QoS-based Resource Allocation in Dynamic Real-Time Systems

R. Judd, F. Drews, D. Lawrence, D. Juedes, B. Leal, J. Deshpande, and L. Welch

Abstract—Dynamic real-time systems require adaptive resource management to accommodate varying processing needs. This paper addresses the problem of resource management on a single resource for soft real-time systems (no hard deadline requirements) consisting of tasks that have discrete Quality of Service (QoS) settings that correspond to varying resource requirements and varying utility. The approach employs a feedback architecture wherein task QoS settings are adjusted on-line in order to maintain a desired amount of resource slack. These adjustments are calculated based on incremental changes in resource slack using computationally efficient algorithms that provide nearly optimal utility with computable performance bounds. The feedback architecture provides robustness in the presence of additional resource load and imperfect task resource requirement specifications.

I. INTRODUCTION

The problem of allocating resources to real-time applications has been studied in literature from different angles. Several authors have addressed resource allocation for real-time systems with QoS constraints [5], [9], [10], [11], [20], [13]. In particular, Q-RAM [18], [16], [17], [12] is an algorithmic approach to the problem of finding an allocation of tasks to resources such that the system can satisfy some quality of service requirements as well as other objectives. The authors of this work do not explicitly consider employing Q-RAM for QoS optimization in dynamic environments.

In addition, the application of control-theoretic methods to the design of real-time systems has recently met with considerable success. Common challenges in real-time system design such as nonlinear and stochastic plant models, effector limitations, unknown disturbances, and noisy sensor data identified in [8] indicate a strong connection with control theory and applications. The papers [2], [14] address performance specifications, mathematical modeling, controller design, and performance analysis for scheduling problems in soft real-time systems. Their feedback control architecture is realized in middleware called ControlWare and its effectiveness for quality of service control is demonstrated in a web server environment. Limitations of linear systems and control methods are discussed in [1] with remedies presented that draw from scheduling and queuing theory. Related studies involving a variety of real-time system applications, performance objectives, mathematical modeling approaches, and feedback control architectures can be found in [3], [6], [7], [15], [21], [19].

This paper addresses the problem of resource management on a single resource for soft real-time systems (no hard deadline requirements) consisting of tasks that have discrete Quality of Service (QoS) settings that correspond to varying resource requirements and varying utility. The objective is to provide on-line control of the tasks’ QoS settings so as to optimize the overall system utility while maintaining a desired amount of resource slack.

Unlike Q-RAM, our approach makes incremental adjustments to QoS settings based on incremental changes in resource availability, thereby avoiding the calculation time that would be incurred by recalculating the QoS settings from scratch based on an updated aggregate resource availability. Moreover, our approach uses a feedback architecture wherein the incremental change in resource availability is derived from the deviation between the desired and actual resource slack. This provides robustness in the presence of additional resource load and imperfect task resource requirement specifications. Since the complexity of the underlying optimization problem precludes true optimal solutions, our controller employs computationally efficient algorithms that provide nearly optimal utility. Specifically, true optimal utility is achieved in many cases, lower bounds on achieved utility can be derived, cases where suboptimal utility is achieved can be identified, and deviations from true optimal utility do not diverge as the algorithms sequentially process resource availability increments. Moreover, the algorithms’ low run-time complexity make them suitable for on-line implementation in dynamically changing environments.

The remainder of this paper is organized as follows. Section II presents the system architecture and the model description. Section III proposes a generic heuristic algorithm and derives its theoretical properties. In addition, a improved heuristic is presented. Section IV describes how the algorithms can be implemented. Section V draws conclusions and describes future work.

II. SYSTEM MODEL AND PROBLEM DESCRIPTION

A. System Architecture

In this paper we focus on soft real-time tasks running on a single host under the control of a QoS Manager (QM). We plan to extend our approach to utility optimization across hosts, to the scheduling of hard real-time tasks, and to the allocation of resources to tasks. The task of the controller is to optimize the tasks’ QoS settings with respect to utility based on dynamically varying resource availability. This is accomplished via closed-loop feedback control as shown in Fig. 1. Herein, the parameter $s_d > 0$ represents the desired resource slack. There are various ways to choose this user-defined parameter. For example, if we associate deadlines with the real-time applications and schedule the
tasks according to the Rate Monotonic Scheduling (RMS) algorithm, we could use $s_d \approx 30\%$.

