Low Overhead Dynamic QoS Optimization Under Variable Task Execution Times

Sergiu Rafiliu, Petru Eles, Zebo Peng
Department of Computer and Information Science,
Linköping University, Sweden
{serra,petel,zpe} @ida.liu.se

***Abstract*—Today's embedded systems are typically exposed to varying load, due to e.g. changing number of tasks and variable task execution times. At the same time, many of the most frequent real-life applications are not characterized by hard real-time constraints and their design goal is not to satisfy certain hard deadlines in the worst case. Moreover, from the user's perspective, achieving a high level of processor utilization is also not a primary goal. What the user needs, is to exploit the available resources (in our case processor time) such that a high level of quality of service (QoS) is delivered. In this paper we propose efficient run-time approaches, able to distribute the processor bandwidth such that the global QoS produced by a set of applications is maximized, in the context in which the processor demand from individual tasks is continuously varying. Extensive experiments demonstrate the efficiency of the proposed approaches.**

I. Introduction and Related Work

Today's embedded systems, together with the real-time applications running on them, have achieved a high level of complexity. Moreover, such systems very often are exposed to a continuously varying load due to e.g. variable number of tasks in the system or variable execution times for tasks. In such circumstances, guaranteeing timing constraints in worst-case scenarios is, in most of the cases, not possible or, if possible, comes at the cost of severe under-utilization of resources. In this context, on-line quality of service (QoS) schemes may come handy, both in improving the resource utilization and in dealing with the complex nature of the application and system. At runtime, and depending on the current level of demand, resources are allocated such that the overall quality obtained from the system is maximized.

Quality of service management systems can be described as control systems where, at certain time instances, samples of performance metrics in the system (deadline misses, response times, utilization, etc.) are taken. With this data, an actuation decision is produced, with the goal of maximizing a certain quality metric. This metric may be implicit – linked with one of the performance metrics sampled – or explicit, related to an application specific quantification of quality. Actuation can mean admission of new jobs, changing of task rates, switching task modes, etc.

A vast amount of research has been done regarding QoS, focusing on different goals and system architectures. A general model is given by Rajkumar et al. [3], [1], [2]. The model considers a number of resources that tasks use and a number of quality dimensions for each task. Each quality dimension is represented as an abstract curve. Assuming concave, increasing curves, the developed algorithms perform optimal allocation of resources to each task, such that quality is maximized. The QoS manager is triggered every time a task is added to or removed from the system, or when the abstract quality curves are changed by the user. The main shortcoming of this line of works is that they do not address variations in the amount of resources required by the task. Also the proposed algorithms can be considered heavy-weight for on-line approaches.

Butazzo et al. [4] introduced the elastic model where each task's rate can change within a certain interval. The change of rate is proportional with the task's weight and the QoS manager runs every time when a task is added or removed from the system. Further work deals with unknown and variable execution times [5], optimal control, when the applications are controllers [6], and when dealing with energy minimization problems [7]. Because it considers a fixed objective of optimization dependent on task rate variation [17], the elastic model less expressive.

Lu et al. [9] described a framework for feedback control scheduling, where the source of undeterminism is task execution time variation, and the actuation method is admission control and task rate change. Due to the non-adaptive nature of this method, which is not based on execution time prediction, it only works for applications with small execution time variations.

Cervin et al. [12] proposed a method for adjusting control performance of tasks that implement feedback controllers. This system assumes that feedforward information is available to the QoS manager, every time one of the controlled tasks changes its operational mode.

Combaz et al. [10] proposed a QoS scheme for applications composed of tasks, each described as graphs of subtasks. A subtask has a number of modes, each corresponding to a different quality level. When early subtasks of a task execute with higher execution times than expected, subsequent subtasks will run in modes with lower quality level and lower resource demand.

Yao et al. [11] presented a Recursive Least Squares based controller to control the utilization of a distributed system by means of rate adjustment. The task model is a set of task chains distributed across multiple CPUs and having end-to-end deadlines that must be kept. In this work, it is assumed that the quality level, for each task, is given by an external block, according to some unknown and possibly time varying functions. The controller needs to adjust itself to the system, by learning this functions, before it can output useful actuations. The adjustment of the controller means that it is very slow to respond to execution time changes in the system.

In this paper we consider a uniprocessor system and we focus on controlling its utilization, under large, unpredictable variations of execution times of the different tasks. We achieve this by developing several algorithms which modify task rates, and try to maximize the global quality of service delivered by the system. The QoS functions for every task in the system are defined as mappings from task rates to quality. They may be chosen by objective means (e.g. quality of control functions, if the tasks implement controllers) or by subjective means [14] (e.g. perceived quality of a multimedia application). Since execution times potentially change with every job release, the QoS manager has to run at a high rate itself. Therefore it is important to find an approach that, while efficiently distributing the processor bandwidth, such that global QoS is maximized, does not incur an excessive run-time overhead. The development of such a resource allocation technique is the goal of this paper.

II. Preliminaries

In this section we describe our system model, together with other definitions and concepts that are used throughout the paper.

A. System and Application Model

Our system (Γ) consists of a set of independent tasks, running on a single CPU.

$$\Gamma = \{\tau_i | i \in I\},$$

where I is a finite index set and τ_i it the ith task in the system. A task releases jobs at variable rates, and jobs have execution times that vary in unknown ways. A task can support any rate in a continuous interval.

Each task has a quality degradation function ($q^D(\rho)$ – quality degradation versus rate), associated with its rate interval. This function describes the absolute performance loss when a task runs at a suboptimal rate. We assume that a task running at a higher rate will produce better quality outputs, the optimal rate being the upper bound of its rate interval. The q^D function, based on our definition, is a positive, smooth, strictly monotonic and descending function. By contrast, quality functions would be smooth, strictly monotonic and ascending functions (Figure 1). Convex quality functions correspond to concave quality degradation functions, and vice versa. If the tasks have different importance levels or weights, we consider them included in the q^D functions, by scaling the functions with the corresponding weights.

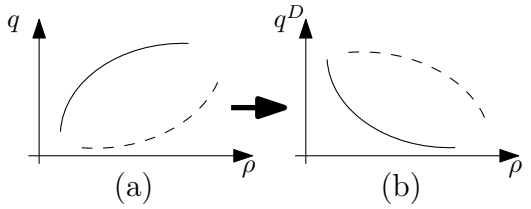

Figure 1. *(a)* quality curves and *(b)* quality degradation curves.

