Energy-Aware Quality of Service Adaptation [*]

Padmanabhan Pillai, Hai Huang, and Kang G. Shin
The University of Michigan
{pillai, haih, kgshin}@eecs.umich.edu

Abstract

In a wide variety of embedded control applications, it is often the energy resources that form the fundamental limits on the system, not the system's computing capacity. Various techniques have been developed to improve energy efficiency in hardware, such as Dynamic Voltage Scaling (DVS), effectively extending the battery life of these systems. However, a comprehensive mechanism of task adaptation is needed in order to make the best use of the available energy resources, even in the presence of DVS or other power-reducing mechanisms. Further complicating this are the strict timeliness guarantees required by real-time applications commonly found in embedded systems.

This paper develops a new framework called *Energy-aware Quality of Service* (EQoS) that can manage real-time tasks and adapt their execution to maximize the benefits of their computation for a limited energy budget. The concept of an adaptive real-time task and the notion of utility, a measure of the benefit or value gained from their execution, are introduced. Optimal algorithms and heuristics are developed to maximize the utility of the system for a desired system runtime and a given energy budget, and then extended to optimize utility without regard to runtime. We demonstrate the effects of DVS on this system and how EQoS in conjunction with DVS can provide significant gains in utility for fixed energy budgets. Finally, we evaluate this framework through both simulations and experimentation on a working implementation.

1 Introduction

With ever-improving semiconductor and architectural technologies, microprocessors have been improving in performance at an exponential rate. This rapid improvement in performance comes at a cost, in terms of system complexity and power dissipation. Although the move to finer-width fabrication technology and lower voltage devices allows lower-power circuits, the rate of increase in complexity, speed, and size of microprocessors has resulted in increasingly power-hungry devices. In contrast, battery and energy storage technologies have been improving at a much slower pace, and as a result, are falling further behind in relation to the energy demands of newer processors.

Energy management has, therefore, become a critical issue in all portable and mobile computing platforms. In embedded systems used for control or communications, this is of even greater importance, as it is often impos-

[*]The work reported in this paper is supported in part by the U.S. Airforce Office of Scientific Research under Grant AFOSR F49620-01-1-0120.

Screen	CPU subsystem	Disk	Power
On	Idle	Spinning	13.5 W
On	Idle	Standby	13.0 W
Off	Idle	Standby	7.1 W
Off	Peak Load	Standby	27.3 W

Table 1: Power dissipation of laptop (HP N3350) components.

sible to increase energy storage capacity in such systems, and the consequence of running out of energy can be catastrophic, rather than merely inconvenient as in consumer devices.

Several approaches to energy management have been proposed and attempted. Most consumer computing platforms implement either Advanced Power Management (APM) or Advanced Configuration and Power Interface (ACPI) to manage the energy consumed by the system. Although implemented very differently, both of these are used in general-purpose platforms to power down certain hardware devices, or place them in low-power modes when not used for some period of time. These techniques work well in office computers that are powered on all night, or laptops that can shut off their modems or network interface cards when not used, and suspend the session to disk when battery is low.

Although the ability to put idle devices into low-power, standby modes is undoubtedly useful, there are limits to such techniques in embedded control applications, where the system is essentially always in an active state. In such cases, we need techniques that can operate at fine granularities to conserve energy while the system and devices operate. As the processor is often the single largest consumer of energy in most small systems (e.g., PDAs and palmtops), these mechanisms tend to focus on reducing the power of a running microprocessor. Table 1 shows measured energy consumption on a modern laptop, illustrating the dominance of CPU on power dissipation. One very simple and effective mechanism requires a *halt* instruction on the processor that stops the execution core until some interrupt triggers resumption. When this instruction is encountered, the CPU is effectively shut off (from an energy standpoint). Using this in an idle task can conserve almost all of the energy that would otherwise be dissipated executing idle loops. Ideally, with this mechanism, only cycles spent performing useful work will consume energy on the processor.

More advanced techniques can further reduce energy consumed even for the non-idle cycles. *Dynamic Voltage Scaling* (DVS) [38] reduces the frequency of the processor until it is just fast enough to complete the useful work (i.e., eliminate idle cycles). Since the speed at which the circuits operate is directly related to the voltage applied, it is possible to reduce the voltage when the frequency is reduced. As energy dissipated per cycle varies quadratically with voltage, DVS can potentially conserve significant amounts of energy, provided the appropriate voltage- and frequency- varying hardware is available. Such DVS hardware is now incorporated in the latest microprocessors [1, 9, 10, 37]. Assuming that timing guarantees can be preserved, embedded real-time systems can particularly benefit from both CPU-halt and DVS, as the systems are designed around worst-case execution times (WCETs) of tasks, which are often much larger than the average case, resulting in significant waste of energy for executed idle loops.

All of these process, battery, and DVS technologies, together can be seen as simply increasing the total number of cycles of computation a particular-sized battery-operated device can perform. However, these provide no guidelines as to how the scarce energy may be best allocated to perform useful computation. The goal of this paper is to introduce the new concept of *Energy-aware Quality of Service* (EQoS), a framework that can be used to maximize the total value gained from performing computation in an energy-restricted environment. In particular, we formulate energy adaptation of task sets into a tractable, optimal-selection problem, and develop solutions that, in conjunction with CPU-halt and DVS mechanisms, meet system runtime goals while maximizing the value gained from the execution of tasks.

In the next section, we describe the overall EQoS framework, and then introduce various optimal algorithms and heuristics for energy and task adaptation. Following this, we describe our implementation of EQoS, and present results from detailed simulations and actual (experimental) measurements. After relating this work with existing literature, we conclude and highlight some future research directions.

2 Energy-Aware Quality of Service

In order to improve energy usage and maximize the benefits of computation, we introduce a comprehensive Energy-Aware Quality of Service (EQoS) framework to regulate the consumption of scarce energy resources. In particular, the EQoS framework will allow the dynamic allocation and reclamation of energy resources from the various applications running on an embedded device. The key novel aspects of the EQoS framework are that it:

1. Brings together various technologies and techniques (including some not intended for energy conservation) to make best use of limited stored energy;
2. Varies the service level provided to each task to meet system runtime goals and maximize the total value of computation; and,
3. Formulates this comprehensive energy adaptation into a tractable, optimal-selection problem.

This entails adapting tasks to the energy available and executing them at various service quality levels, which in turn incur varying power dissipation rates. The framework to implement this comprehensive adaptation consists of components spanning from the hardware level up to the application level, as summarized in Figure 1. The most important technology components integrated into the EQoS framework are:

1. Low-level mechanisms to reclaim energy and reduce idle-time waste.
2. Methods of executing real-time tasks at varying quality of service and energy levels.
3. Methods of specifying task energy requirements and adaptation limits.
4. Algorithms to maximize benefits of system execution for limited energy.

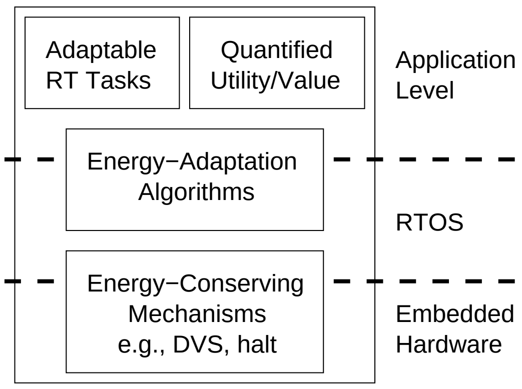

Figure 1: EQoS framework overview.

The individual components of our EQoS framework are detailed below, with discussions on constraints and restrictions necessary to keep the adaptation of the real-time task set to maximize benefits of computation a tractable optimization problem.

2.1 Energy-Conserving Mechanisms

When employed aggressively, low-level mechanisms to conserve energy in the hardware can significantly reduce CPU power dissipation. The techniques mentioned earlier work well and are incorporated in our EQoS framework. Both the processor halt and the DVS mechanisms eliminate processor energy costs associated with idle time. DVS goes further and reduces the energy overheads of over-engineering computation capacity, by reducing per-cycle energy costs as frequency and voltage are reduced to match the actual processing load. Hence, when computational resources are not fully utilized, these mechanisms can reclaim the energy that would otherwise be wasted on idle processor cycles. One goal of EQoS adaptation is to reduce the computational load of the system to allow the low-level mechanisms to kick in, provide large energy savings, and meet system runtime requirements.

However, there are several critical problems in implementing such schemes in a real-time embedded system. In particular, varying frequency influences the execution time of the tasks in the system. As real-time systems must provide strict timeliness guarantees, changes in CPU frequency can result in timing constraint violations and deadline misses if not carefully implemented. Although many DVS mechanisms exist for general-purpose systems, very few are available for real-time systems. Recent research [8, 26] has produced some practical real-time DVS (RT-DVS) mechanisms. The EQoS framework uses RT-DVS techniques that defer task execution, reclaim unused processing cycles online, and aggressively reduce processor frequency and voltage. Assuming the task set is schedulable, the algorithms take real-time characteristics into account and ensure all deadlines are met while reducing energy expenditure.

However, the actual energy benefits of DVS algorithms depend entirely on the actual execution times of the tasks in the system, which in general are not predictable. This can cause a significant error in any adaptation that tries to achieve a system runtime requirement. Our EQoS framework deals with this problem by introducing the concept of an idealized DVS response, and using this to compensate for the effects of DVS. This is discussed in detail in Section 3.4.

2.2 Varying QoS for Real-Time Tasks

The low-level energy-conserving mechanisms are effective at converting any surplus computing capacity into significant energy savings. The goal of our EQoS mechanism is to maximize their effectiveness through selectively reducing load in the system by adapting the real-time task set to the available energy. When the system is energy-constrained, if we can reduce the load on the system, then RT-DVS mechanisms can kick in to extend system runtime and improve per-cycle energy consumption. However, this reduction in load must be done in a controlled manner, and cannot be arbitrarily applied when real-time and mission constraints are involved.

We could improve system runtime and make better use of scarce energy if we could limit the energy consumed by less important auxiliary tasks, and instead divert it to the execution of the more critical ones. This is, in a sense, a method of adapting the task set to the available energy resources and providing varying QoS to the tasks to maximize the return on the energy used for their execution. In order to use such a QoS mechanism, we need tasks that are amenable to adaptation and can be executed and different QoS levels with varying energy consumption rates. Application-level energy adaptation for multimedia has been well researched in the literature [6], but as applied to real-time tasks, it is under-explored.

However, in the field of fault-tolerant computing, one commonly deals with real-time tasks that have multiple operating modes. Particularly for large, multiprocessor and distributed real-time systems, methods of executing tasks at reduced service levels requiring less computational resources have been well studied. The idea behind this is that in case of failure, when parts of the system stop working and less resources are available, the system can continue to operate, using alternate task execution modes to ensure that all tasks are still able to run, although at a degraded quality level. This "graceful degradation" is greatly preferred over a system that simply stops working. Of course, any degraded task that requires less computation time also requires less energy to perform those computations. An important component in our EQoS framework, therefore, is to use real-time task adaptation developed for fault-tolerance in the context of energy savings.