The tasks expose their QoS settings as knobs that the controller can turn to change both their resource usage and utility. In order to calculate the resource usage for a selected QoS setting, the controller makes use of resource profiles that specify the estimated resource usage as a function of QoS settings. Each task is also characterized by a utility function that expresses the user-perceived benefit from running task $T_i$ at QoS level $q_i \in Q_i$.

The overall utility derived from running each task at a given QoS setting is measured by some aggregate of the individual utility functions. The controller aims to always run the tasks in states for which the total utility is maximized. The host controller also monitors the actual current resource slack $s$. Disturbances in the resource utilization due to changes in the environment and dynamically arriving or terminating tasks, as well as inaccurate resource profiles will generally lead to an error in the predicted resource usage of the tasks.

**B. Basic Model**

Here, we adopt a simplified model of a dynamic real-time system. A dynamic real-time system consists of a single resource $R$, available in $R \in \mathbb{R}^+$ units, and a collection of $n$ independent periodic (soft real-time) tasks $T = \{T_1, \ldots, T_n\}$. Both the number of tasks $n$ and the availability of the resource may vary over time. The period of task $T_i$ is denoted $\pi_i$. The period is not necessarily fixed and may undergo dynamic changes at runtime.

We introduce some notation used throughout the paper:

- We use two basic measures that represent time in our model: the first is the point in time as denoted by $t \in \mathbb{R}^{\geq 0}$. The second measure is the number of the $k_i^{th}$ periodic interval of task $T_i$. The $k_i^{th}$ instance of task $T_i$ is denoted by $T_i(k_i)$.
- With each task $T_i$ we associate a (possibly) multidimensional Quality-of-Service (QoS) vector $q_i(k_i) \in Q_i$, for interval $k_i$, that takes values from a finite (not necessarily numerical) set of possible choices for QoS inputs $Q$. Typical examples of QoS levels include frame rate, cryptographic security, compression method, etc.
- The utility of a task $T_i$ is measured by function $u_i : Q_i \rightarrow \mathbb{R}$. The value $u_i(q_i)$ specifies the user-perceived benefit from running task $T_i$ at QoS level $q_i \in Q_i$.
- The total system utility $u : Q_1 \times Q_2 \times \cdots \times Q_n \rightarrow \mathbb{R}$ is defined to be the sum of the task utilities, i.e., $u(q_1, q_2, \ldots, q_n) = \sum_{i=1}^{n} u_i(q_i)$.
- A resource profile $\rho_i^{tot} : Q_i \rightarrow \mathbb{N}$. The value $\rho_i^{tot}(q_i)$ specifies the total amount of resources necessary to achieve QoS level $q_i \in Q_i$. Each task $T_i$ has a minimal resource requirement which is denoted by $\rho_i^{min}$. $\rho_i^{add}(q_i)$ denotes the amount of the resource in addition to $\rho_i^{min}$ that is required to achieve QoS level $q_i$. We assume there exists a QoS setting $q_{i,0}$ such that $\rho_i^{add}(q_{i,0}) = 0$.

By $r_i(k_i)$, we denote the actual amount of resource consumed by $T_i$ as measured (and monitored) by the QoS Manager during the $k_i^{th}$ interval.

We define the actual resource slack of the system at time $t$ as

$$s(t) = R - \sum_{i=1}^{n} r_i \left( \left\lfloor \frac{t}{\pi_i} \right\rfloor \right),$$

where $R$ specifies the current maximum availability of resource $R$. The error between the actual slack and the desired slack is denoted by $s_e$.

In a dynamic environment tasks may enter, leave, or the required resource requirements change due to the environment, or imprecise knowledge of the profiles. In our approach, the QoS Manager will monitor $s_e(t)$ during intervals $k_i$, and make adjustments in the tasks QoS settings for the next interval $k_i + 1$ in $q_i(k_i + 1)$ such that some measure of the aggregate system utility is optimized.