The tasks in the system may be scheduled according to any scheduling policy. In this paper we use *earliest deadline first*(EDF) [13], but other policies, such as *rate monotonic*(RM) can also be used.

Our goal is to maximize the total quality during the runtime of the system. To solve our problem, we employ a controller, running on the system in parallel with the other tasks, that will adjust task rates subject to the current load demand generated by the variable execution times of the tasks. The controller will also adjust its own rate.

A task is defined as a tuple:

$$\tau_i = (\mathbf{P}_i = [\rho_i^{min}, \rho_i^{max}]; q_i^D(\rho_i))$$

where $\mathbf{P}_i$ is the rate interval and q_i^D is the quality degradation function. Throughout the rest of the paper, we use the notations q_i^D and ρ_i to mean both a function and a value. It will be clear from the context, which one we are referring to. A job of τ_i is defined as:

$$\tau_{ij} = (c_{ij}, \rho_{ij}, q_{ij}^D = q_i^D(\rho_{ij}), {q^D}_{ij}^{actual}, r_{ij}, d_{ij} = \frac{1}{\rho_{ij}})$$

where: c_{ij}, ρ_{ij}, r_{ij}, d_{ij}, are the jth job's execution time, rate, response time, and working deadline for the scheduler. q_{ij}^D is the job's expected quality degradation level, defined based on its rate. Due to the variability of execution times, transient overloads may occur, when jobs are not executed before their deadlines. Therefore we also define an actual quality degradation level (${q^D}_{ij}^{actual}$) for each job. ${q^D}_{ij}^{actual}$ is dependent on the job's response time, and is defined as:

$${q^D}_{ij}^{actual} = \begin{cases} q_{ij}^D = q_i^D(\rho_{ij}) & \text{if } r_{ij} \leq \frac{1}{\rho_{ij}} (= d_{ij}) \\ q_i^D(\frac{1}{r_{ij}}) & \text{if } \frac{1}{\rho_{ij}} \leq r_{ij} \leq \frac{1}{\rho_{ij}^{min}} \\ \mathbf{\Phi}_i(\frac{1}{r_{ij}}) & \text{if } r_{ij} > \frac{1}{\rho_{ij}^{min}} \end{cases}$$

where $\mathbf{\Phi}_i(\rho)$ is a penalty function, for the case when response times are larger than the largest period admitted for the task ($\frac{1}{\rho_{ij}^{min}}$).

B. Overall Quality of Service

The quality degradation level of a system, at time instance k is the sum of quality degradation levels corresponding

to the running jobs:

$$\omega[k] = \sum_{i \in I} q_i^D[k]$$

$q_i^D[k]$ represents the quality degradation level of task i's job, running at time k, with the assigned rate $\rho_i[k]$

The total quality degradation level of a system, over its whole runtime $\mathbf{\Theta}$, is

$$\mathbf{\Omega} = \sum_{i \in I} \frac{\sum_{j=1}^{[\mathbf{\Theta}]_i} q_{ij}^D}{[\mathbf{\Theta}]_i},$$

where the inner sum is the cumulative quality degradation level of all jobs released by task τ_i during the system's runtime. We denote with $[\mathbf{\Theta}]_i$ the number of jobs released by task i during the whole runtime $\mathbf{\Theta}$.

We have defined ω and $\mathbf{\Omega}$ based on the expected quality degradation level. However, we are interested in the actual quality degradation, therefore we define ω^{actual} and $\mathbf{\Omega}^{actual}$ in similar fashion, but considering $q^{D^{actual}}$ instead of q^D.

C. Load Demand

The load demand of a task i, at time instance k, is

$$\mathbf{L}_i[k] = \gamma_i[k] \cdot \rho_i[k]$$

where $\rho_i[k]$ is the rate of the last released job τ_{ij} of task i, and $\gamma_i[k]$ is the processor time demand of τ_i at time instance k. As will be discussed in Section II-D, in the case of no overload, $\gamma_i[k]$ is the execution time of τ_{ij}. The load demand of the whole system, at time instance k is

$$\mathbf{L}[k] = \sum_{i \in I} \mathbf{L}_i[k] = \overrightarrow{\gamma}[k] \overrightarrow{\rho}[k] \quad (1)$$

where $\overrightarrow{\gamma}[k]$ is the vector of processor time demands γ_i's and $\overrightarrow{\rho}[k]$ is the vector of task rates ρ_i's.

We can observe that the load demand depends both on job rates and job execution times. Execution times are variable and our task is to assign job rates such that we maximize quality.

In order for all tasks to run with assigned rates, as selected by the controller, it is needed that the total load on the processor does not exceed a certain threshold $\mathbf{L_{ref}}$:

$$\mathbf{L}[k] \leq \mathbf{L_{ref}}, \forall k \in [0, \mathbf{\Theta}]$$

For the EDF scheduler, this threshold is represented by its schedulability bound ($\mathbf{L_{ref}} = 1$). To the extent to which this constraint is violated, job response times are increased and that translates to large quality degradation. This phenomenon is captured by the $q^{D^{actual}}$ attribute.

D. Execution Time Prediction

If the schedulability bound was not kept, we say that the system overloaded. In this case previous jobs have not yet finished and will continue executing into the current job's periods. When computing the load demand of a task, at the current time instance, we account for the overload by considering that the processor time demand is the sum of execution times of all jobs of that task, that have to be executed within the current job's period. Figure 2 illustrates this problem for a task τ_i. The last released job at the current time instance is τ_{ij}, however due to previous overloads, when τ_{ij} was released, job τ_{ij-2} was only partially executed and job τ_{ij-1} was waiting in the queue to be executed. To accurately compute the load demand of task τ_i during the interval of time $1/\rho_{ij}$, we must consider all the execution time that the CPU should spend processing jobs of τ_i during this interval. This is the processor time demand of τ_i at the time instance k.