To understand how real-time task adaptation is implemented, it is helpful to look at the characteristics of typical real-time (RT) tasks and systems. The canonical model of a RT system includes multiple tasks that are executed periodically with a strict guarantee of their completion by a certain deadline. More specifically, each task i has an associated period, t_i. Every t_i time units, the task is started or released (or invoked). The task also has an associated worst-case execution time (WCET), C_i, and a relative deadline, d_i, measured relative to its release time. As these parameters are necessary for proper real-time scheduling, they are generally explicitly specified for RT tasks. The real-time scheduler guarantees that the task will receive up to C_i units of execution time within d_i time units of each release of this task. The actual execution time for each invocation of a task can vary greatly from the specified WCET, but as long as the tasks use less than their WCETs for each invocation, a provable guarantee of completion within deadlines is provided. Note that in many RT systems, including the classical RT scheduling algorithms [15], it is often assumed that the relative deadline equals the period ($d_i = t_i$, i.e., a task is guaranteed to complete its execution by the time its next invocation is released). Furthermore, this periodic model is used extensively in practice for a wide range of real-time applications, including various embedded control tasks, but is also quite general and has been used to accommodate other types of tasks, including sporadic and aperiodic event handlers [14].

The precise mechanisms employed in executing real-time tasks at degraded service levels depend on the nature of the applications. For control systems, one simple technique employed is to stretch the period of the real-time control task so the frequency of execution is reduced. The control task normally runs at a rate that has been deemed optimal for the system being controlled, providing the desired tradeoff between response time and overshoot limitation, while guaranteeing stability. When processing capacity is reduced due to failures, or we need to restrict computation to conserve dwindling energy resources, the process can be run at a lower periodic frequency, resulting in suboptimal, degraded control. Although performance becomes suboptimal, this may be desirable in order to keep the system operating longer to avoid catastrophes. Of course, this approach has its own limits, as stretching period too much may result in instability, but this is very dependent on the nature of the control system and can be usually determined at the the time the system is constructed.

For other applications, different types of degradation techniques also exist. Imprecise computation [16] models trade off execution time for precision. An example of this is iterative computations — the more iterations performed, the greater the resulting precision. In such tasks, it is possible to reduce the precision and the computational load, and therefore conserve energy resources. Yet other systems may use several completely different algorithms to perform similar tasks. For example, in the case of voice compression, two different CODECs employing different algorithms may produce the same level of compression, but at differing levels of loss or noise, and inversely correlated differences in computing time. In other cases, for a non-critical task, the most degraded service level may simply be not running it at all. Stopping/deleting the unimportant tasks can allow mission-critical tasks to run for a longer period of time when energy is low.

Our intention is not to devise new methods of providing degraded service-levels to real-time tasks, nor enumerate the existing techniques. Rather, we want to take the existing techniques that provide graceful degradation in the case of failures, and apply them to reduce energy consumption within a real-time EQoS framework.

However, we do need to restrict the set of tasks and their service levels to a certain extent. For any set of real-time tasks, one must test whether the particular combination of task requirements is *schedulable*, i.e., all deadlines can be met under a particular scheduling paradigm. In the EQoS framework, each task has multiple service levels, and each service level may have vastly differing real-time requirements, particularly if algorithmic change is involved in the service degradation. Hence, each combination of task service levels needs to be checked and only the schedulable combinations are valid outputs of any adaptation algorithm. This, in the worst case, requires an exponential search through all combinations of task service levels.

To simplify the schedulability problem, we first restrict scheduling to use the *earliest-deadline-first* (EDF) dynamic priority scheduler. This scheduler has the nice property that, assuming negligible scheduling / preemption overheads and independent tasks, all tasks are guaranteed to meet their deadlines as long as the total processor utilization, $\sum C_i/t_i$ over all tasks i, is no greater than one [15]. In the EQoS framework, where each task has multiple service levels, we define u_i^{max} as the largest C/t among all of the service levels of task i. Using this, we state a sufficient condition for schedulability:

$$\sum_{i=1}^{n} u_i^{max} \leq 1, \tag{1}$$

where there are n tasks in the system. By restricting task sets to only those that meet this sufficiency condition, we can guarantee that regardless of which service level is selected for each task, the system will always meet the EDF scheduling requirements. With these restrictions, the additional complexity of testing for schedulability is removed, eliminating a potential obstacle to efficient adaptation solutions. On most systems, these restrictions are met by default as schedulability conditions are usually satisfied when all tasks are executing in their highest service level.

2.3 Specifying Task Utility

Given a set of real-time tasks that can be executed at degraded QoS levels, it is fairly straightforward to enumerate a valid set of degraded operating modes for each task that will ensure mission-critical goals (e.g., maintaining stability) are met. When determining other characteristics, such as execution times, it is easy to measure energy consumption corresponding to these reduced QoS levels, which can be used in QoS level selection. As real-time tasks are already well-specified with respect to WCET, period, and deadlines, we simply need to specify one additional parameter, the average power dissipation for executing this task, for each QoS level defined for the task. However, to determine the best tradeoff between these task QoS levels, one critical input is needed — a quantification of the value or benefit gained, called *utility*, from running a task at a particular QoS level.

In general, utility is an abstract notion of value or gain, and need not be based on real units. This leaves the actual specification flexible to the type of applications or systems that are designed. Each QoS level for a task is assigned a utility value, reflecting the relative benefits executing different tasks at the various QoS levels defined. Similar notions of a generic utility metric have been successfully used elsewhere [12] for QoS-related optimizations.

With some forms of task QoS degradation, it is relatively simple to assign utility values. With the class of imprecise computation methods that provides "increasing rewards for increasing service" (IRIS) [2], a simple monotonically-increasing function of maximum computation time suffices for utility. Depending on the application, this may be a linear function or a fractional power relationship reflecting decreasing marginal returns.

For real-time control tasks that allow period extension as a graceful degradation method, a mechanism exists for quantifying the effects on the controlled system. Using a control-theoretic measure of *performance index* (PI), one can derive a reward function for various task-periods [33]. In this context, we can generally just use this reward function, scaled by some constant, to find the utility for the various task QoS levels.

For tasks that deal with multimedia, or voice compression, the utility assignment is somewhat trickier. In particular, the output of these applications can only be evaluated in terms of human-perceived quality, which is difficult to quantify. Running a task with reduced computation and energy resources may produce results indistinguishable from the original service level to one user, but seem to incur severe quality loss to a different user. Much research has focused on modeling average human perception of quality in the context of compression, and although such metrics may be used to assign utility, the utility assignment for multimedia tasks remains a difficult and somewhat arbitrary process.

Although in utility assignment there are no particular units of utility or ranges of acceptable values, the assigned utility values need to be consistent among the different tasks. This utility method allows for a wide range of possible task set adaptations when utility is maximized. It is possible to have some tasks run at their maximal (i.e., maximum computational and energy demand) QoS levels, while simultaneously others are at their minimum specified levels. On the other hand, the utility-based specification does have its limits. In particular, in this approach, the utility of running a task is independent of the tasks in the system, so it is not possible to specify constraints such as task A has utility x only if task B is running, or at least two of three tasks A, B, and C must run. However, for most systems, the notion of utility provides sufficient flexibility to define a wide range of preferences or relative benefits of task adaptation.

2.4 Maximizing System Utility

The final component of the EQoS framework is the algorithm used to actually select the QoS levels of tasks to be run. In order to obtain the greatest benefits from the limited energy sources, we need to select the QoS levels such that the utility is maximized. Even though all of the task QoS levels, their power requirements, and the utility gained through their executions are specified, there still remains a question of whether it is better to run tasks at minimal levels for a long duration, or execute for a shorter duration at high-utility, high QoS levels, or even a mixture of service levels so the system can run for a specified amount of time. The problem of energy adaptation is somewhat ambiguous, and the optimal solution depends on both the task set and the actual constraints on the system. In the following section, we present in detail the problem of energy-adaptive task QoS-level selection and describe algorithms to maximize utility in an energy-constrained adaptive system.

3 Adaptation Goals, Problems, and Algorithms

The goal of EQoS adaptation is to maximize the system utility or value for the given, limited energy resources, subject to mission constraints. By selectively reducing the QoS level provided to individual tasks, the total system load can be reduced, allowing low-level mechanisms to reclaim energy that can be used to run other tasks at high QoS levels, or keep the system functioning for a longer duration. The selection of task QoS levels can be expressed as a constrained utility maximization problem, but the actual algorithms to solve such problems depend on the constraints of the system. In this section, we first formally describe an adaptive system and formulate the selection of adaptation levels as a tractable, utility maximization problem. We then present several algorithms to optimally select QoS levels, as well as some simple heuristics, and then extend these to solve related problems with different mission constraints.

3.1 Adaptive System Description

An adaptive task set is formally described as follows. We are given a set of n tasks, $T_1, \ldots, T_n$, of which each task T_i has m_i different QoS levels defined as $T_{i,1}, \ldots, T_{i,m_i}$. For each level $T_{i,j}$, we specify its canonical real-time parameters, period $t_{i,j}$ and WCET $C_{i,j}$, and we also specify a utility value $U_{i,j}$ and an average energy expenditure $E_{i,j}$. Both of these are expressed as per-invocation values. The energy parameter can be measured easily by running the task on the target hardware platform and using standard electronic instruments (e.g., DVM or oscilloscope with current probe).

Given a particular selection of QoS levels, $j_1, \ldots, j_n$, for tasks $T_1, \ldots, T_n$, respectively, the number of iterations executed for task i over a time interval t is in the range, $\left[\left\lfloor \frac{t}{t_i} \right\rfloor, \left\lceil \frac{t}{t_i} \right\rceil\right]$. Assume further that the system dissipates P_{fixed} power when idle, and that the $E_{i,j}$ values indicates the average additional energy consumed for each invocation of task i at service level j, beyond what would have been used for idle. Now, by multiplying the number of iterations by the average energy per invocation for each task and adding these to the fixed energy consumed, we can determine the energy consumed over interval t. Dividing this by t, we obtain a range for the average power, P_{sys} over the interval t:

$$\frac{1}{t}\left(t \cdot P_{fixed} + \sum_{i=1}^{n} \left\lfloor \frac{t}{t_i} \right\rfloor E_{i,j}\right) \leq P_{sys} \leq \frac{1}{t}\left(t \cdot P_{fixed} + \sum_{i=1}^{n} \left\lceil \frac{t}{t_i} \right\rceil E_{i,j}\right)$$

As system runtimes are in the range of minutes to hours, while task periods are on the order of tens of milliseconds, we can safely assume long observation intervals relative to task periods. For large values of t/t_i, P_{sys} converges to:

$$P_{sys} = P_{fixed} + \sum_{i=1}^{n} \frac{E_{i,j_i}}{t_{i,j_i}}. \tag{2}$$

Similarly, we can determine the rate of utility gain as:

$$\sum_{i=1}^{n} \frac{U_{i,j_i}}{t_{i,j_i}}.$$

Assuming that we have E_{sys} energy units initially available, the total expected system runtime is expressed as E_{sys}/P_{sys}. Multiplying this runtime by the utility gain rate, we can determine the total system utility, U_{sys}, as:

$$U_{sys} = \frac{E_{sys} \sum_{i=1}^{n} \frac{U_{i,j_i}}{t_{i,j_i}}}{P_{fixed} + \sum_{i=1}^{n} \frac{E_{i,j_i}}{t_{i,j_i}}}. \tag{3}$$

The goal of any EQoS adaptation algorithm is to select the per-task QoS levels, indicated by $j_1, \ldots, j_n$, to maximize U_{sys} subject to system runtime constraints. The actual algorithm depends heavily on these other constraints, so we need to consider each specific type of problem separately in the following sections. In addition, we note that the $E_{i,j}$ values indicate average energy consumption assuming that the processor operates at its maximum frequency and voltage. Employing DVS will reduce the actual power dissipated, and we will explore ways of compensating for its effects in Section 3.4.