Let $\rho$ describe some amount of resource availability. The utility profile $u_i^{\star} : \mathbb{R}^{\geq 0} \rightarrow \mathbb{R}$ of a task $T_i$ is defined to be the solution to the following optimization problem:

$$u_i^{\star}(\rho) = \max_{q_i \in Q_i} [u_i(q_i)]$$

such that $\rho_i^{tot}(q_i) \leq \rho + \rho_i^{min}$.

The value $u_i^{\star}(\rho)$ is the maximum benefit achieved by allocating $\rho \in \mathbb{R}^{\geq 0}$ amount of resource to task $T_i$ beyond $\rho_i^{min}$. Clearly, we have that $u_i^{\star}(\rho) = 0$ for $\rho = 0$. We define the $q_i^{\star}(\rho)$ as the $q_i$ that maximize $u_i^{\star}(\rho)$. Note that Rajkumar et al. have proposed a solution to this problem [16].

For task $T_i$ assume that $\rho$ can be set to any discrete value in the sequence $\langle \rho_{i,m} \rangle$. Without loss of generality, we assume that $\Delta \rho_{i,m} = \rho_{i,m+1} - \rho_{i,m} > 0$. Let

$$\Delta \rho_{min} = \min_{i=1}^{n} \min_{m=0,1, \ldots} [\Delta \rho_{i,m}]$$

and

$$\Delta \rho_{max} = \max_{i=1}^{n} \max_{m=0,1, \ldots} [\Delta \rho_{i,m}].$$
Moreover, let
\[ \Delta u_{i,m}^* = u_i^*(\rho_{i,m+1}) - u_i^*(\rho_{i,m}) \]
and
\[ d_{u_{i,m}} = \Delta u_{i,m}^*/\Delta \rho_{i,m}. \]

We assume that the utility profiles \( u_i^*(\rho_{i,m}) \) are increasing and concave, that is, \( \Delta u_{i,m}^* > 0 \), and \( d_{u_{i,m+1}}^* - d_{u_{i,m}} < 0 \). Note that the utility profiles can be calculated off-line and stored in tables.

C. Optimization Problem 1 (OP1)

Now, we can define the DQRAM optimization problem. In the following, the desired actual resource availability is described by \( R_d \), which represents the available amount of resource beyond the minimum resource requirements.

**DQRAM Optimization Problem**: Given utility profiles \( u_i^*, i = 1, 2, \ldots, n \), determine QoS settings \( q_i \) for all tasks \( T_i \) such that \( u^*(\rho) \) is maximized subject to

\[
\sum_{i=1}^{n} \rho_i^{add}(q_i(k_i)) \leq R_d.
\]

The above formulation of the optimization problem is equivalent to the following problem formulation which we will use in the remainder of this paper.

**Optimization Problem 1 (OP1)**: Given utility profiles \( u_i^*, i = 1, 2, \ldots, n \), determine for each \( i = 1, 2, \ldots, n \) an index \( m[i] \) for the sequence \( \langle \rho_{i,m} \rangle \), such that the sum

\[
U(m) = \sum_{i=1}^{n} u_i^*(\rho_{i,m[i]})
\]
is maximized subject to constraints

\[
\rho(m) := \sum_{i=1}^{n} \rho_{i,m[i]}^{add} \leq R_d,
\]

where \( m \) denotes the sequence of indices \( m[i], i = 1, 2, \ldots, n \).

Note that the problem OP1 is, in general, NP-complete. This can be shown straightforwardly by reduction from the subset sum problem. But as we will see, there are cases for which the problem can be solved in polynomial time. Also, we will present a good heuristic for the general case.