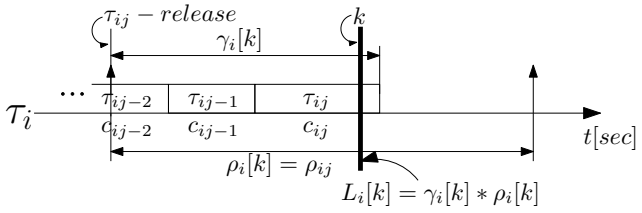

Figure 2. Load model

Figure 2 also shows another problem. At a certain moment, we only know execution times of jobs that have finished (τ_{ij-2} and τ_{ij-1} in our example). For jobs that have not yet finished, we need to predict them. We have chosen to predict execution times using the *single exponential average* method [16]:

$$c_{ij}^{pred} = c_{ij-1}^{pred} + (1 - \alpha) \cdot (c_{ij-1} - c_{ij-1}^{pred}), \alpha \in [0, 1]$$

This formula uses the history of previous measurements to predict the future values. The amount of history that is used, is controlled by the coefficient α. A value of $\alpha = 0$ corresponds to a prediction which is equal to the last known measurement. A value of $\alpha = 1$ corresponds to a prediction which is the initial value (c_{i0}^{pred}) which we consider chosen close to the average execution time of the task. For the experiments presented in Section VI, the value $\alpha = 0.4$ has been found to be appropriate.

III. Problem Formulation

Our goal is to maximize the total quality level, of all tasks, over the whole runtime of the system. This implies minimizing the total quality degradation over the total system runtime $\mathbf{\Theta}$.

$$min\{\mathbf{\Omega}^{actual}\} = min\left\{\sum_{i \in I} \frac{\sum_{j \in [\mathbf{\Theta}]_i} q^{D^{actual}_{ij}}}{[\mathbf{\Theta}]_i}\right\} \quad (2)$$

Because of unknown execution times, the load demand and *actual quality* are unknown. Moreover, when the control decision is taken, the number of released jobs and $\mathbf{\Theta}$ are also unknown. To solve our problem, we adopt a greedy approach by minimizing the expected total quality degradation level at a time instance k, and adding the

constraint that overloads should be avoided:

$$min\{\omega[k]\} = min\left\{\sum_{i\in I} q_i^D[k]\right\} \quad (3)$$

subject to:

$$\mathbf{L}[k] \leq \mathbf{L_{ref}},\ \forall k \in [0, \mathbf{\Theta}] \quad (4)$$

.

IV. Solutions

To meet our goal, we will construct a QoS controller to (1) keep the load by adjusting task rates and (2) minimize quality degradation by properly choosing the adjustments for task rates.

A necessary, but not sufficient, condition for minimizing ω is maximizing the use of processor bandwidth. Therefore, we instruct our controller to follow a reference load ($\mathbf{L_{ref}}$) which we set close to the schedulability bound of our scheduler. Equation (4) thus becomes:

$$min\{\mathbf{L_{ref}} - \mathbf{L}[k]\},\ \forall k \in [0, \mathbf{\Theta}] \quad (5)$$

A general layout of the controller is given in Figure 3. The controlled plant consists of the system of tasks, which are represented by blocks containing their execution times c_i. The variation of execution times, from job to job, is represented by the noises $n_i[k]$. The inputs to the plant are task rates $\rho_i[k]$ and the output are task load demand $\mathbf{L}_i[k]$. The output of the system is the global load demand $\mathbf{L}[k]$. This is fed back into the controller and compared to the reference load $\mathbf{L_{ref}}$ that the system is supposed to follow. The controller takes as an input the difference between the actual load demand and the reference and adjusts task rates accordingly. The controller uses the tasks' quality degradation curves and employs a QoS manager to minimize the expected quality degradation, for each change that is done in the system.

From Equation (1) we can observe that the load demand is a non-linear equation, where both $\overrightarrow{\gamma}[k]$ and $\overrightarrow{\rho}[k]$ are unknown. According to current practice in control theory [15], we linearize our controller, by considering the non-controllable variables, processor time demand, and thus the execution times, to be constant. The real, non-constant execution times of jobs are composed of this constant values and some noise n_i. This provides the source of perturbations that the controller reacts to. When running the controller, we will consider predicted, variable, execution times instead of this hypothetical constant values. We can rewrite the load demand as a function of quality degradation, because the quality degradation functions are inversable:

$$\mathbf{L}[k] = \sum_{i\in I} \gamma_i \cdot \rho_i(q_i^D[k])$$

The minimization goal stated in Equation (5) thus becomes

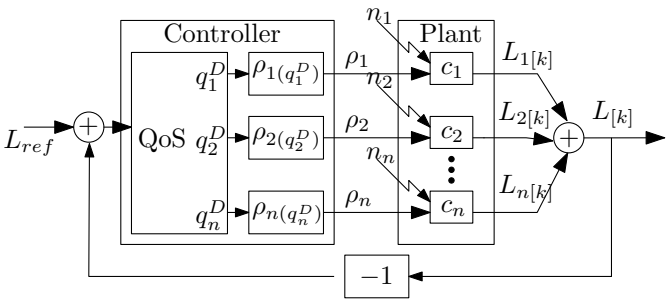

Figure 3. General controller layout.

$$\mathbf{L_{ref}} - \mathbf{L}[k] \approx \mathbf{L}[k+1] - \mathbf{L}[k] = \\ = \Delta L[k] = \overrightarrow{\gamma}\overrightarrow{d\rho}[k] = \nabla L[k]\overrightarrow{dq^D}[k] = 0 \quad (6)$$

where

$$\nabla L[k] = \left(\gamma_1[k] \cdot \rho_1'(q_1^D[k]),\quad \ldots,\quad \gamma_n[k] \cdot \rho_n'(q_n^D[k])\right)$$

is the gradient of the load demand, $\overrightarrow{\gamma}$ is the vector of processor time demand for all tasks, $\overrightarrow{d\rho}$ is the difference vector of rates, and $\overrightarrow{dq^D}$ is the difference vector of expected quality degradation levels which is the unknown variable in our equation.

The controller will choose new task rates by solving a system of equations formed by Equation (6) and a set of relations between all dq_i^D's. We will further specify this relations when we present our alternative solutions in the following subsections. In Figure 3, Equation (6) is represented by the final summation and the feedback loop, and the set of relation between dq_i^D's is embedded into the QoS manager.

We have described our controller layout by considering a fixed number of tasks. However, in practice, our layout can accept a variable number of tasks, coming in, or leaving the system.