3.2 Known Time-to-Charge Problem

We first look at task adaptation for cases where the system will need to operate on stored energy for a finite time, or the system runtime is bounded by a known value. This is a common scenario, where one knows when primary energy sources will become available to power the system and recharge batteries. An example of this is a solar-powered satellite that has entered the shadow of the planet — given the orbital mechanics, the required time until it emerges out of the shadow can be computed very accurately. Stored energy must be used during this interval, and we would like to adapt the task set to maximize the utility of the system during this interval. We note that in such a system, due to size and weight restrictions, the gradual deterioration of the batteries, software upgrades that change task sets, variations in shadow transition times, and physical inaccessibility, a dynamic system of adaptation is preferable to static techniques or the over-engineering/replacement of the batteries.

In this scenario, we are given the system energy, E_{sys}, and a time, t_{run}. This time can be thought of as the required system runtime or the time until the next recharge for the system. We want to maximize the utility gained during t_{run}, but do not care about execution beyond this time. The reason is that after this time, primary energy sources are available, so there are no longer energy constraints and the system can simply operate using the maximal task QoS levels. Since we are concerned only with utility during this time interval, total system utility simplifies to:

$$U_{sys} = t_{run} \cdot \sum_{i=1}^{n} U_{i,j_i}/t_{i,j_i},$$

assuming that the actual system runtime is at least t_{run}:

$$\begin{aligned} \frac{E_{sys}}{P_{sys}} = \frac{E_{sys}}{P_{fixed} + \sum_{i=1}^{n} \frac{E_{i,j_i}}{t_{i,j_i}}} &\geq t_{run} \\ \sum_{i=1}^{n} \frac{E_{i,j_i}}{t_{i,j_i}} &\leq \frac{E_{sys}}{t_{run}} - P_{fixed} = P_{budget} \end{aligned}$$

The right side of the inequality can be considered a power budget for task execution that ensures a system runtime of t_{run}. With these derivations, we now need to select per-task QoS levels $j_1, \ldots, j_n$ to maximize the total utility by maximizing the utility rate, $\sum_{i=1}^{n} U_{i,j_i}/t_{i,j_i}$, while ensuring that the system runs for the desired time t_{run} by constraining the total power for task execution, $\sum_{i=1}^{n} E_{i,j_i}/t_{i,j_i} \leq P_{budget}$.

Expressed in this way, the optimization problem reduces to the *multiple-choice knapsack problem*

(MCKP) [18], a lesser-known variant of the famous *0-1 knapsack problem*. In MCKP, we have a set of categories, each with a number of non-overlapping items, each of which, in turn, has an associated value and weight/size/cost. Given a knapsack limit, the goal is to select exactly one item from each category to maximize value subject to the knapsack size limit.

Our problem is expressed as an MCKP by treating each task as a category and its set of defined QoS levels as the items within the category. The knapsack size is set to the power budget, $K = E_{sys}/t_{run} - P_{fixed}$. The item values and weights are the utility rates and power dissipation of the tasks, respectively, at each quality level, i.e., $v_{i,j} = U_{i,j}/t_{i,j}$ and $w_{i,j} = E_{i,j}/t_{i,j}$. It now suffices to solve this MCKP to determine the optimal set of task QoS levels, $j_1, \ldots, j_n$, that maximizes the total expected utility of the system with E_{sys} energy units during the time t_{run} until the next recharge.

3.3 Solving MCKP

The naive approach to solving this MCKP optimally is a simple state-space search. We systematically iterate through every combination of task QoS level assignment, checking the total power against the power budget, and keeping track of the selection that results in the largest total utility rate. Unfortunately, the search space grows exponentially — if the n tasks have m quality levels each, the search takes $O(m^n)$ time. Therefore, this approach is generally not practical.

There is, however, no known general solution to MCKP that can be performed in polynomial time, since MCKP is an NP-hard problem for the following reason. We can express any 0-1 knapsack problem as an MCKP: for each item a_i in the former, we create an additional item a'_i with weight and value equal to zero. Then, each category consists of the item and its zero value counterpart, i.e., $\{a_i, a'_i\}$. The knapsack size is the same for both problems. If a_i is included in the MCKP solution, it is also in the 0-1 knapsack solution; if a'_i is included, then a_i is not in the 0-1 knapsack solution. With this mapping, any polynomial solution to MCKP can be used to get a polynomial-time solution to the NP-hard 0-1 knapsack problem. Hence, MCKP is NP-hard as well.

However, if we assume that the weights (i.e., power) can be expressed as integers, then we can obtain a pseudo-polynomial-time optimal solution using dynamic programming (DP) techniques. We first solve trivially the MCKP containing just one category (task) for all possible knapsack sizes (power budgets), which are also integers. Using the optimal solutions to this subproblem, we can find the solution to MCKP for the first two tasks in linear time for each possible power budget. Repeating this process of building on the previous partial solutions, we obtain the solution to MCKP with all n tasks, for all power budgets. This has a pseudo-polynomial-time complexity of $O(nmk)$, assuming there are n tasks with m QoS levels each, and a maximum power budget (knapsack size) of k. Unfortunately, as k can be large, the time required may be significant. Furthermore, DP requires significant amounts of memory as well — the space complexity is $O(nk)$. This latter may make it impractical for small embedded controllers, which are typically memory- as well as processing- and energy-constrained.

One final approach to the optimal solution is to use a branch-and-bound (BB) algorithm. This involves a depth-first traversal of the decision tree, but only promising branches are visited, greatly reducing the time relative

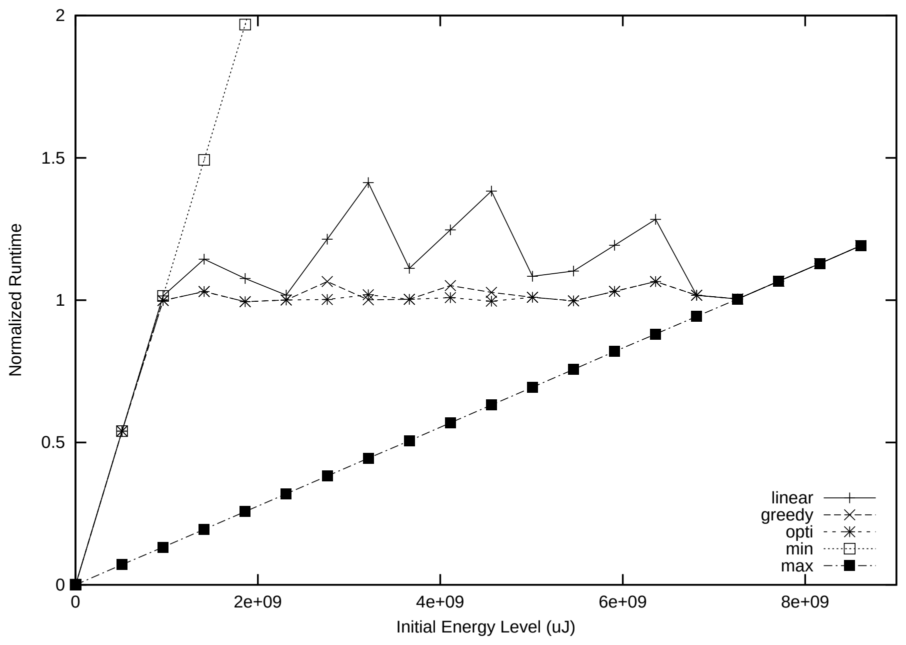

Figure 2: Example showing effects of adaptation on runtime, normalized to t_{run}.

to simple exponential search. Each level of the decision tree is associated with a task, and each node at level i has m_i branches, corresponding to the m_i QoS levels defined for T_i. A fast bounding algorithm is used to compute an upper bound to the total value that is possible if a particular branch is taken. Only branches with an upper bound on value greater than any seen so far are visited.

This technique requires a fast algorithm to compute a good upper bound on the value of MCKP. We use the linear relaxation of MCKP called linear MCKP (LMCKP) [27, 39], which can be solved quickly. LMCKP allows making fractional selections, which permits interpolation between two defined QoS levels of a task, e.g., 0.75 $T_{2,3}$ and 0.25 $T_{2,4}$, providing weighted average values for the power and utility rates. To solve LMCKP, we start with all tasks at minimal service levels. We enumerate all of the possible "upgrades," or changes in selection from a lower QoS level to a higher one, for each task, and then sort these for all tasks by the utility-change to power-change ratios. We apply the upgrades in order, the one with the highest ratio first. If applying an upgrade exceeds the power budget, we apply it proportionally to the available power, resulting in a fractional selection for one task, a fully-used budget, and task selections providing the greatest marginal utility. This LMCKP solution is optimal, and is guaranteed to always give a value no lower than the optimal discrete MCKP solution, so it can be used as a fast upper bound on MCKP.

The BB algorithm produces an optimal solution and is not affected by large knapsack size as with DP. The drawback is that there is no guarantee of effective pruning of the search space, so this may, in the worst case, require very long execution time as with an exponential search.

To overcome this drawback, we also consider a couple of simple heuristics that have very short execution times. The first heuristic uses the solution to LMCKP, which was used as the upper bound in the BB algorithm. We

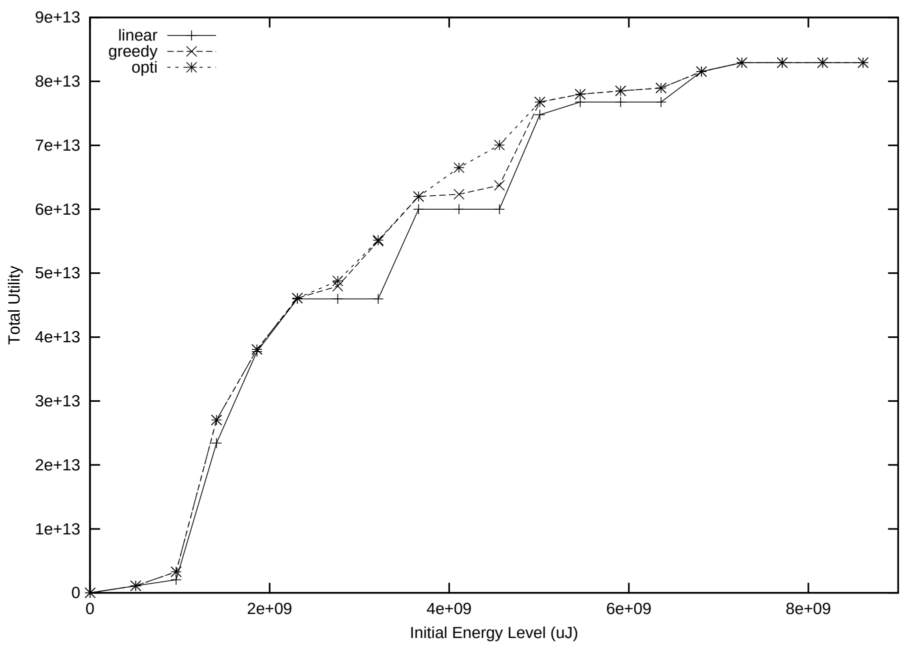

Figure 3: Example task set total utility gain comparison using adaptation.

simply drop any fractional selection, replacing it with the lower of the fractionally-selected QoS levels, to produce a valid, though not optimal, discrete solution.

A slightly better solution is achieved through a greedy algorithm. This starts out just like the linear approach, “upgrading” selections in order of the largest utility-change to power-change ratios. Rather than using a fractional selection to fill out the knapsack, the greedy heuristic continues to look through the sorted list, performing any possible QoS-level upgrade. This should result in a total utility no lower than the linear heuristic.