### III. Algorithmic Approaches

In this section we present and analyze an algorithm for problem OP1. Given indices \( m \). Let

\[ d_{\text{max}} = \max_{i=1}^{n} \left[ d_{u_{i,m[i]}} \right]. \]

**Property 3.1 (P1)**: Suppose \( m \) is selected such that the following equation \( d_{u_{i,m[i]}} \geq d_{\text{max}}, i = 1, 2, \ldots, n \), is satisfied. That is, the slope of \( u^* \) before each index \( m[i] \) is greater than or equal to the slopes after. Then \( m \) is said to satisfy P1.

**Algorithm 1 QoS Optimization (A1)**

<table>
<thead>
<tr>
<th>Input:</th>
<th>( R_d ) the maximum amount of resources to be allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>( m ) the index of each utility profile to be used</td>
</tr>
<tr>
<td></td>
<td>( U ) the total utility achieved by this assignment</td>
</tr>
</tbody>
</table>

The first lemma establishes the complexity of the algorithm A1.

**Lemma 3.1**: The complexity of Algorithm A1 is given by \( \mathcal{O}\left(\frac{R_d}{\Delta \rho_{min}} \cdot \log(n)\right) \).

**Proof**: The values of \( d_{u_{i,m[i]}} \) can be maintained in a heap. Then the complexity of the first line in the while loop is simply the complexity of maintaining the heap, or \( \mathcal{O}(\log(n)) \). Every time the loop is executed, \( r \) is incremented by at least \( \Delta \rho_{min} \), so the number of times through the loop is bounded by \( \left[ \frac{R_d}{\Delta \rho_{min}} \right] \).

**Lemma 3.2**: Algorithm A1 terminates with an \( m \) that satisfies P1.

**Proof**: Since the algorithm adds increments of \( \Delta u_{i,m[i]} \) to \( U \) in order of decreasing \( d_{u_{i,m[i]}} \), and for a given task \( T_i \), the values of \( d_{u_{i,m[i]}} \) are monotonically decreasing, when A1 terminates, \( d_{u_{i,m[i]+k}} < d_{u_{i,m[i]}} \) for all \( i \leq i \leq n \) and \( k > 0 \). Therefore, \( m \) must satisfy P1.

**Theorem 3.1**: Given the sequence of indices \( m \) which satisfies P1 and \( R_d = \rho(m) \), then these indices also solve OP1.

**Proof**: The proof is by contradiction. Suppose there exists a \( m' \neq m \) such that \( U(m') > U(m) \) and \( \rho(m') \leq \rho(m) \). Clearly not all \( m[i] \) can be less than their corresponding \( m[i] \)’s. Since \( \Delta u_{i,m[i]} > 0 \) for all \( i \) and \( m \), this would only decrease \( U(m') \) from the original value of \( U(m) \). Let \( G \) be the set of \( i \) where \( m'[i] > m[i] \) and \( L \) be
the set of $i$ where $m'[i] < m[i]$. Then

$$U(m') = U(m) + \sum_{i \in G} \sum_{j = m[i] + 1}^{m'[i]} du_{i,j} \Delta \rho_{i,j}$$

$$- \sum_{i \in L} \sum_{j = m[i] + 1}^{m'[i]} du_{i,j} \Delta \rho_{i,j}$$

$$\leq U(m) + d \left( \sum_{i \in G} \sum_{j = m[i] + 1}^{m'[i]} \Delta \rho_{i,j} \right)$$

$$= U(m) + d_{\text{max}} (\rho(m) - \rho(m')),$$

where $d_{\text{max}} > 0$. Since the right term of the sum in (1) must be non-negative, we have that $U(m') \leq U(m)$, hence the contradiction.

**Corollary 3.1:** Let $m$ be the sequence of indices which result from executing $A1$ with the resource constraint $R_d$. If $\rho(m) = R_d$ then $m$ solves OP1.

**Proof:** The result follows directly from the application of Theorem 3.1 and Lemma 3.2.

**Theorem 3.2:** Let the discretization of the $\rho_i$’s be equidistant, that is, $\Delta \rho_{\text{min}} = \Delta \rho_{\text{max}}$. Then algorithm A1 will result in a solution of OP1 for all constraints $R_d > 0$.