The controller's job is also to set its next activation point, which might not be based on a fixed period. At a given time, not all tasks may be important for the controller's outcome, therefore the controller also has to determine the relevant subset of tasks ($J \subseteq I$) that must be considered. We will further discuss these issues in Section V.

For our problem, we have developed four different solutions. The first two do not explicitly consider quality maximization, and have been mainly introduced as baselines for comparison. The other two explicitly approach the goal stated in Equation (3).

A. Constant Bandwidth

Our first approach, called the *constant bandwidth* method, allocates apriori, a constant bandwith (B_i) to each task in the system, by some means (e.g. based on their wheights). Whenever a new job of a task is released, a prediction of

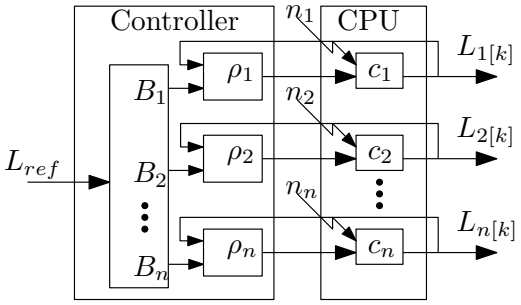

Figure 4. Controller layout for the *constant bandwidth* method.

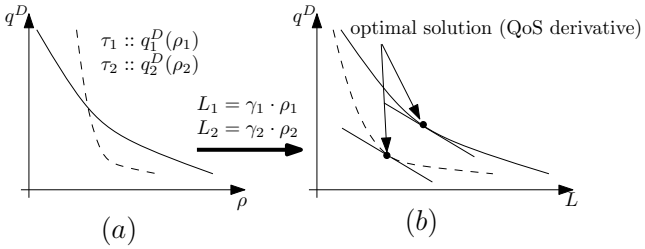

Figure 5. Transformation from $q^D(\rho)$ to $q^D(\mathbf{L})$.

that jobs execution time is made, and based on it, the job's rate is computed, such that the job uses all its allocated bandwidth. Formally put:

$$\rho_{ij} = \frac{B_i}{c_{ij}^{pred}}$$

where

$$\mathbf{L_{ref}} = \sum_{i \in I} B_i$$

and c_{ij}^{pred} is the predicted execution time for the jth job of task i. Figure 4 presents the controller layout for this method. The controller is composed of several independent loops, one for each task in the system. A control loop is activated when its corresponding task releases a new job in the system.

This method does not consider quality degradation curves, and therefore does not meet our goal of maximizing quality. It only addresses the load control aspect. However, this method has a very low overhead.

B. Uniform QoS

Our second approach is called the *uniform QoS* method. This method uses the general controller layout described in Figure 3. In this method, we try to satisfy our goal stated in Equation (3) by choosing task rates such that all tasks have the same expected quality degradation level:

$$q_i^D[k+1] = q_j^D[k+1] = \lambda, \ \forall i \neq j \in J \tag{7}$$

where

$$q_i^D[k+1] = q_i^D[k] + dq_i^D[k], \ \forall i \in J \tag{8}$$

Equations 7 and 8 describe the relations between all dq_i^D.

Our controller will solve the system of equations formed of Equation (6) and Equation (7). We can observe that all dq_i^D's can be described in terms of one value ($dq_i^D[k] = \lambda - q_i^D[k]$) . This value ($\lambda$) can then be obtained by solving Equation (6). Since all equations are linear, the solution can be hard-wired in the code and, therefore, is fast to compute.

This approach can be seen as a superset of the elastic model described in [4], considering variable elastic coefficients, and their values given by the corresponding q^D functions.

C. QoS Derivative

Our third approach is called *QoS derivative* and it uses the general controller layout (Figure 3). Unlike the previous two methods, this method addresses the minimization of the total expected quality degradation level.

To understand how to solve the minimization as required by Equation (3), we will first define quality degradation functions in terms of the resource used ($q_i^D[k] = q_i^D(\mathbf{L}_i[k])$). We do this by applying the linear transformation $\mathbf{L} = \gamma \cdot \rho$ to each q^D curve. These curves change with every change of execution time. Figure 5(a) shows an example of two tasks having quality functions q_1^D and q_2^D. At a certain moment in time, by predicting for these tasks processor task demands γ_1 and γ_2 we can represent the quality functions as functions of load instead (Figure 5(b)). The solution to minimizing ω appears when the derivatives of all $q^D(\mathbf{L})$ curves, in the set points, are equal. A proof of this can be found in [3]. The intuition behind it is the following: when we set rates such that the derivatives are not equal, it happens that one task has the steepest derivative or slope, therefore a slight increase in resources produces a large quality degradation reduction. Another task has the shallowest slope, therefore, the corresponding reduction in resources will produce only a small increase in quality degradation. This rearrangement reduces ω. We can continue doing so until all derivatives become equal. This method only finds the minimum quality degradation if all tasks have convex q^D curves (concave quality curves). This leads to the following equation:

$$\gamma_i[k] \cdot \rho_i'(q_i^D[k+1]) = \gamma_j[k] \cdot \rho_j'(q_j^D[k+1]), \quad \forall i \neq j \in J \tag{9}$$

Because the unknowns $q_i^D[k+1]$[1] appear as arguments to $\rho_i(q_i^D)$ functions, we cannot use these relations directly, therefore we manipulate them using Taylor's approximation. Equation (9) thus becomes:

$$\gamma_i[k] \cdot (\rho_i'(q_i^D[k]) + \rho_i''(q_i^D[k]) \cdot dq_i^D[k]) = \gamma_j[k] \cdot (\rho_j'(q_j^D[k]) + \rho_j''(q_j^D[k]) \cdot dq_j^D[k]) = \lambda, \quad \forall i \neq j \in J \tag{10}$$

[1]We remind the reader that $q_i^D[k]$, $q_i^D[k+1]$ represent expected quality degradation levels.

As with the previous method, our controller solves online the system of equations formed by Equation (10) and Equation (6). We can write all the unknown $\Delta L_i[k]$ in terms of one value λ, that we obtain by solving Equation (6). The solution to this system of equations can also be hard-wired into the code and has an overhead similar to the one of our previous approach.