The execution times for these are very low, both incurring a linear complexity, in addition to any overhead of sorting the QoS-level upgrades. This sorting is needed only when the actual task set changes, so this overhead is not always incurred. Unfortunately, there is no guarantee on how close to optimal the resulting solutions are with these heuristics. It is possible to construct task sets, for which these heuristics produce arbitrarily poor solutions relative to the optimal algorithms. However, for most realistic task sets, the heuristics will produce reasonable solutions, albeit with some deviation from optimality.

To illustrate the effects of these different adaptation algorithms, we show in Figure 2 the resulting system runtime normalized to the desired runtime for a sample task set while varying initial energy, where “opti” refers to the optimal solution obtained from the DP and BB approaches, and “linear” and “greedy” refer to the heuristic solutions. As for comparison, we also include a non-adaptive use of the minimal and maximal QoS levels for the tasks, labeled as “min” and “max”, respectively. Adaptation produces selections between these extremes and maintains system runtime near the desired value of t_{run}. At the extremes of the initial energy range, however, we cannot adapt any further, and therefore overlap the minimal or maximal curve as initial energy is varied. The total utility until the known time-to-charge, t_{run}, for this example task set is plotted in Figure 3. We present the specifics

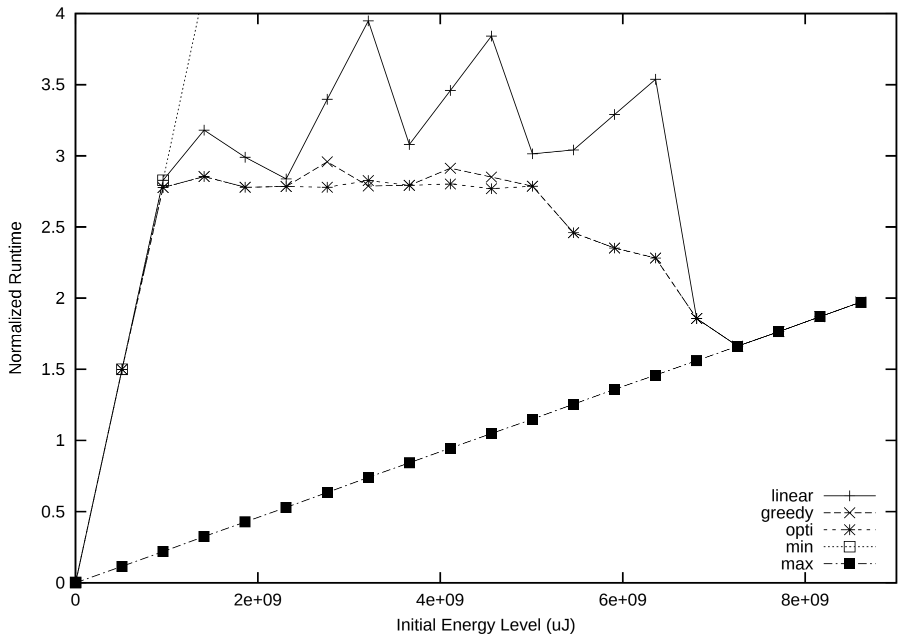

Figure 4: Example showing effects of DVS on runtime.

of our simulations and detailed evaluations of these algorithms in Section 5.

3.4 Effects of DVS

Thus far, we have not considered the effects of DVS on adaptation. In particular, DVS techniques allow for greatly reduced per-cycle energy costs when the processor is lightly-loaded. This translates into a greatly-increased runtime for the system. Figure 4 shows the system runtime normalized to desired runtime after adaptation of the example task set used in the previous figures, but in addition, an aggressive real-time DVS algorithm is employed. The system in this example has 3 voltage-frequency combinations shown in Table 2. Compared to the previous results, we see much longer system runtimes, particularly when the system has very low utilization.

Of course, the problem we are dealing with — maximizing utility until a known recharge time — does not directly benefit from the extended runtime. Instead, We would rather use the benefits of DVS to run tasks at higher-power, greater-utility QoS levels. Unfortunately, the exact effects of DVS depend on the actual execution times for each invocation of all tasks, which is a random component and cannot be predicted *a priori*. As a result, we cannot take DVS directly into account in the adaptation algorithms. Instead, we would like to compensate for the effects of DVS and make the system run to the desired time with higher utility by providing higher QoS to the tasks.

To do this, we introduce the concept of the *idealized DVS response*. Based on our prior work in DVS algorithms [26], we note that advanced DVS algorithms that aggressively reclaim unused slack time achieve energy performance close to an easily-computed lower bound. Figure 5 shows the relationship between average power and processor-capacity utilization for idealized DVS for the particular settings in Table 2. The idealized response

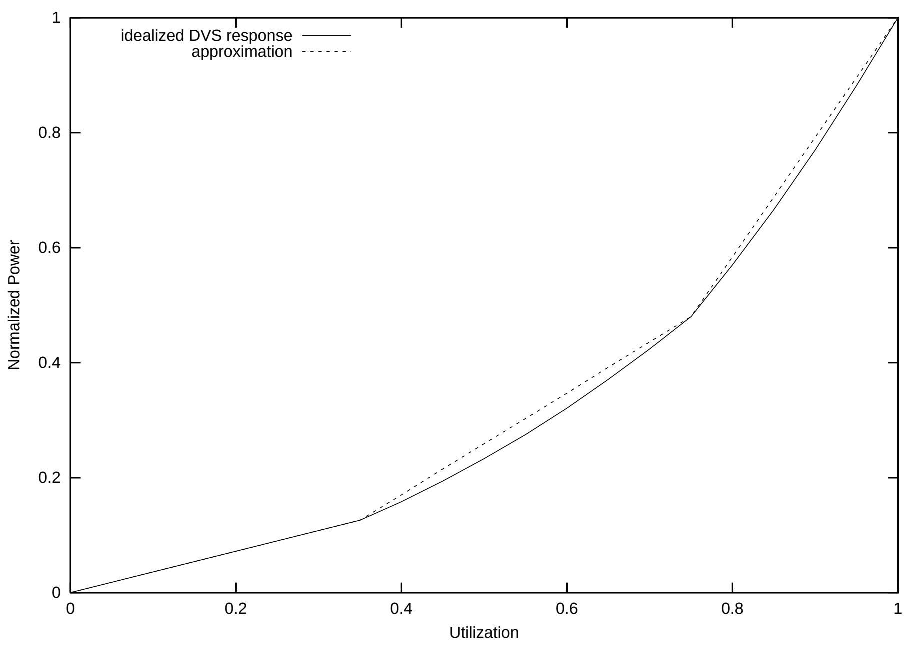

Figure 5: Relationship between utilization and power for ideal DVS.

Normalized frequency	Normalized voltage
1.0	1.0
0.75	0.8
0.35	0.6

Table 2: Normalized frequency and voltage settings for DVS.

reflects energy-per-cycle proportional to the voltage squared when the utilization equals the normalized frequency settings that are available. For utilization values between these, the average energy-per-cycle is a linear interpolation of these. Idle cycles are assumed to consume no energy (i.e., a perfect processor halt mechanisms), so the solid line in Figure 5 is obtained by multiplying the average (normalized) energy-per-cycle by the processor utilization, which equals the normalized number of non-idle cycles per second.

We can use the inverse relationship to determine the processor utilization that results in a particular normalized power budget. Call this inverse relationship the *compensation function*, or $F_{comp}()$. As the previously-discussed adaptation algorithms do not use utilization directly, we can convert the utilization value to a power value by multiplying it by $(P_{max} - P_{fixed})$, the maximum additional power dissipation at maximum system utilization, thus obtaining the power dissipation expected for the target processor utilization if DVS were not employed:

$$P_{comp} = (P_{max} - P_{fixed}) F_{comp}\left(\frac{P_{budget}}{P_{max} - P_{fixed}}\right).$$

We now feed P_{comp} as the power budget parameter to the adaptation algorithms. In other words, we use F_{comp}

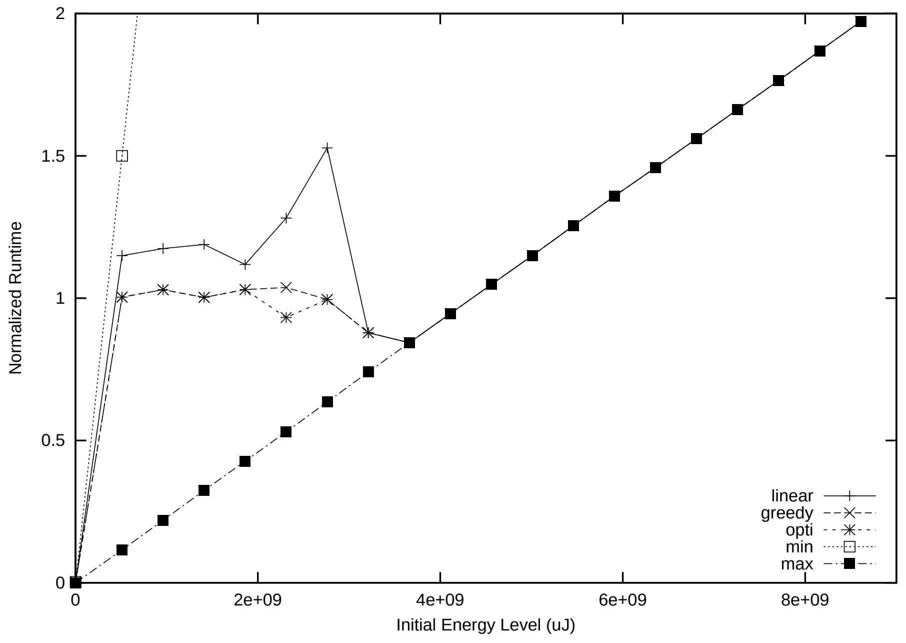

Figure 6: Example showing effects of DVS on runtime, normalized to t_{run}, when compensation is used.

to adjust the power budget to compensate for the effects of DVS. In fact, we simplify this a little bit by using a piecewise-linear approximation of F_{comp}, the inverse of the dotted line in Figure 5. By compensating for DVS in this way, we implicitly make the assumption that all power dissipated for executing a task is scaled when DVS is employed. This, of course, is not entirely accurate, since the energy for task execution, the $E_{i,j}$ terms, include consumption in buses, main memory, and I/O devices, in addition to the voltage-scaled CPU. However, since processor power generally dominates non-idle time energy expenditure, this approximation is generally acceptable, especially since F_{comp} is based on an approximation to an idealized response anyway. Figure 6 shows the same task set as above under adaptation and DVS, but with a compensated power budget. The system runtimes are now close to the desired runtime.

In practice, DVS mechanisms cannot actually provide the idealized response, as actual task execution times are random and cannot be expected to hold perfectly to their averages over short runs. Furthermore, some distributions and combinations of tasks simply do not work as well with DVS as others (e.g., due to on-off distribution of computation time). Due to this, combined with the fact we approximate all task energy as scaled by DVS, it is very likely that this method of adjusting the power budget for adaptation may over-compensate for the true effects of DVS. This is apparent in Figure 6, where the normalized runtimes dip slightly below 1.0 for the adaptation curves just before they merge with the maximum service curve. It is therefore best to use this compensation mechanism as a first approximation, then adapt the task set again at later times. We note that if the adaptation algorithm uses the DP approach, no additional work is really needed for subsequent adaptations, since DP computes optimal solutions for all possible power budgets with the given task set.

3.5 Applicability to Other Adaptation Problems

So far, we have considered the optimization of utility when we know the time until the primary power source becomes available. In addition to this scenario, we can also look at several other types of optimization problems.

First, one can consider maximizing system runtime. This is a very trivial problem — one simply needs to run each task at its minimal QoS level to maximize the runtime. As this is not a very interesting problem, we will not consider it any further.

