behaviors, the application can mask the intermediate range of r and allows only discrete-valued adjustment at either end of the range. This results in absolutely longer battery life when r is close to 0 and shorter battery life with more critical tasks meeting deadlines when it is close to 1.

Also, it is more desirable if we can dynamically control the parameters concerning period stretching such as the stretch constant and update factor in the middle of the scheduling session. An ideal EA-CSD would be one that is capable of analyzing the past history of task execution scheduled by EA-CSD and automatically adjusts the stretching parameters based on the history in order to obtain a higher utilization of the CPU while still fulfilling the users' objectives. A feedback mechanism in which more idle periods in the middle of scheduling dynamically lower the stretching parameters may also help. Shrinking periods appropriately for those tasks whose periods have been stretched before can also be one solution to the problem.

There remain some unresolved issues, including the fact that it is relatively difficult to determine the optimal values of the parameters. By setting the parameters to certain values, one can only expect that the parameters can improve the energy adaptability of the application and there is no guarantee that the parameters constitute the optimal set. Of course, trial and error can increase the optimality, but reaching the optimal point is quite difficult. EA-CSD does not target at optimality, but only improvement over non-EA-CSD.

Moreover, it is somewhat difficult for either developers or users to have a standardized way to determine COP for each task since the concept of criticality lacks a concrete way of measurement. One can only say whether a task is critical or not, but it is difficult to supply exact quantified values to classify tasks with intermediate levels of criticality. Inaccurate assignment of COP can result in undesirable consequences since real critical tasks may not be able to get executed in a timely fashion. A more systematic approach is needed to evaluate the COP for each task.

6 Conclusion

The EA-CSD in this paper is not aimed at providing an optimal algorithm for assigning task priority based on energy consumption and criticality of tasks. Rather, it represents a ''good'' approach that takes into account both the energy consumption and criticality of tasks in making scheduling decisions to suit the user-specified goals in conserving energy and favoring the execution of critical tasks. To our best knowledge, there are no known scheduler-based

approaches that deal with the problem of energy conservation in mobile applications, let alone the problem of criticality maximization of tasks executed. EA-CSD does not advocate system-wide energy conservation approaches which globally restrict energy usage of all tasks, indiscriminately treating all tasks as equally eligible for energy resources. Instead, EA-CSD reaches the very root of the problem and provides a built-in, automatic approach that assigns priorities based on intrinsic power and criticality characteristics of individual tasks to increase energy savings and level of criticality of tasks timely executed. User variability is realized when users have complete freedom and ease of choosing the level of performance desired in the above areas without constant manual monitoring and caring about internal details of implementation. EA-CSD is therefore a good starting point for further research and development in energy-adaptive scheduler.

References

[1] A. P. Chandrakasan, S. Sheng, R.W. Broderson, "Low power CMOS digital design", *IEEE Journal of Solid-State Circuits*, Vol. 27, No. 4, pp. 473-484, 1992.

[2] F. Douglis, P. Krishnan, B. Marsh, "Thwarting the power-hungry disk", In *Proceedings of 1994 Winter USENIX Conference*, January 1994.

[3] C. Ellis, "The case for higher-level power management", In *the 7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pp. 162-167, Rio Rico, AZ, March 1999.

[4] J. Flinn, M. Satyanarayanan, "PowerScope: a tool for profiling the energy usage of mobile applications", In *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications*, pp. 2-10, New Orleans, LA, February 1999.

[5] J. Flinn, M. Satyanarayanan, "Energy-aware adaptation for mobile applications", In *Proceedings of 17th ACM Symposium on Operating Systems Principles*, Dec 1999.

[6] M. Horowitz, "Self-clocked structures for low power systems", *ARPA semi-annual report*, Computer Systems Laboratory, Stanford University, December 1993.

[7] C.-H. Hwang, A. C.-H. Wu, "A predictive system shutdown method for energy saving of event-driven computation", *IEEE/ACM International Conference on Computer-Aided Design*, pp. 28-32, 1997.

[8] R. Kravets and P. Krishnan, "Power management techniques for mobile communication", In *Proceedings of The Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM'98)*, October 1998.

[9] J. R. Lorch, A. J. Smith, "Apple Macintosh's energy consumption", *IEEE Micro*, 18(6): 54-63, November/December 1998.

[10] J. R. Lorch, A. J. Smith, "Scheduling techniques for reducing processor energy use in MacOS", *Wireless Networks*, Vol. 3, pp. 311-324, 1997.

[11] M. Stem, R. H. Katz, “Measuring and reducing energy consumption of network interfaces in hand-held devices”, *IEICE Transactions on Communications, Special Issue on Mobile Computing*, 80(8), August 1997.

[12] M. Srivastava, A. P. Chandrakasan, R. W. Broderson, “Predictive system shutdown and other architectural techniques for energy efficient programmable computation”, *IEEE Transactions on VLSI Systems*, Vol. 4, No. 1, pp. 42-55, 1996.

[13] M. Weiser, B. Welch, A. Demers, S. Shenker, “Scheduling for reduced CPU energy”, In *Proceedings of First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 13-23, Monterey, CA, November 1994.

[14] K. M. Zuberi, P. Pillai, K. G. Shin, “EMERALDS: a small-memory real-time microkernel”, In *Proceedings of 17th ACM Symposium on Operating Systems Principles, Published as Operating Systems Review*, 34(5): 277-291, Dec 1999.

[15] K. M. Zuberi, “Real-time operating system services for networked embedded systems”, PhD Thesis, University of Michigan, 1998.