D. Corner Case

Our final approach is called *corner-case*. Similarly to *QoS derivative*, this method addresses the minimization of the total expected quality degradation level, but for tasks with concave q^D curves. The controller implementing this approach is not based on feedback control theory and does not have the structure described in Figure 3.

The following theorem provides a necessary condition for a resource allocation to minimize $\omega[k]$, in the case of concave quality degradation curves:

Theorem 1. A necessary condition for a resource allocation to be optimal, when all quality degradation curves are concave, is the following: there $\exists$ at most one $i \in J$ such that: $q^{D}{}_{i}^{min} \leq q^{D}{}_{i}^{sol} \leq q^{D}{}_{i}^{max}$. $\forall j \in J \setminus \{i\}, q^{D}{}_{j}^{sol} \in \{q^{D}{}_{j}^{min}, q^{D}{}_{j}^{max}\}$. We call such solutions *corner-cases*.

Proof:

Since $q_i^D(\rho_i)$ are concave, then $q_i^D(\mathbf{L}_i)$ are concave, and $\omega[k]$ in Equation (3) is a concave function as well. We reach the minimum of $\omega[k]$, therefore, when we choose a point on the borders of its definition space. For the purpose of this proof, we will replace $\omega[k]$ with $\omega_n = \sum_{i \in J} q_i^D[k]$, where $n = size\{J\}$.

A border of the definition space of ω_n, has the following property: $\exists i \in J$ for which $q^{D}{}_{i}^{sol} = q^{D}{}_{i}^{min}$ or $q^{D}{}_{i}^{sol} = q^{D}{}_{i}^{max}$. Constructing the overall minimum of our function requires, constructing the minimum in each border and comparing them. To construct the minimum value in a border $q^{D}{}_{i}^{sol} \in \{q^{D}{}_{i}^{min}, q^{D}{}_{i}^{max}\}$, we have to minimize the border function $\omega_{n-1} = \omega_n - q^{D}{}_{i}^{sol}$. We can observe that the border function represents the original function, without one of its dimensions. Its minimum value sits on its borders which are constructed by removing another dimension. Through induction we can see that the minimum value of $\omega[k]$ is a corner-case. ■

At every iteration of the controller, we have to construct all corner-cases and determine which is the one that satisfies Equation (3) and Equation (5). It is not feasible, in general, to generate all corner-cases, and then compare them. Therefore, we will use a greedy algorithm to construct a solution, close to the optimum. The idea behind it is the following: at each step determine the task with the highest expected quality degradation drop, between its maximum and minimum rate, and set its rate to the maximum. The algorithm performs the following steps:

1) Set all tasks to their minimum rate and determine the amount of resources used by all tasks.
2) For each task, compute the slope: $s_i = \frac{q_i^D(\rho_i^{max}) - q_i^D(\rho_i^{min})}{c_i \cdot (\rho_i^{max} - \rho_i^{min})}$).
3) Sort all tasks according to their slopes.
4) While resources are still available, choose the task with the highest slope and give it resources until it reaches its maximum rate, or until there are no more resources available.

V. Controller period and relevant task subset

Except for the *constant bandwidth* method, our controllers work by considering that an overload or underload discovered at the current time instance, will be corrected until the next time instance, by assigning new task rates to tasks, and thus reallocating resources. However tasks only change their rate when they release new jobs. If no job is released between two controller actuations, then no change in rate and load demand occurs. From this, we can observe that we should have a controller period sufficiently large, such that all tasks can release a new job by the next actuation. However, high rate tasks will release several jobs in that period, and any new job can potentially produce a large variation in load demand, due to large execution time variations.

We try to mitigate this problems by changing the controller period and the number of tasks that the controller acts upon, for each controller actuation.

We have experimented with different period assignment policies for the controller (see experiments in Section VI). We have reached the conclusion that between two successive job releases of the same task there should be at least one actuation of the controller. We approximate this behavior by choosing the next actuation of the controller to be equal with the current smallest period out of all running jobs (see also Figure 10).

As stated earlier, it might happen that between two controller actuations, some tasks will not release new jobs, and will not contribute to meeting the controller's design goal. We therefore should instruct the controller not to consider these tasks (see experiments in Section VI). Figure 6 shows an example of a system with two tasks: τ_1 with a high rate and τ_2 with a low rate. By the next controller actuation, only τ_1 releases a new job therefore only it can mitigate the overload or underload observed at the current time instance.

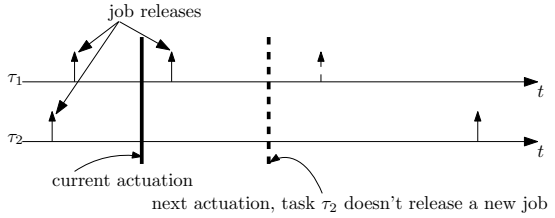

Figure 6. Actuation model

In order to implement such policies, we first determine our next actuation time for the controller. After that,

we determine all tasks that will release new jobs by that time, and these tasks form the relevant subset of tasks (J) considered by the controller at the current time instance.

VI. Experiments

In order to evaluate our solutions we have built a simulation environment that accepts as an input a description of a real-time system, as presented in Section II-A. In a configuration file different configuration parameters are set, such as the simulated runtime of the system, the policy for generating execution times, controllers used, logging options and CPU speed. The simulator outputs the task schedule produced during the runtime of the system, together with a number of metrics such as load demand, utilization, deadline misses and overall actual quality.

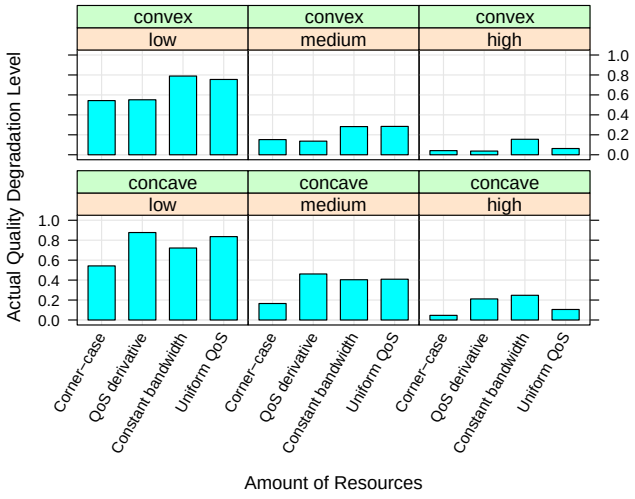

Figure 7. Actual quality degradation level for low, medium, and high amount of resources (convex and concave q^D curves are considered).