A more interesting problem is the unconstrained maximization of utility, i.e., regardless of runtime. In this case, we need to deal with the unconstrained complex formulation of system utility presented in Equation 3. Despite this complexity, this adaptation problem is still tractable. In particular, we can solve this using the solution to the known time-to-charge problem discussed earlier. We use the adaptation algorithms to find optimal solutions for all possible power budgets, and then compute the resultant runtime and total utility for each power budget. We now simply select the power budget that gives the maximum expected utility. We note that with the DP approach, there is little additional work here, since the DP algorithm already finds optimal solutions for all possible power budgets (knapsack sizes) anyway. We simply have to determine the runtime and total utility for these. In addition, we can obtain a suboptimal approximation, possibly at a lower computational overhead, by evaluating the greedy heuristic for all possible power budgets.

A third variant is the maximization of utility subject to a required minimum runtime. Unlike the known time-to-charge problem, additional runtime beyond the requirement does contribute to the total benefits gained. The simplest, most direct solution to this problem is to implement the algorithm to solve the unconstrained utility maximization problem, but limit the final selection to the power budgets that provide at least the required system runtime.

3.6 Dealing with Dynamic Systems

Thus far, we have basically treated the task set as static. In general, real-time systems, especially in embedded systems, tend to have static or very infrequently changing task sets, so dealing with dynamic tasks is not a core concern of our adaptation. With the assumption of infrequent changes to the task set, the simplest method of dealing with dynamic tasks is to simply redo the energy adaptation on each task set change. As there are usually additional overheads associated with adding or deleting tasks (particularly when admission control is employed), this simply adds one additional computation during these changes.

With the branch-and-bound algorithm, we need to redo the entire search, incurring the full computational overhead on each task set change. In case of the DP approach, adding a task is relatively simple, since the existing solutions for n tasks are simply reused as the partial solutions for $n + 1$ tasks, and we incur only an $O(mk)$ overhead, where the new task has m QoS levels defined, and the maximum power budget is k. However, when deleting a task, all partial solutions may be invalidated, so the complete computation with $O(nmk)$ overhead may be needed. If the heuristics discussed earlier are used, the runtime overhead is too small to worry about, as long as an efficient sorted queue structure is used to keep the sorted QoS "upgrade" lists.

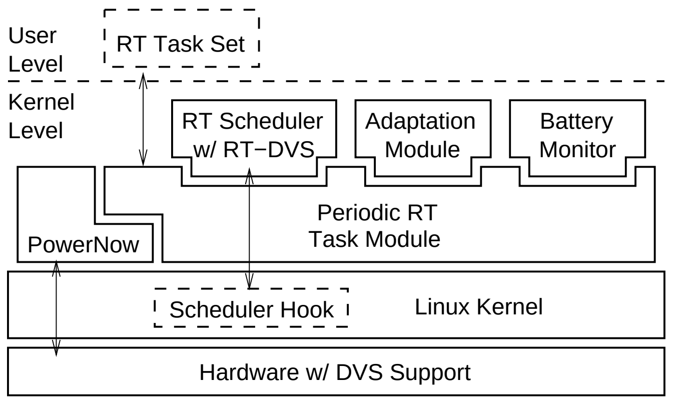

Figure 7: Software architecture of EQoS implementation for Linux.

We note that in addition to performing adaptation when the task set changes, it is best to perform these adaptations periodically. This will ensure that the errors introduced by inaccurate or imprecise task energy values, or the overcompensation for DVS effects are fixed and we can achieve close to the desired runtime with maximal utility.

4 Implementation

We have developed a working implementation of the proposed EQoS framework. It is built on top of an existing RT-DVS system [26], which provides a periodic real-time task model and support for DVS on top of the Linux operating system. DVS requires hardware support, and our implementation currently works on notebook computers with AMD Mobile Athlon, Duron, and K6-2 processors that implement AMD's PowerNow! voltage and frequency scaling support. Due to the changes introduced in Linux 2.4 kernels, the current implementation only works on the 2.2 kernels.

Figure 7 depicts the overall software architecture of the EQoS implementation. The proposed EQoS is implemented as modules that extend the Linux kernel. The core of the system is a module that provides the periodic real-time task model on top of Linux, interacts with tasks, and provides connection points to "plug in" various extensions. Various scheduler modules can plug in to this module, and implement one of several real-time scheduling policies, both with and without DVS support. The actual hardware control of frequency and voltage is done through a third module, the `PowerNow` module, which abstracts the specifics of the hardware.

Our EQoS adaptation algorithms extend this core RT-DVS through the `Adaptation` module. Although these adaptation mechanisms are ideally implemented as middleware, it was simplest for us to create another kernel module to perform this functionality. A fifth module, `batmon`, is intended to measure the available energy in the battery. It can be interfaced with energy monitoring mechanisms such as SmartBattery API [31]. For our implementation, we instead use it as a user interface to specify initial energy values. It also allows us to simulate controlled partial system power failures to see how the system adapts.

All task interactions are performed through the Linux `/proc` file system. A new RT task will open a special

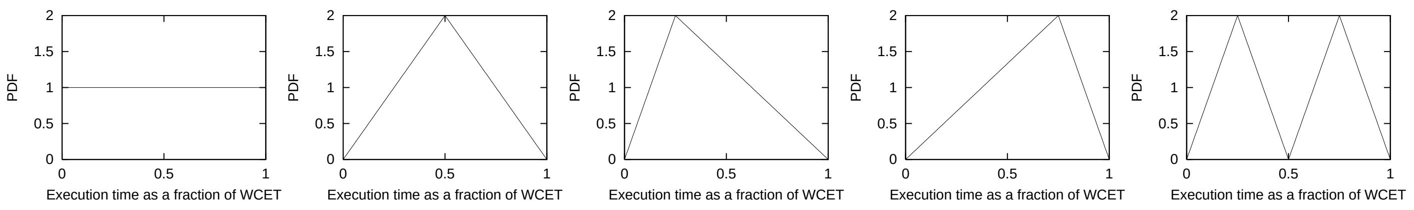

Figure 8: Probability distributions for actual execution time of tasks expressed as a fraction of WCET used in the simulator.

file and write its parameters. These include the number of valid QoS levels, and for each level, the real-time parameters (i.e., period and WCET), average energy, and utility. Once registered, a real-time task will be blocked by our module and invoked periodically, according to the specifications. A task indicates it has completed the current invocation by writing to the special file, and the task is immediately blocked until its next execution period. When the task is re-invoked, the return value from this write operation will indicate the QoS level selected so the appropriate degradation mechanisms can be employed.

The adaptation algorithms select QoS levels for all of the tasks in the system. These algorithms are executed whenever a new task is added to the system, or when a task is removed. In addition, they are also run when the `batmon` module indicates a new measure of energy capacity. In our current implementation, we use a dummy battery capacity measure, so the algorithms are run when the user supplies a new power budget through the `/proc` interface to the `batmon` module.

Our prototype EQoS implementation for Linux 2.2.x kernels is publicly available for those interested [25].

5 Evaluation

In order to evaluate the benefits of EQoS and the relative performance of the adaptation algorithms for a wide range of task sets, we have developed a parametric simulator that models various aspects of energy-adaptive real-time systems. This allows us to explore a large space of multidimensional task set properties very quickly and determine the expected range of behavior for our adaptation mechanisms. In addition, we also perform actual measurements on our implemented system, and validate some simulation results on a more limited set of tasks.

5.1 Simulation Methodology

Our simulator can model the energy consumption of a processor with voltage and frequency scaling hardware. This simulator can use a variety of real-time scheduling policies, as well as several different real-time DVS mechanisms. The simulator takes an input of various parameters describing the simulated hardware, scheduling policies, and a task set with multiple QoS levels defined. In addition, the total available energy as well as the desired system runtime is supplied. The simulator applies one of our EQoS adaptation algorithms, and then simulates the execution of the task set on the modeled processor to determine the total system runtime and the utility gained. The simulation is repeated for each of our other adaptation algorithms.

The simulator assumes that the energy-per-cycle of computation is constant for a given operating voltage. This value is chosen to be the average energy-per-cycle and is scaled by a normalized voltage squared term to account for the reduced power dissipation from reduced voltage operation when DVS is employed. This simplifies the simulation, since the type of instruction executed is not taken into account, so instruction traces are not needed. Instead, it suffices to count the number of execution cycles between scheduling events to determine energy consumed, allowing for an event-driven simulation rather than a much slower cycle-by-cycle trace simulator. Furthermore, we assume that idle cycles dissipate negligible energy, i.e., the system employs an efficient `halt` instruction instead of idle loops. Only the processor energy dissipation is considered here. When DVS is used, we assume the normalized voltage and frequency combinations described earlier in Table 2. Finally, since we are not interested in comparing real-time schedulers or DVS mechanisms, we restrict the system to earliest-deadline-first (EDF) scheduling and the *laEDF* RT-DVS mechanism [26], although one can also trivially apply other scheduling and DVS policies.

We use random real-time task sets to simulate a wide variety of tasks and evaluate adaptation across a wide range of initial energy states. When creating these random task sets, we first consider only the characteristics at maximum quality of service. To reflect the wide range of task periods found in real-time systems, each task has an equal probability of having a short (1–10 ms), medium (10–100 ms), or long (100–1000 ms) period. Within each range, the task periods are selected according to a uniform distribution. WCET for the tasks are assigned according to a similar three-range uniform distribution, and then scaled by a constant to maximize worst-case processor utilization while ensuring the task set is EDF-schedulable (i.e., $\sum C_i/t_i \leq 1$) [15]. Each task is also assigned one of five execution time distributions for its actual execution times to be used in the simulation. Figure 8 shows the five possible probability distributions of task execution time, expressed as a fraction of WCET. Assigned is a random utility value, as well as an average power dissipation value, computed from the average execution time and task period.

Next, we generate the task characteristics for degraded service quality. For each task we define a random number of QoS levels, up to a user-specified maximum. We model three different mechanisms of degraded execution: period extension, imprecise computation, and algorithmic change. To model period extension, WCET and execution-time distribution are kept the same, while the period and average power are elongated and scaled down, respectively, by a uniform random variable within the range (1.0, 2.0), and the utility scaled down by a random number for each degradation of QoS levels. For imprecise computation, WCET, average power, and utility are similarly changed. An algorithmic change reflects a change in all of the terms including execution-time distribution, so all are randomly selected such that the worst-case utilization (C_i/t_i), average power, and utility decrease for each additional QoS level defined. Finally, approximately half of the tasks are assigned an additional QoS level, in which they incur zero power and produce zero utility, reflecting that the tasks are non-critical and may be stopped/dropped altogether if the energy budget warrants it.