**Proof:** Let $m$ be the output of A1 for input $R_d$. Let $R'_d = \rho(m)$. If $R_d = R'_d$ then by Corollary 1 $m$ solves OP1. Now suppose $R_d > R'_d$. It is clear that $R'_d + \rho_j, m[j] + 1 > R_d$ for all $1 \leq j \leq n$ since all $\rho_j, m[j] + 1$ are equal. Hence, any indices $m'$ that satisfy $R_d$ must also satisfy constraint $R'_d$, and again by Corollary 1, the output of A1 satisfies OP1.

Now we examine the case where the $\rho_i$’s are not equidistant. There are cases where A1 will no longer return an $m$ that satisfies OP1. Before examining these situations, notice that Lemma 3.1 did not depend on the equidistant property. Therefore, the output $m$ of A1 will solve OP1 whenever its input constraint $R_d = \rho(m)$. Clearly the number of distinct indices $m$ that A1 produces is equal to the sum of the size of the sequences $(\rho_i, m)$, or

$$N = \sum_{i=1}^{n} |(\rho_i, m)|.$$

That is, there are $N$ values of $R_d$ that are distributed in intervals no smaller than $\Delta \rho_{\text{min}}$ and no larger than $\Delta \rho_{\text{max}}$ for which algorithm A1 will optimally solve OP1. So A1 may not give the correct solution to OP1 all the time when $\rho_i$’s are not equidistant; however, when it does not, a small increase (decrease) in $R_d$ will produce a correct solution. In other words any error in A1 will not accumulate. Further, we can bound the error by

**Lemma 3.3:** Let $m'$ be the output of $A1$ for some constraint $R'_d$ and $m$ be the solution of OP1 for the same constraint. Then

$$U(m) \leq U(m') + d_{\text{max}} \cdot p,$$

where $p = \max_{j \in G} \rho_j, m'[j]$, where $G = \{i | du_{i,m'[i]} = d\}$.

**Proof:** Assume $\rho(m') < R'_d$, then $m'$ must solve $\text{OP1}$ for the constraint $R_d = \rho(m')$. Therefore $U(m)$ cannot exceed $U(m')$ by more than the maximum slope of the utility profiles (or simply $d_{\text{max}}$) times $R'_d - R_d$. But $R'_d - R_d$ cannot exceed $p$, otherwise, the algorithm would have incremented $m'[j]$.

Now we examine the causes and possible solutions to problems introduced into $A1$ when $\rho_i$’s are not equidistant.

**Issue 1 (I2).** Suppose the algorithm A1 exits with $r < R_d$, and the values for $d_{\text{max}}$ have been equal for the past 3 times through the loop. Let

$$J = \{j | du_{j,m[j] - 1} = d_{\text{max}}\}$$

$$K = \{k | du_{k,m[k]} = d_{\text{max}}\}$$

then incrementing $m[i]$ of some of the tasks $i \in J$ while decrementing the $m[i]$ for tasks in $i \in J$ may result in a better solution.

**Issue 2 (I2).** The algorithm terminates when $p + \Delta \rho_j, m[j]$ exceeds $R_d$. There may exist an $k \neq j$, where $p + \Delta \rho_k, m[k] \leq R_d$. In this case incrementing $m[k]$ will result in a better solution (see Figure ??).

**Issue 3 (I3).** The algorithm terminates when $p + \Delta \rho_j, m[j]$ exceeds $R_d$. Let $N$ denote the number of times the while loop in A1 executed, and $\Delta p[i]$ and $\Delta u[i]$ (1 $\leq i < N$) be the amounts that $p$ and $U$ were incremented during the $i$th iteration of the loop. Note that $i < N$, since the last increment is removed outside the loop. There may exist an $k \neq j$ and an $1 \leq l < N$ where

$$p + \Delta \rho_k, m[k] - \sum_{i=1}^{l} \Delta p[N - i] \leq R_d,$$

and

$$\Delta u_{k,m[k]} - \sum_{i=1}^{l} \Delta u[N - i] > 0,$$

then incrementing $m[k]$ and decrementing indices corresponding to tasks that were incremented during the last $n$ iterations of the loop, will result in an improved solution.