For our experiments we have generated a set of 240 uniformly distributed [8] synthetic applications, with the number of tasks varying between 5 and 100. Each task has a rate interval where ρ^{max} is between 2 and 100 times larger then ρ^{min}, and an execution time interval where c^{max} is 100 times larger then c^{min}. We have divided our test cases into two sets, where all tasks have randomly genrated convex and concave q^D curves, respectively. In all experiments, for the *constant bandwidth* method the amount of bandwidth assigned to each task is proportional to its weight. For all other methods, the task weights are included in the q^D functions, as mentioned in Section II-A. In all our experiments we required the controllers to keep a load of $\mathbf{L_{ref}} = 0.95$. We run the test cases for a simulated runtime of $\mathbf{\Theta} = 10^6$ time units. This makes the controllers used, activate between 2000 and 20000 times.

We use the overall actual quality degradation level as a metric for performance. In order to compare different test cases, we scale this metric in the following way: by knowing the number of jobs released during a simulation run, one can know the maximum and minimum quality degradation that can be obtained, and thus scale the expected overall quality degradation level so that it's values lie between 0 and 1. The overall actual quality degradation level, since it also contains the penalty due to deadline misses (see Section II), can, based on this scaling, become larger than 1 for some test cases. In all our experiments we present the average scaled overall actual quality degradation level, during a simulation run of all targeted test cases, with the system set to the same configuration parameters. For ease of presentation, in this section we will call this averaged, scaled metric the actual quality degradation level.

Figure 7 presents the actual quality degradation level, considering cases with *convex* and *concave* q^D curves. For each type of curve, three different cases are considered with regard to the available amount of resources: (1) *"low"* – the processor speed is considered such that $\mathbf{L_{ref}}$ is reached when all tasks are run with rates close to ρ^{min}; (2) *"medium"* – the processor speed is considered such that $\mathbf{L_{ref}}$ is reached when all tasks are run with rates around $(\rho^{min}+\rho^{max})/2$; (3) *"high"*– the processor speed is considered such that $\mathbf{L_{ref}}$ is reached when all tasks are run with rates close to ρ^{max}. These experiments have been run considering that execution times of tasks vary randomly, with a uniform distribution between their corresponding minimum and maximum execution time.

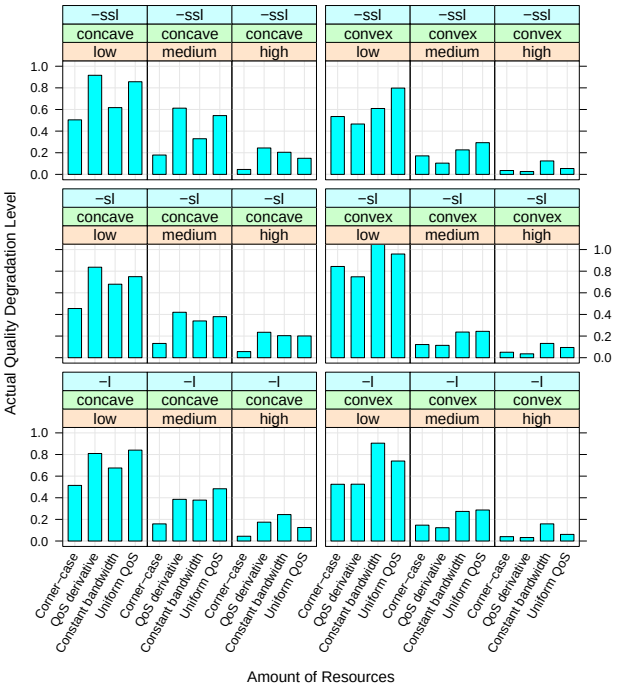

Figure 8. Actual quality degradation level for different amount of resources and different execution time policies (convex and concave q^D curves are considered).

We also consider other policies for generating execution

times for jobs. For a task, execution times can keep constant for several iterations, before they change to another value (*"-l"*). Execution times can also change to new, values, simultaneously, for all tasks, at certain moments in time (*"-ssl"*) or they can change simultaneously and in the same direction (all increasing or all decreasing) for all tasks ("-sl"). The idea behind these policy assignments is the following: tasks might represent pieces of code that have several branches and inputs to tasks might trigger them to take a certain branch ore another for a number of iterations. Also tasks might share a common input and therefore change their execution times at the same time instance. Figure 8 shows the same type of experiments as in Figure 7, but considering the different policies for execution times.

For the above experiments, we have considered no overhead for our controllers, and we have used our best policies for controller period assignment and relevant subset of tasks at each actuation. We will present these policies later on in this section (see Section V). In both Figure 7 and Figure 8 we compare the actual quality degradation level obtained with the approaches described in Section IV. We can observe that for test cases with concave quality curves, the *corner-case* method is the best, and by far better than the rest of the methods. For test cases with convex q^D curves, the best methods are the *corner-case* and the *QoS derivative* methods, with a slight advantage for the last one. The worst methods are the *constant bandwidth* and the *uniform QoS*, which is not surprising since they do not explicitly consider the QoS optimization aspect. We remind that for these experiments we have ignored the controller time overhead (considering that it executes in zero time). We will consider the overhead later in this section.

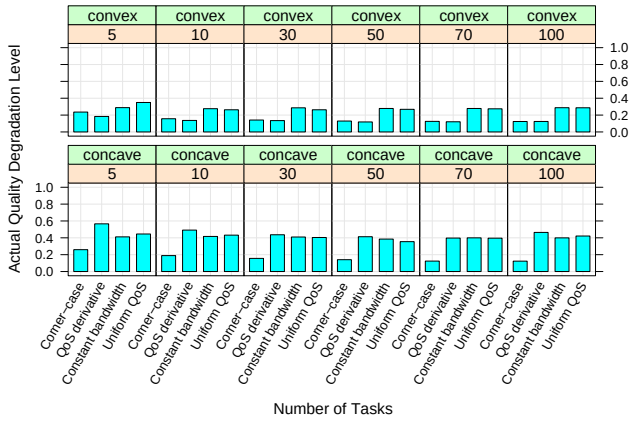

Figure 9. Actual quality degradation level versus number of tasks for test cases of tasks with concave and convex q^D curves.