We create 1000 task sets, each with 10 tasks, and each of which, in turn, has up to 5 QoS levels. We run each of the adaptation algorithms on all of these task sets to generate the following results. The simulations use a desired

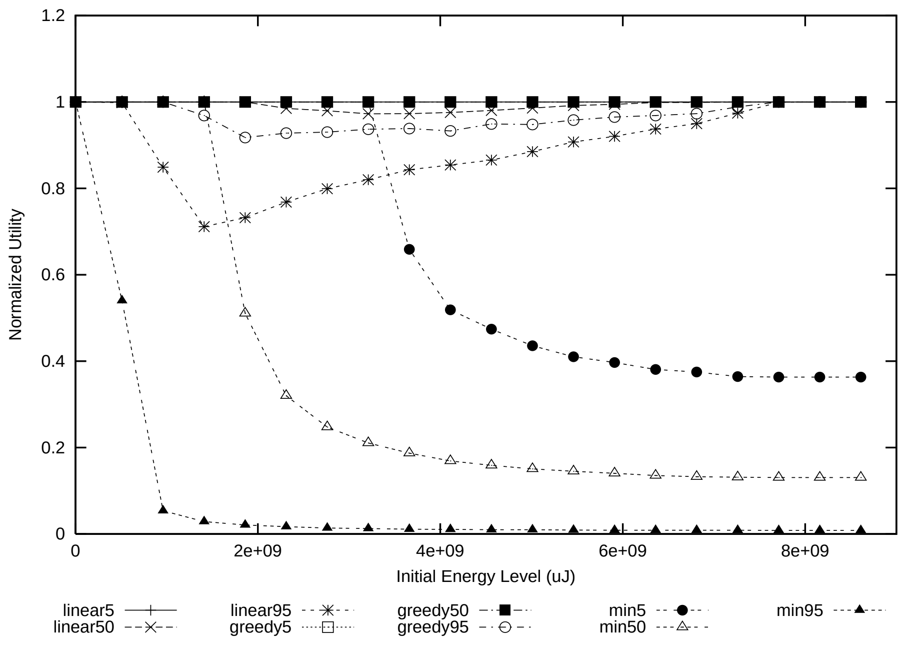

Figure 9: Utility with adaptation, normalized to optimal.

runtime (t_{run}) of 10 minutes, and vary the initial energy, which is specified in μJ. The processor is assumed to dissipate a maximum of 25 W, which is comparable to most current laptop processors. Note that the adaptation is performed only once at the beginning of each run.

5.2 Simulation Results

We have performed extensive simulations to evaluate the benefits and relative performance of EQoS adaptation algorithms. We first compare the algorithms to see how well they adapt task sets to maximize utility, and compare their overheads. We then evaluate the effects of DVS and our compensation mechanism.

5.2.1 Comparison of Adaptation Algorithms

Figure 9 shows the relative total system utility provided under various adaptation algorithms. In this figure, we normalize the results for the algorithms relative to the optimal solutions (produced by the DP or BB algorithm). To show the range of results, we plot the 5^{th}, 50^{th} (i.e., median), and 95^{th} percentile normalized utility values from the 1000 different task sets run for each of the initial energy values. Also for comparison, we show the utility of simply running all tasks at the minimum QoS levels. The greedy heuristic, in particular, performs very close to the optimal solutions, being within 0.9 of the optimal for at least 95% of the task sets for all initial energy values.

The suboptimal adaptations result in runtimes longer than the desired t_{run}, but recall that for the known time-to-recharge scenario, only the utility until time t_{run} is of value. The actual runtimes achieved for this set of experiments is shown, normalized to t_{run} (in this case 600 seconds), in Figure 10. Again, we use percentile plots

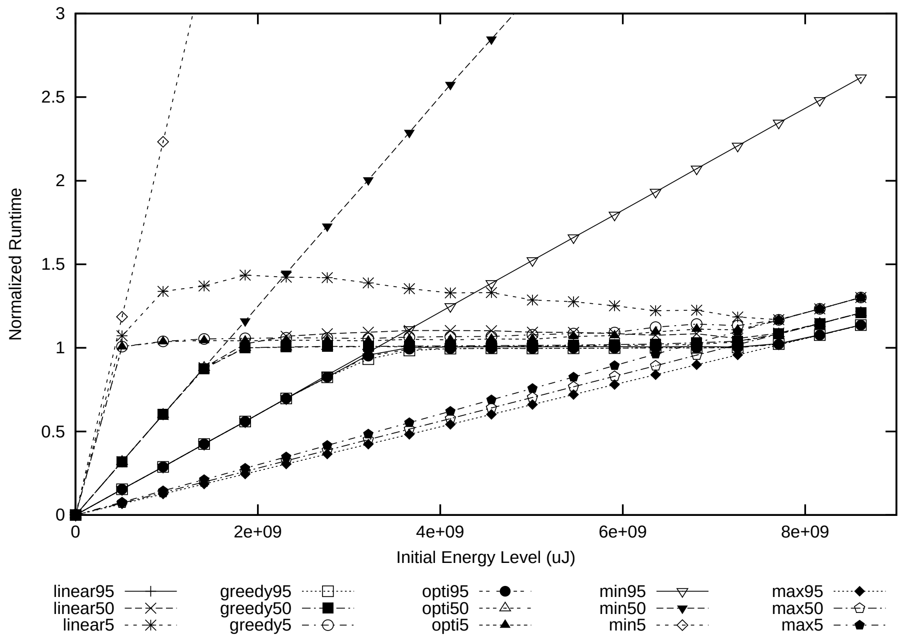

Figure 10: System runtime with adaptation, normalized to t_{run}.

to indicate the range of results. Plots for running tasks at the minimal and maximal QoS levels are shown for comparison. Here, the optimal and greedy methods always result in close to the desired runtime, but the linear heuristic may vary quite considerably.

Figure 11 shows the execution time overheads for the different adaptation methods measured while running

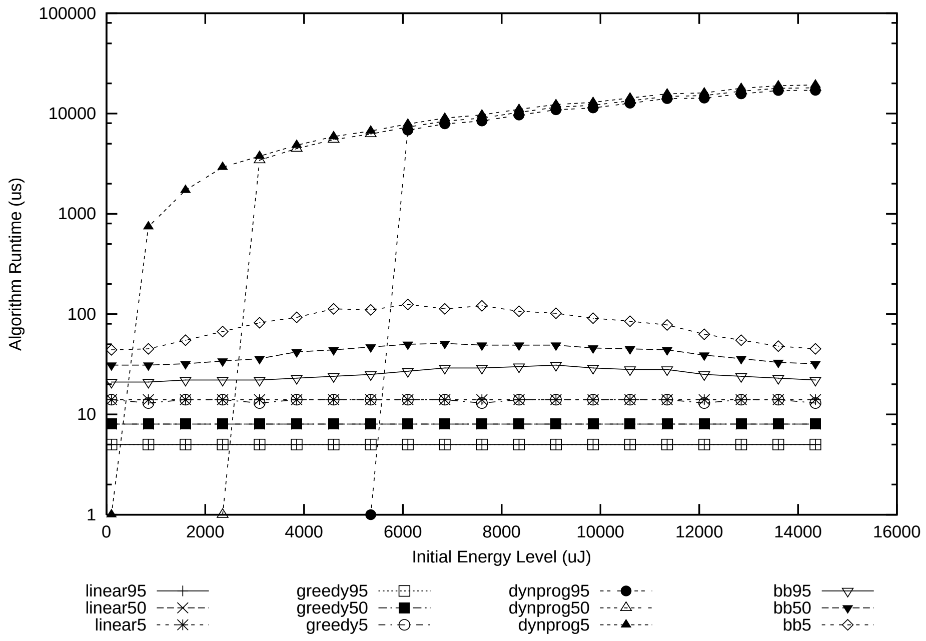

Figure 11: Execution overheads of adaptation algorithms.

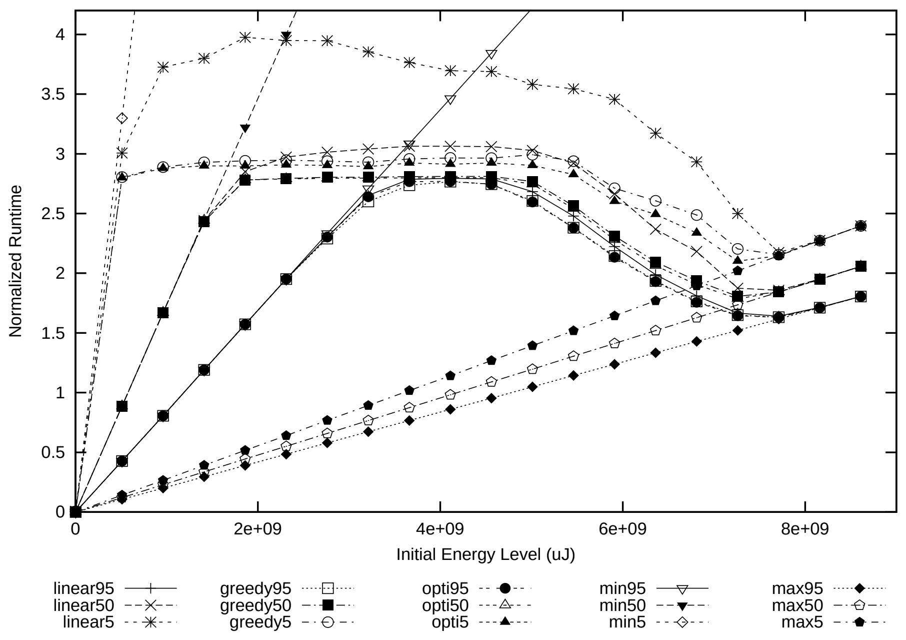

Figure 12: System runtime with adaptation, RT-DVS, normalized to t_{run}.

our experiments. Note that this is plotted on a log scale. All of these were measured on an AMD Athlon XP1500 machine. DP has very consistent, very long execution times, on the order of a few milliseconds. This is due to the large range in power values in our task sets, resulting in a large table of partial solutions. The two heuristics, greedy and linear, are very fast, typically requiring under 10 μs. Most of this time is spent sorting the possible task QoS upgrades, so the actual selection heuristics take only about 2 μs. For these task sets, with 10 tasks and up to 5 QoS levels per task, BB incurs fairly low overheads, a few hundred μs. However, its execution overhead varies greatly, and there is no guaranteed time bound, so in the worst case it may degenerate to an exponential search that can take on the order of tens of minutes with these parameters.

5.2.2 Compensating for the Effects of DVS

We repeat the experiments with the same 1000 task sets, but now also employ an RT-DVS scheduler using the voltage and frequency settings in Table 2. The energy-conserving DVS greatly reduces the average per-cycle-energy consumption of the adapted task sets. The results are plotted in Figure 12. Again, we plot the 5^{th}, 50^{th}, and 95^{th} percentile system runtimes that are normalized with respect to the target runtime among the 1000 task sets for each initial energy value. In general, we see a significant increase in total runtime, often close to 3 times the desired t_{run}. However, since only the utility of task execution until t_{run} is counted, this does not directly benefit the system.