To totally resolve all three issues is in general a NP-hard problem. This is easy to see since I1 by itself is equivalent to the knapsack problem with the weights of the knapsack items equal to the $\rho_i$ values, the values of the knapsack items equal to the $u_i$ values, and finally, the capacity of the knapsack equal to the remaining slack $R$.

However, we can develop a good heuristics to take advantage of the structure of the problem that will help to
reduce the affects of the three issues. First, if there is more than one task with equal \( \Delta u_{j,m[i]} \), then in the while loop choose the one with largest \( \Delta \rho_{j,m[i]} \). Next, after exiting the loop, examine each entry in the heap in order (except for the top element which caused the loop to exit) to see if it can be added, if so it is added. If the task cannot be directly added but it can be added if \( \Delta \rho[N - 1] \) is removed and the resulting utility is increased, then make the adjustments.

Finally, we modify the algorithm for incremental evaluation. These modifications are presented in algorithms Allocate (A2) and Free (A3).

Algorithm A2 initially follows the logic behind A1 except it initializes \( r = R_0 \), \( U = U_0 \), and \( m = m_0 \). After the first loop is exited, the heap is searched in order in the second loop. This loop looks for two situations. First, if incrementing the index \( m[j] \) will increase the system utility without exceeding the constraint then it is incremented. Second, if incrementing \( m[j] \) exceeds the constraint, the loop makes one more attempt to see if incrementing \( m[j] \) will help. It checks if incrementing \( m[j] \) in conjunction with decrementing the index of the last task added by the first loop will meet the resource constraint. If so, it further checks if this will increase the system utility. If it does it performs the modifications to the indices.

This amazingly simple heuristic modification addresses all three issues. When tasks have equal \( \Delta u_{j,m[i]} \) they will be added in decreasing order of \( \Delta \rho_{j,m[i]} \). When the first loop is exited, all the remaining tasks \( T_i \) with \( \Delta u_{j,m[i]} = \Delta u_{j,m[i]} \) will be searched in decreasing order of \( \Delta \rho_{j,m[j]} \) and added if there is room. This addresses issues I1 by using the largest first heuristic, which has been shown to give good results [4]. Once all the tasks with equal \( \Delta u_{j,m[i]} \) have been searched, the second loop continues until the end of the heap. It will increment \( m[i] \) for tasks with slopes less than \( \Delta u_{j,m[j]} \) that meet the constraint. This clearly addresses issue I2 in a greedy fashion. If \( m[i] \) cannot be incremented without exceeding the constraint, but can be if \( m[last] \) is decremented, then these adjustments are made only if an improvement in utility is achieved. Since the tasks are examined in the order they appear in the heap, then I3 is also addressed in a greedy fashion. For efficiency sake, we examine only the case where one previously incremented task is adjusted. This is a good rule as long as \( \Delta \rho_{max} \) and \( \Delta \rho_{min} \) are close in value.

The incremental nature of the A2 and A3 algorithms are evident. However, A2 must begin with an \( m_0 \) that satisfies P1. This is easily done by recording the amount of resource allocated during the second loop. This amount of resource is called a loan. If A2 returns with a loan, we simply call A3 to free the loan first and then call A2. The resulting \( m \) from A3 will satisfy P1, since allocations are freed in exactly the opposite order that they are allocated in.

**Lemma 3.4:** The complexity of Algorithm A2 is given by \( O \left( \left[ \frac{R_d - R_0}{\Delta \rho_{min}} \right] + n \right) \log(n) \) and A3 is given by \( O \left( \left[ \frac{R_d - R_0}{\Delta \rho_{min}} \right] \log(n) \right) \)