Our next experiment, presented in Figure 9, looks at test cases with different number of tasks and compares their results. We can observe that our approaches behave consistently independent of the number of tasks in the system.

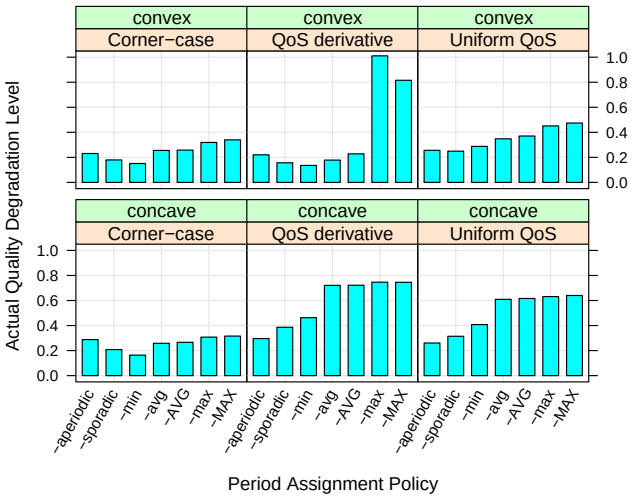

Figure 10. Actual quality degradation level for different controller period assignment policies.

We have run several experiments in order to evaluate various approaches to controller activation (see Section V). The alternative policies are: (1) *"-aperiodic"* – where the controller is activated every time a new job is released; (2) *"-sporadic"* – where the controller runs for every job release, but not more often than the highest possible rate in the system; (4) *"-min"* – where the controller's period is the minimum period among the currently last released jobs of every task; (4) *"-avg"* – where the controller's period is the average period among the currently last released jobs of every task; (5) *"-max"* – the controller's period is the maximum period among the currently last released jobs of every task; and (6) *"-MAX"* – the controller's period is the maximum possible period in the system. We have enumerated our controller period assignment policies in decreasing order of the amount of controller activations that they demand. It is expected that the *"-aperiodic"* method has the largest overhead, since it demands the largest number of controller actuations, and the "-MAX" method has the lowest overhead. Figure 10 shows, for each of our methods, which controller period assignment leads to the smallest increase in quality degradation. We can observe that the policies *"-aperiodic"*, *"-sporadic"*, and *"-min"* generally give the best results. We consider the *"-min"* policy to be the overall best method because of its lower overhead among these three.

Regarding the relevant subset of tasks to be considered by the controller (see Section V), we have run experiments considering two policies: (1) *"-all"* – where we consider all tasks, all the time; and (2) *"-release"* – where we consider only the tasks that will release new jobs until the controller's next actuation point. Figure 11 presents our

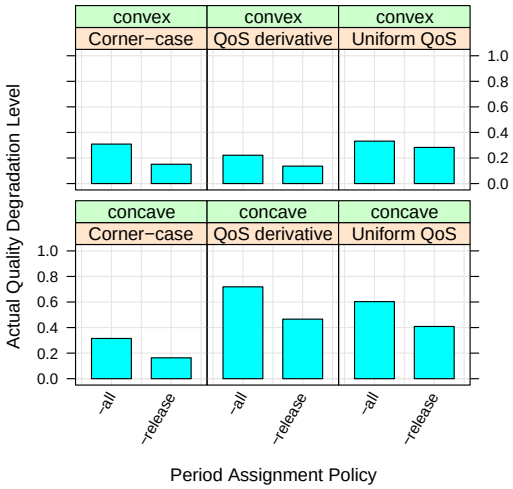

Figure 11. Actual quality degradation level for different task subset policies.

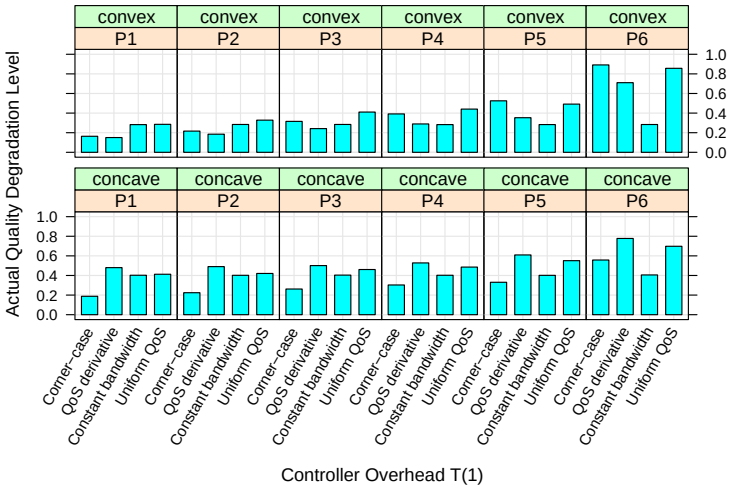

Figure 12. Actual quality degradation level considering controller overheads.

results. We can observe large improvements when using the *"-release"* approach. This approach, also has the lowest overhead, therefore we consider it to be the preferable one[2].

The experiments presented in Figure 10 and Figure 11 support the actuation technique presented in Section V.

We were also interested to determine the performance of the various approaches considering their actual run-time overhead.

We have run the same experiments as before but with considering the actual overhead produced by the controller. The processor load generated by the control actions depends on the complexity of the control algorithm, the number of tasks considered at each control step (see Section V), the number of actuations per time unit and the speed of the actual processor.

In Table I we show the time complexity of the four control approaches. n represents the number of tasks considered during the actuation and m denotes the number of linear segments used to approximate the quality curves.

We have run our simulations considering seven different processors, P_1 to P_6, in decreasing order of their speed. The results show that, also when considering the actual overhead, the corner-case method remains the most efficient in the case of concave quality curves while, in the case of convex curves, the derivative method still produces the best results. Exceptions are the case with processor P_6 for concave curves and P_5 and P_6 for the convex ones. In these last cases, due to the overhead produced on extremely slow processors, the constant bandwidth method, with its very low overhead, produces the best results.

In Table II we show the actual average load produced by the different control approaches when running on the six different processors. As expected, the overhead implied by the constant bandwidth approach is the smallest. Nevertheless, with the exceptions indicated above, the larger overhead produced by the corner-case and derivative methods is compensated by their capacity to produce resource allocations which maximize the obtained QoS.