We repeat the experiment again, and this time employing the compensation mechanism discussed in Section 3.4. By selecting higher-energy tasks, we now reduce the runtime to be closer to t_{run} and increase the utility gained. Figure 13 shows the resulting runtimes. For the most part, the resulting runtimes for our adaptation al-

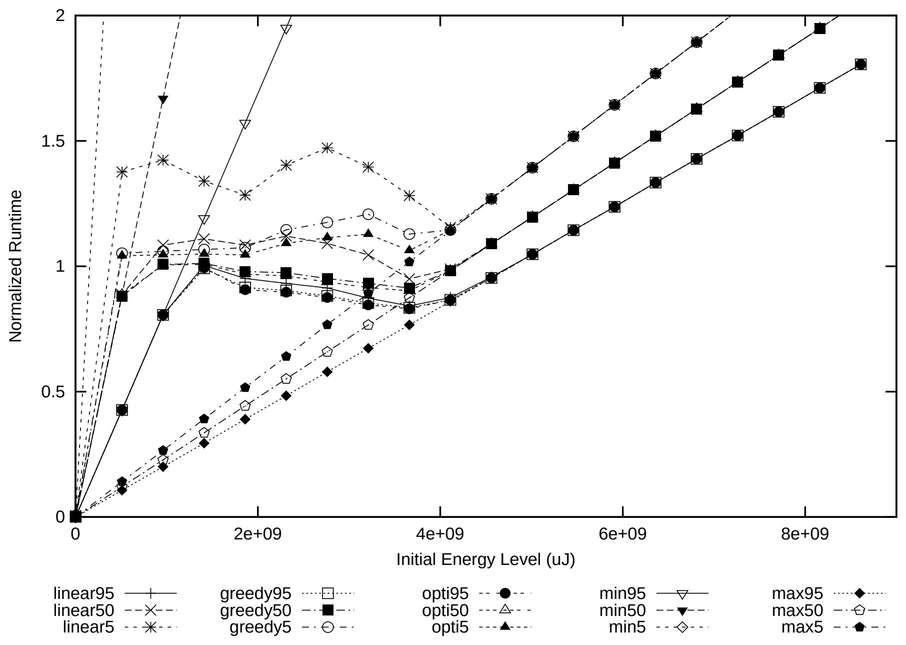

Figure 13: System runtime resulting from adaptation with compensation for RT-DVS, normalized to t_{run}.

gorithms straddle close to 1.0, indicating that a runtime close to t_{run} is achieved. Figure 14 shows the change in utility due to compensation. The utility with compensation mechanism enabled is shown normalized to the utility when compensation is disabled. Although the utility gain will depend heavily on the utilities assigned to tasks at degraded QoS levels, there is a very large and consistent change with our randomized task sets. It is interesting to see that we can achieve largest benefits when the energy in the system is relatively low, which is exactly when the energy per computation is most precious. Furthermore, the distribution of utility gain is fairly independent of the adaptation mechanism used, as indicated by the nearly overlapping percentile curves.

One important observation about DVS compensation from Figure 13 is that, although on average it achieves very close to the desired runtime, there is a high probability that it will over-compensate and have runtime less than t_{run}. It is, therefore, particularly important that, when DVS and compensation are used, the adaptation is not simply computed once. Rather, the system should be re-adapted periodically to ensure that over-compensation is corrected and the desired runtime, t_{run}, is achieved.

5.3 Experimental Measurements

In addition to the extensive simulations, we also evaluate the EQoS framework through measurements of power dissipation using our working implementation on top of Linux OS. The platform for our experiments is a Compaq Presario 1200Z laptop (AMD Mobile Athlon, 1 GHz). By removing the battery and connecting the AC adapter through a current probe attached to a digital oscilloscope, as shown in Figure 15, we are able to accurately measure power dissipation of the system. The machine draws 16–18 W while idle (screen and disk on, no active processes), and peaks at approximately 41 W with full processor load.

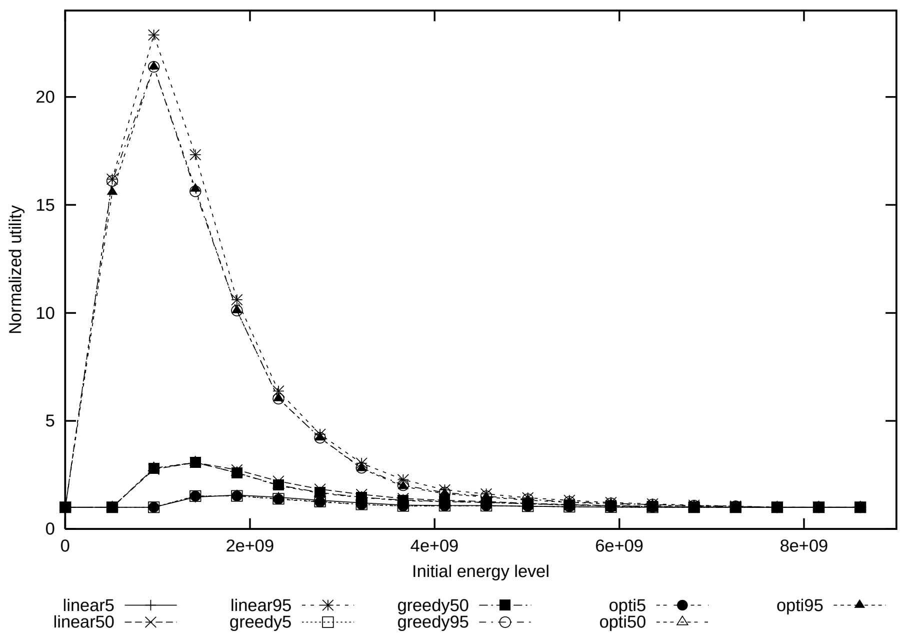

Figure 14: Utility with RT-DVS compensation normalized with respect to utility value without RT-DVS compensation.

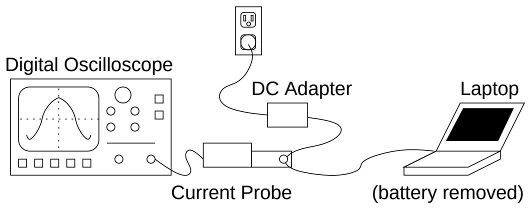

Figure 15: Power measurement on laptop implementation.

This processor incorporates DVS support in the form of AMD's PowerNow! [1] technology. The recommended voltage and frequency settings are shown in Table 3. Note that, based on this and the quadratic relationship between voltage and energy, at most 27% reduction in per-cycle energy cost can be expected.

As we do not have access to actual real-time control applications (for proprietary reasons), we create a set of real-time tasks by modifying `lame` [36], an open-source MPEG Layer-3 (MP3) audio encoder to operate as a periodic real-time task on top of our EQoS framework. Adaptation is performed by varying the "quality" parameter, which selects psychoacoustic models of varying complexity, resulting in a tradeoff between output quality and processing time/energy. The real-time and power characteristics of this task for various QoS levels are shown in Table 4. Note that at the lowest service level, the task is simply not run, and that the utility values were selected

MHz	≤ 500	600	700	800	1000
Volts	1.20	1.25	1.30	1.35	1.40

Table 3: DVS settings for 1GHz Mobile Athlon [1].

QoS level	Period (ms)	WCET (ms)	Avg. CPU Power (W)	Utility
0	22.0	0	0	0
1	22.0	1.45	0.77	100
2	22.0	2.5	1.78	150
3	22.0	3.7	2.72	190
4	22.0	4.3	3.35	220

Table 4: `rt-lame` task characteristics at various QoS levels. Note that WCET and power are specified for 1.0 GHz, 1.4 V operation.

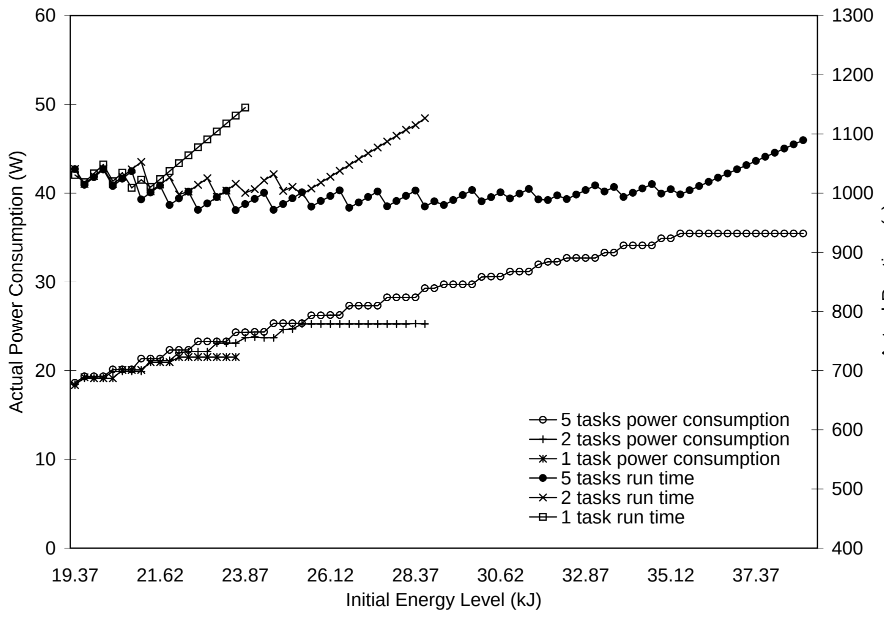

Figure 16: Measured power dissipation and the resulting system runtime after adaptation, no DVS.

such that they provide decreasing marginal returns for each higher QoS level.

We ran the system using the greedy adaptation heuristic, with a target runtime, t_{run}, of 1000 seconds. The task sets consist of multiple instances of our adaptive `rt-lame` task. We vary the total energy parameter, set the power budget assuming 17 W fixed draw, perform adaptation, and measure the power dissipation of the laptop. Figure 16 shows the power dissipation after adaptation for task sets with 1, 2, and 5 instances of `rt-lame` and DVS disabled. Also shown is the resulting runtime based on the input energy parameter. As we can see, when energy is constrained, all three cases closely achieve the desired runtime of 1000 s. Once there is sufficient energy, of course, all tasks are run at the maximum service level and the system runtime increases linearly beyond the target time.

Repeating these experiments with DVS and compensation enabled for additional energy conservation, we obtain the results plotted in Figure 17. Here, task are executed at higher QoS levels in an attempt to keep the same power dissipation to meet the target runtime. However, in spite of the compensation, the DVS results in 5–10% lower power dissipation and reciprocal increase in runtimes. With a known time-to-charge scenario, the energy

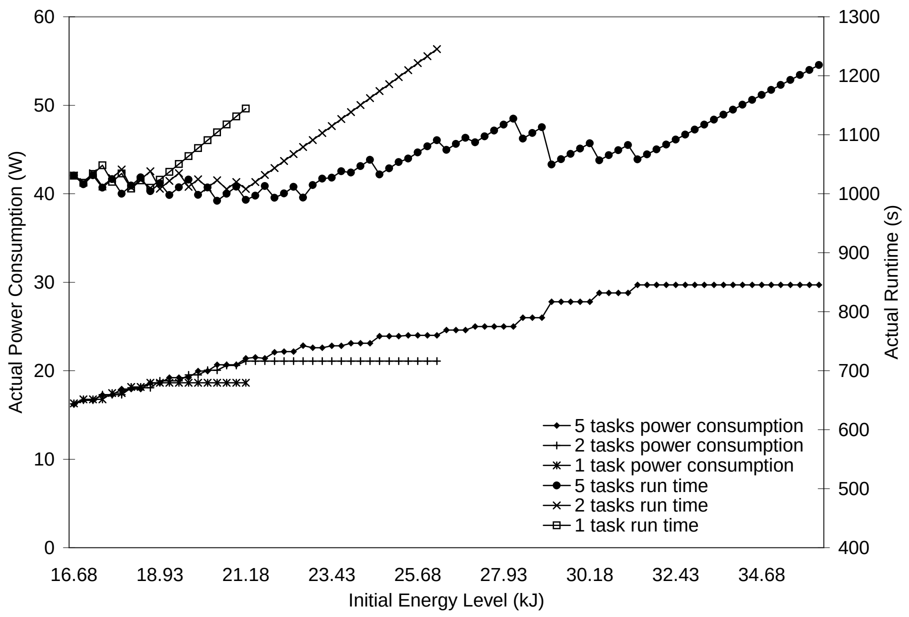

Figure 17: Measured power dissipation and the resulting system runtime after adaptation, with DVS and compensation.

providing this extra runtime could have been better spent running tasks at higher QoS levels to provide greater utility over the target runtime. This again shows that, especially when DVS is used, it is best to adapt task sets periodically, rather than just once, to minimize deviation from the target runtime.