<table>
<thead>
<tr>
<th>Algorithm 2 Allocate (A2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> ( R_d ) the maximum amount of resources to be allocated</td>
</tr>
<tr>
<td>( R_0 ) the current resource allocation (it is assumed ( R_0 &lt; R_d ))</td>
</tr>
<tr>
<td>( U_0 ) the current system utility</td>
</tr>
<tr>
<td>( m_0 ) the current index of each utility profile (it is assumed that ( m_0 ) satisfies P1)</td>
</tr>
<tr>
<td><strong>Output:</strong> ( m ) the index of each utility profile to be used</td>
</tr>
<tr>
<td>( U ) the total utility achieved by this assignment</td>
</tr>
<tr>
<td>( loan ) the amount of resource on loan</td>
</tr>
</tbody>
</table>

```plaintext
set r = R_0;
set U = U_0;
set m[i] = m_0[i], for all i;
set j = 0;
while r ≤ R_d do
  set last = j;
  set j = index of the top tasks in \( H_{max} \);
  set r = r + \( \Delta \rho_{j,m[j]} \);
  set U = U + \( \Delta u_{j,m[j]} \);
  set m[j] ++;
  add task j back into \( H_{max} \);
end while

/* went one increment too far */
set r = r - \( \Delta \rho_{j,m[j]} \);
set U = U - \( \Delta u_{j,m[j]} \);
set m[j] --;
/* see if we can fit a task that does not have the greatest
derivative */
remove top element from heap \( H_{max} \);
set loan = 0;
while r < R_d and \( H_{max} \) is not empty do
  set j = index of the top task in \( H_{max} \);
  if r = r + \( \Delta \rho_{j,m[j]} \) < R_d then
    set loan = loan + \( \Delta \rho_{j,m[j]} \);
    set r = r + \( \Delta \rho_{j,m[j]} \);
    set U = U + \( \Delta u_{j,m[j]} \);
    set m[j] ++;
  else if p + \( \Delta \rho_{j,m[k]} \) - \( \Delta \rho_{last,m[last]} \) < R_d and
  \( \Delta u_{j,m[k]} \) - \( \Delta u_{last,m[last]} \) > 0 then
    set loan = loan + \( \Delta \rho_{j,m[k]} \) - \( \Delta \rho_{last,m[last]} \);
    set r = r + \( \Delta \rho_{j,m[k]} \) - \( \Delta \rho_{last,m[last]} \);
    set U = U + \( \Delta u_{j,m[k]} \) - \( \Delta u_{last,m[last]} \);
    set m[j] ++;
  set m[last] --;
end if
end while
```
OP1. way as in Algorithm A1. Hence, by Theorem 3.1, 

\[ m \text{ and } R \]

and (3) The end of every task period.

The controller described in this paper is invoked at any of 

the following events: (1) Task arrival, (2) Task completion, 

and (3) The end of every task period.

This is a direct result of Lemma 3.1 and the 

observation that the second loop in A2 is executed exactly 

\( n \) times.

**Theorem 3.3:** If Algorithm A2 terminates with \( loan = 0 \) 
and \( R_d = \rho(m) \), then \( m \) solves OP1.

**Proof:** Since \( loan = 0 \), then the second loop did not 
change \( m \). Under this condition, \( m \) is calculated in the same 
way as in Algorithm A1. Hence, by Theorem 3.1, \( m \) solves OP1.

Since we free the loan before calling A2, any errors not 
resolved by the second loop cannot accumulate. Therefore, 
like A1, A2 will return exact solutions to OP1 for at least \( N \) 
values of \( R_d \) and these values occur at intervals no farther 
than \( \Delta \rho_{\text{max}} \) apart. Further, it is obvious that any allocation 
made in the second loop of A2 only improves the system 
utility. So the output of A2 is equal or better than A1. 
Finally, if \( R_d - R_0 \ll R_d \), the execution time of A2 is 
significant faster than A1.

**IV. IMPLEMENTATION ASPECTS**

The implementation of the controller is straightforward. 
The controller described in this paper is invoked at any of 

the following events: (1) Task arrival, (2) Task completion, 

and (3) The end of every task period.