We have also implemented the *corner-case* and *QoS derivative* controllers on an *AMD Athlon X2* based PC, running at $2GHz$. Considering 100 tasks and an average number of controller activations of 1000 act/sec, the controller load produced was 0.0043 for the *corner-case* and 0.0045 for *QoS derivative*. If we compare to the values in Table II, we can observe that these loads are more than an order of magnitude smaller than those produced with processor P_1 in the experiments illustraded in Figure 12. This means that on a processor 10 times slower than the one used above (e.g. running at $200MHz$), the load produced with 100 tasks and a high activation rates such as 1000 act/sec is still so low that the two control approaches are highly efficient.

Table I
CONTROLLER TIME COMPLEXITY.

Controller	Complexity $O_{(n_i)}^{ctrl-method}$
corner-case	$O(n \cdot log(n))$
QoS derivative	$O(n \cdot log(m))$
constant bandwidth	$n \cdot O(1)$
uniform QoS	$O(n \cdot log(m))$

[2]We have omitted the *constant bandwidth* method in the experiments presented in Figure 10 and Figure 11. The issue of activation policy is irrelevant to this approach since it consists of a number of loops having the periods equal to that of their corresponding tasks. At each actuation of a loop only that particular task is considered.

Table II
CONTROLLER LOAD OVERHEAD.

Controller	$T(1)$					
	P_1	P_2	P_3	P_4	P_5	P_6
corner-case	0.047	0.083	0.112	0.134	0.153	0.219
QoS derivative	0.040	0.072	0.100	0.123	0.144	0.210
constant bandwidth	0.001	0.003	0.005	0.006	0.008	0.016
uniform QoS	0.032	0.059	0.082	0.102	0.120	0.186

VII. Discussions

In the previous section we have considered test cases with different q^D curves and different number of tasks, and we have run them considering different amount of resources and various policies for selection of the controller period and of the subset of considered tasks. The actual runtime overhead of the controllers has also been taken into consideration. The experiments have confirmed the superiority of the *corner-case* method in the case of concave quality curves and the *QoS derivative* method in the case of convex curves.

We have also identified the most efficient policies for dynamic period selection (*"-min"* policy) and for selection of the actual subset of tasks to consider at controller activation (*"-release"* policy).

Another interesting observation is that, while the derivative method is the most efficient for convex quality curves, it performs poorly for concave q^D curves. The *corner-case* method, however, while being the best one in the case of concave curves, performs very well (close to the best) even in the case of convex q^D curves. Thus, if we have a mixed set of applications, with both concave and convex curves, the *corner-case* method will be the most efficient.

VIII. Conclusion

In this paper we have addressed the issue of efficiently allocating resources to concurrent tasks with dynamically varying execution times, based on the tasks' abstract quality curves. We have proposed QoS management approaches and policies and, based on extensive experiments, we have demonstrated their efficiency.

References

[1] C. Lee, J. Lehoczky, R. Rajkumar, D. Siewiorek. *"On Quality of Service Optimization with Discrete QoS Options."* In proceedings of Real-Time Technology and Applications Symposium, pp.276, 1999.

[2] C. Lee. *"On Quality of Service Management."* PhD thesis, Carnegie Mellon University, August 1999.

[3] R. Rajkumar, C. Lee, J. Lehoczky, D. Siewiorek. *"A Resource Allocation Model for QoS Mangement."* In Proceedings of the IEEE Real-Time Systems Symposium, pp. 298-307, December 1997.

[4] G. C. Buttazo, G. Lipari, L. Albeni. *"Elastic Task Model for Adaptive Rate Control."* In Proceedings of the IEEE Real-Time Systems Symposium, pp. 286, December 1998.

[5] G. C. Buttazo, L. Albeni. *"Adaptive Workload Management through Elastic Scheduling."* Journal of Real-Time Systems, vol. 23, pp. 7-24, July 2002.

[6] G. C. Buttazo, M. Velasco, P. Marti and G. Fohler. *"Managing Quality-of-Control Performance Under Overload Conditions."* In Proceedings of the Euromicro Conference on Real-Time Systems, pp. 53-60, July, 2004.

[7] M. Marioni, G. C. Buttazo. *" Elastic DVS Management in Processors With Discrete Voltage/Frequency Modes."* IEEE Transactions on Industrial Informatics, vol. 3, pp. 51-62, February, 2007.

[8] E. Bini, G.C. Buttazzo. *"Measuring the Performance of Schedulability Tests"*, Real-Time Systems, Vol. 30, pp. 127-152, March 2005.

[9] C. Lu, J. A. Stankovic, S. H. Son, G. Tao. *"Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms."* Real-Time Systems, vol. 23, pp. 85-126, 2002.

[10] J. Combaz, J. C. Fernandez, J. Sifakis, L. Strus. *"Symbolic Quality Control for Multimedia Applications."* Real-Time Systems, vol. 40, pp. 1-43, October, 2008.

[11] J. Yao, X. Liu, M. Yuan, Z. Gu. *"Online Adaptive Utilization Control for Real-Time Embedded Multiprocessor Systems."* In Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, pp. 85-90, 2008.

[12] A. Cervin, J. Eker, B. Bernhardsson, K. E. Årzén. *"Feedback-Feedforward Scheduling of Control Tasks."* Real-Time Systems, vol. 23, pp. 25-53, July, 2002.

[13] C. L. Liu, J. W. Layland. *"Scheduling algorithms for multiprogramming in hard-real-time environment."* Journal of ACM, pp. 40-61, 1973.

[14] J. K. Ng, K. Leung, W. Wong, V. Lee, and C. Hui. *"A Scheme on Measuring MPEG Video QoS with Human Perspective."* Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications, pp. 233-241, 2002.

[15] K. J. Åström and B. Wittenmark. *"Computer-Controlled Systems."* Prentice Hall, 1997.

[16] Ya-lun Chou. *"Statistical Analysis"*. Holt International, 1975.

[17] T. Chantem, X. S. Hu, and M.D. Lemmon. *"Generalized Elastic Scheduling."* Proceedings of 27th IEEE Real-Time Systems Symposium (RTSS), pp. 236-245, 2006.