Finally, in Figure 18, we show the utility gained from the completed computation of the system over the target 1000 seconds of runtime. The task set is adapted with available energy, and total utility increases stepwise,

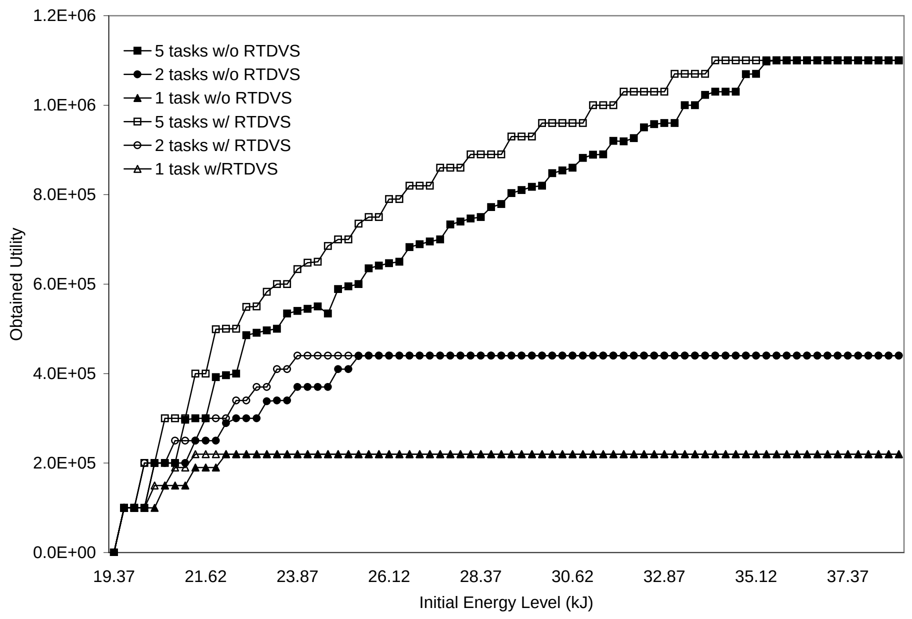

Figure 18: Resulting total utility until 1000s, with adaptation, with and without DVS.

indicating the transition between particular combinations of QoS levels, until all tasks are at their maximum service levels. In these particular task sets, with this laptop's voltage-scaling capability, using DVS provides up to 15% improvement in utility. Of course, this depends on the utilities assigned (e.g., whether marginal utility gains are decreasing or increasing when running tasks at higher QoS levels) and the actual voltage scaling improvements possible on the specific hardware.

6 Related Work

Energy is becoming an increasingly common objective in system optimization. Much of recent research has focused on energy-conserving techniques, especially on the use of DVS. Since the earliest work by Weiser *et al.* [38] on voltage scaling, several papers [7, 22–24, 28] have dealt with DVS for energy savings in general-purpose machines by scaling frequency and voltage to the detected processor load/idle time, or by stretching execution to some target time. More recently, some have used prediction of episodic interaction [5] or soft deadlines and task workload estimation [17] to maintain good human-interaction and multimedia performance with DVS. Although some of the earlier works claim real-time capability, they are not sufficiently rigorous to work with the canonical model of periodic real-time tasks, and, therefore, are not directly applicable to our real-time EQoS framework.

In the domain of real-time applications, it is more difficult to apply DVS, as it is not easy to provide scheduling guarantees in the presence of changing frequencies and worst-case execution times. A few recent works [8, 11, 19, 26, 35] have managed to provide DVS in a true real-time context. Generally, they combine offline analysis with some form of online reclamation or slack-stealing [13] mechanism to ensure deadline guarantees while minimizing energy consumption. Practical heuristics, such as those implemented in [26], are an important element in our EQoS framework.

Various approaches to application adaptation [12, 21] in resource-constrained environments exist in the current literature. A few projects are targeted more specifically to energy adaptation. The Milly Watt project [4] explores application involvement in energy and power management in PDA-class devices. Flinn and Satyanarayanan [6] adapt multimedia applications to ensure a user-specified runtime on battery-powered laptops. Our work extends to real-time systems, where we are restricted by timeliness guarantees and worst-case execution limitations, but have the advantage of generally well-specified task sets.

In real-time systems, adaptation has mostly been restricted to the fault-tolerance domain. Generally, adaptation results in graceful degradation through period extension [32] or by eliminating occasional invocations of tasks [29]. Other works provide reduced service using imprecise computation models [2, 16]. Our EQoS framework leverages these fault-tolerant service degradation techniques and applies them to reduce computational and energy requirements.

In a real-time context, one recent paper [30] addresses value maximization subject to energy constraints. However, the model assumed is much more restricted, using tasks with a common deadline and selecting a subset of the tasks to maximize value, rather than the more general problem of varying QoS to an unrestricted set of real-time tasks. One advantage of the restricted model is that voltage scaling decisions are directly accounted for, rather than compensated later, though the paper does not address random effects of task execution time or optimizing for actual system runtime.

We have formulated the basic energy adaptation problem as a selection of QoS levels. This is reducible to a 0-1 multiple-choice knapsack problem [18], a lesser-known variant of the 0-1 knapsack problem. A few optimal solutions [3, 18, 20, 34] exist for this problem, as well as for its linear relaxation [27, 39]. We use simple optimal algorithms [18] and approximation heuristics for solving MCKP in our EQoS framework.

7 Conclusions and Future Directions

In this paper, we have developed an EQoS framework that provides adaptation of task sets in energy-constrained embedded real-time systems. By leveraging existing methods of real-time adaptation for fault-tolerance and graceful degradation, we have proposed a general adaptive task model and formulated the energy-adaptation problem in a tractable and solvable form. We have shown a couple of optimal solutions as well as simple heuristics to provide maximum benefits or utility with a limited energy budget and a known time-to-recharge. This solution may, in turn, be used to achieve other energy-adaptation goals, such as maximizing benefits irrespective of system runtime.

We first presented detailed simulations showing the relative performance of different adaptation algorithms in maximizing utility for a wide range of task sets. Overall, the optimal solutions outperformed the heuristics, but incurred significantly higher execution overheads. The greedy heuristic turned out to be a good compromise, achieving 0.9 of the optimal value for at least 95% of the task sets we examined. Although the inclusion of energy-conserving DVS mechanisms can greatly increase runtimes and variability, we were able to effectively compensate for these effects and achieve the desired runtime with much improved utility gain.

This EQoS framework has been implemented as an adaptive real-time extension to the Linux operating system. Through measurements on a laptop running a set of audio encoding applications, we have demonstrated the efficacy of energy adaptation in a real-time environment. The measured power consumption after adaptation closely matches the power budget specified, resulting in the desired runtimes and providing maximal utility.

Our current design relies on having real-time tasks for which energy consumption at various service levels is known. In future, we would like to extend this to use monitoring and feedback to determine task energy characteristics online. In this paper we have considered only the low-level conservation mechanisms dealing with CPU power and how these affect adaptation. We would also like to investigate how the energy conserving techniques for other system components, like disk, main memory, and communication networks, can be incorporated into our model.

8 Availability

Our prototype EQoS implementation for Linux 2.2.x kernels and the related utility programs are now made publically available for those who are interested at [25].

References

[1] Advanced Micro Devices Corporation. *Mobile AMD Athlon 4 Processor Model 6 CPGA Data Sheet*, Nov. 2001. Publication # 24319E.

[2] J. K. Dey, D. F. Towsley, C. M. Krishna, and M. Girkar. Efficient on-line processor scheduling for a class of iris real-time tasks. In *SIGMETRICS*, pages 217–228, 1993.

[3] K. Dudzinski and S. Walukiewicz. Exact methods for the knapsack problem and its genralizations. *European Journal of Operational Research*, 28:3–21, 1987.

[4] C. S. Ellis. The case for higher-level power management. In *Proceedings of the 7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 162–167, Rio Rico, AZ, Mar. 1999.

[5] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM'01*, Rome, Italy, July 2001.

[6] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 48–63, Kiawah Island, SC, Dec. 1999. ACM Press.

[7] K. Govil, E. Chan, and H. Wassermann. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st Conference on Mobile Computing and Networking MOBICOM'95*, Mar. 1995.

[8] F. Gruian. Hard real-time scheduling for low energy using stochastic data and DVS processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01*, Huntington Beach, CA, Aug. 2001.

[9] Intel Corporation. http://developer.intel.com/design/intelxscal/.

[10] Intel Corporation. *Mobile Intel Pentium III Processor in BGA2 and MicroPGA2 Packages*, 2000. Order Number 245483-003.

[11] C. M. Krishna and Y.-H. Lee. Voltage-clock-scaling techniques for low power in hard real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 156–165, Washington, D.C., May 2000.

[12] C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek. On quality of service optimization with discrete qos options. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. IEEE, June 1999.

[13] J. Lehoczky and S. Thuel. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1994.

[14] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. of the 8th IEEE Real-Time Systems Symposium*, pages 261–270, Los Alamitos, CA, Dec. 1987.

[15] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[16] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. In *Proceedings of the IEEE*, pages 83–93, Jan. 1994.

[17] J. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 50–61, Cambridge, MA, June 2001.

[18] S. Martello and P. Toth. *Knapsack Problems*. John Wiley and Sons, Ltd., 1990.

[19] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low-Power (COLP'00)*, Philadelphia, PA, Oct. 2000.

[20] R. M. Nauss. The 0-1 knapsack problem with multiple choice constraints. *European Journal of Operational Research*, 2:125–131, 1978.

[21] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St. Malo, France, Oct. 1997.

[22] T. Pering and R. Brodersen. Energy efficient voltage scheduling for real-time operating systems. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium RTAS'98, Work in Progress Session*, Denver, CO, June 1998.

[23] T. Pering and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'98*, pages 76–81, Monterey, CA, Aug. 1998.

[24] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'00*, Rapallo, Italy, July 2000.

[25] P. Pillai. http://kabru.eecs.umich.edu/rtos/eqos.tar.gz.

[26] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 89–102, Banff, Alberta, CA, Oct. 2001.

[27] D. Pisinger. The multiple-choice knapsack problem. *European Journal of Operational Research*, 83:394–410, 1995.

[28] J. Pouwelse, K. Langendoen, and H. Sips. Energy priority scheduling for variable voltage processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01*, Huntington Beach, CA, Aug. 2001.

[29] P. Ramanathan. Graceful degradation in real-time control applications using (m, k)-firm guarantee. In *IEEE FTCS 27*, pages 132–143, 1997.

[30] C. Rusu, R. Melhem, and D. Mosse. Maximizing the system value while satisfying time and energy constraints. In *Proceedings of the Real-Time Systems Symposium (RTSS'02)*, Austin, TX, Dec. 2002.

[31] SBS Implementers Forum. *Smart Battery Data Specification, Revision 1.1*, Dec. 1998. http://www.sbs-forum.org.

[32] D. B. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control systems. In *IEEE RTSS 96*, pages 13–21, 1996.

[33] K. G. Shin and C. L. Meissner. Adaptation and graceful degradation of control system performance by task reallocation and period adjustment. In *11th Euromicro Conf. on Real-Time Systems*, 1999.

[34] P. Sinha and A. A. Zoltners. The multiple-choice knapsack problem. *Operations Research*, 27(3):503–515, 1979.

[35] V. Swaminathan and K. Chakrabarty. Real-time task scheduling for energy-aware embedded systems. In *Proceedings of the IEEE Real-Time Systems Symp. (Work-in-Progress Session)*, Orlando, FL, Nov. 2000.

[36] The LAME Project. http://www.mp3dev.org/mp3/.

[37] Transmeta Corporation. http://www.transmeta.com/.

[38] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23, Monterey, CA, Nov. 1994.

[39] E. Zemel. The linear multiple choice knapsack problem. *Operations Research*, 28(6):1412–1423, 1980.