The controller maintains a state for each task which is the 
current value of the index \( m[i] \) for the task and the desired 
value for the index which is denoted by \( n[i] \). At the end of 
a period, the controller determines slack error by querying the 
available resource, subtracting the amount of resource 
reserved for future task periods by previous invocations of 
the controller, and finally subtracted the desired slack. This 
yields the excess resource to be allocated 

\[ s_x = s - \left( \sum_{i=1}^{n} \rho_{i,n[i]} - \rho_{i,m[i]} \right) - s_d. \]

If \( s_x \) is positive, the previous loan is freed, and then 

\( R_d = R_0 + loan + s_x \) is allocated. the current values of 
\( n[i] \) for each task are used to initialize \( m_0 \) in allocated,

and ultimately the output of allocate updates these same 
\( n[i] \)’s. Similarly, if \( s_x \) is negative, \( R_d = R_0 - s_x \) amount is 
freed the \( n[i] \)’s are updated.

At the beginning of each period, the task checks its \( n[i] \). 
If \( n[i] \neq m[i] \), the task updates its QoS setting to \( q_{n[i]} \), sets 
\( m[i] = n[i] \), and starts its next invocation.

When a new task \( T_{\text{new}} \) arrives, the controller temporarily 
assigns its index \( m[\text{new}] \) to its maximum allowed value. 
The excess resource is calculated as follows:

\[ s_x = s - \left( \sum_{i=1}^{n} \rho_{i,n[i]} - \rho_{i,m[i]} \right) - s_d - \rho_{\text{new}}^\text{tot} (q_{\text{new}}'), \]

where \( q_{\text{new}}' \) is the QoS level for task \( T_{\text{new}} \) for \( m[\text{new}] \).

Any loan is freed from the original tasks and then \( R_d = R_0 - (s_x - loan) \) is freed from all tasks including the 
new one. If free terminates with \( m = 0 \), and \( R_d \) has 
not being achieved then there is not enough resources to add 
the tasks and it is rejected. Otherwise, the output \( m \) of free is 
assigned to the \( n[i] \)’s of all tasks. then all of the original 
tasks complete their current period and have updated their 
QoS settings to accomodate the new tasks, the new task is 
launched.

Finally, when task \( T_i \) ends, the excess resource is calcu-
lated as follows:

\[ s_x = s - \left( \sum_{i=1}^{n} \rho_{i,n[i]} - \rho_{i,m[i]} \right) - s_d + \rho_{\text{new}}^\text{tot} (q_{\text{new}}'), \]

where \( q_{\text{new}}' \) is the QoS level for task \( T_{\text{new}} \) for \( m[\text{new}] \). The previous 
loan is freed and then \( R_d = R_0 + loan + s_x \) is allocated.

**V. SIMULATION RESULTS**

The feedback architecture depicted in Fig. 1 was simu-
lated with 50 soft real-time tasks with utility profiles \( u_i^*(\rho) \), 
\( i = 1, ..., 50 \) as defined in Section II-B and a desired slack 
set-point \( s_d = 10\% \). In addition, a random disturbance 
representing additional demand on resource utilization was 
included. At a given instant, the total resource utilization 
consisted of the aggregate resource utilization of the 50 
tasks under QoS Manager control plus the disturbance. 
The actual slack was then 100\% less the total resource 
utilization. The QoS Manager (controller) was implemented 
as described in the preceding section for the case where 
completed periods were the only type of event. Figure 2 
shows excellent tracking of the desired slack in the presence 
of a significant disturbance affecting resource utilization.
VI. CONCLUSIONS AND FUTURE WORK

Extending the work of Q-RAM, we have presented a control-theoretic approach to the problem of optimizing utility with respect to QoS settings of soft real-time tasks competing for a single resource. Our approach employs a feedback architecture to make incremental adjustment in order to provide set-point tracking of desired resource slack, near optimal utility using computationally efficient algorithms, and improved robustness with respect to additional resource load and imperfect task profiles. In future work we intend to extend the theory in the following ways. (1) Handle multiple instances of the same resource (such as multiple hosts or multiple networks). (2) Handle multiple resource types (both hosts and networks, for instance). (3) Handle hard real-time tasks, for single and multiple instances of a resource, and for multiple resource types. (4) Handle the problem of moving tasks from one resource to another to ensure schedulability and optimize utility.

REFERENCES