Feedback Control Scheduling in Distributed Real-Time Systems*

John A. Stankovic, Tian He, Tarek Abdelzaher, Mike Marley, Gang Tao, Sang Son and Chenyang Lu
Department of Computer Science, University of Virginia
Charlottesville, VA 22903
e-mail: { stankovic, tianhe, zaher, son, chenyang }@cs.virginia.edu, gt9s@virginia.edu

Abstract

Distributed soft real-time systems are becoming increasingly unpredictable due to several important factors such as the increasing use of commercial-off-the-shelf components, the trend towards open systems, and the proliferation of data-driven applications whose execution parameters vary significantly with input data. Such systems are less amenable to traditional worst-case real-time analysis. Instead, system-wide feedback control is needed to meet performance requirements. In this paper, we extend our previous work on developing software control algorithms based on a theory of feedback control to distributed systems. Our approach makes three important contributions. First, it allows the designer for a distributed real-time application to specify the desired temporal behavior of system adaptation, such as the speed of convergence to desired performance upon load or resource changes. This is in contrast to specifying only steady-state metrics, e.g., deadline miss ratio. Second, unlike QoS optimization approaches, our solution meets performance guarantees without accurate knowledge of task execution parameters - a key advantage in an unpredictable environment. Third, in contrast to ad hoc algorithms based on intuition and testing, our solution has a basis in the theory and practice of feedback control scheduling. Performance evaluation reveals that the solution not only has excellent steady state behavior, but also meets stability, overshoot, and settling time requirements. We also show that the solution outperforms several other algorithms available in the literature.

1. Introduction

Many soft real-time systems such as smart spaces, financial markets, processing audio and video, and the world wide web are not amenable to traditional worst-case real-time analysis. Most of these systems are distributed. They operate in open environments where both load and available resources are difficult to predict. Monitoring and feedback control are needed to meet performance constraints. Several difficulties are observed in meeting performance constraints in these systems. One main difficulty lies in their data-dependent resource requirements, which cannot be predicted without interpreting input data. For example, the execution time of an information server (a web or database server) heavily depends on the content of requests, such as the particular web page requested. A second major challenge is that these systems have highly uncertain arrival workloads; it is not clear how many users will request some resource in the www or how many users might walk into a smart space. A third challenge involves the complex interactions among many distributed sites, often across an environment with poor or unpredictable timing behavior. Consequently, developing certain types of future real-time systems will involve techniques for modeling the unpredictability of the environment, handling imprecise or incomplete knowledge, reacting to overload and unexpected failures (i.e., those not expressed by design-time failure hypotheses), and achieving the required performance levels and temporal behavior.

We envision a trend in the theory of real-time computing that aims at providing performance guarantees without the requirement of fine-grained task execution models such as those involving individual task execution times. Instead, we shall see the emergence of coarse-grained models that describe the aggregate behavior of the system. Coarse-grained models are easier to obtain and they need not be accurately computed. These models are more appropriate for system analysis in the presence of uncertainty regarding load and resources. In this paper, we explore one such model based on difference equations. Unlike the more familiar queuing-theory models of aggregate behavior, difference equation models do not make assumptions regarding the statistics of the load arrival process. Thus, while both types of models describe queuing dynamics, difference equation models are independent of load assumptions and consequently more suitable for systems where load statistics are difficult to obtain or where the load does not follow a distribution that is easy to handle analytically. The latter is the case, for example, with web traffic, which cannot be modeled by a Poisson

* This work was supported, in part by NSF grants CCR-0098269, the MURI award N00014-01-1-0576 from ONR, and DAPRPA ITO office under the NEST project (grant number F336615-01-C-1905).

distribution. Differential equation models include input load as a measured variable. Hence, they are particularly suitable for feedback control architectures in which input load is measured at run-time.

In this paper, we show that our solution outperforms several other algorithms available in the literature. Some of our key performance results are the ability to adapt to unpredictable environments and better transient and steady state response. In addition to improvements in performance, our solution has a basis in the theory and practice of feedback control scheduling. This is in contrast to the more common *ad hoc* algorithms based on intuition and testing where it is very difficult to characterize the aggregate performance of the system and where major overloads and/or anomalous behavior can occur since the algorithms are not developed to avoid these problems.

2. DFCS Framework Overview

Early research on real-time computing was concerned with guaranteeing avoidance of undesirable effects such as overload and deadline misses. Solutions were predicated on knowing worst-case resource requirements a priori. In contrast, in highly uncertain environments, the main concern is to design adaptation capabilities that handle uncertain effects dynamically and in an analytically predictable manner. To address this issue, we propose a framework called Distributed Feedback Control Real-time Scheduling (DFCS). The framework is based on feedback control that incrementally corrects system performance to achieve its target in the absence of initial load and resource assumptions. One main performance metric of such a system is the quality of performance-convergence to the desired level. In our framework, the desired convergence attributes may be specified and enforced using mechanisms borrowed from control theory. These mechanisms are very robust and have been applied successfully for decades in physical process-control systems that are often non-linear and subject to random external disturbances. Before establishing our DFCS framework, we give an overview of the software system being controlled and describe the feedback-control mechanism involved.

We assume that there are N computing nodes connected via a network. Tasks arrive at nodes in unknown patterns. Each task is served by a periodically invoked schedulable entity (such as a thread) with each instance having a soft deadline equal to its period. The periodicity constraint is motivated by the requirements of real-time applications such as process control and streaming media. It is also motivated by recent trends in real-time operating system design, such as temporal isolation of independent applications. Note that temporal isolation is usually achieved using operating system constructs such as capacity reserves, hierarchical schedulers, and resource shares, all of which rely on periodic scheduling in the kernel.

We abstract a typical adaptive system by two sets of performance metrics. The primary set represents metrics to be maintained at specified levels, for example, the optimal utilization of a server, or the desired altitude of an airplane. The secondary set represents negotiable metrics such as service quality. The objective of adaptation is to incur minimum degradation in secondary metrics while maintaining the primary metrics at their desired values. To represent multiple levels of degradation in secondary metrics, we assume that each task has several service levels of different quality. For example, a task can execute for varying amounts of time with the quality of the result improving with greater execution time.

The goal of our DFCS architecture is to maintain the primary performance metrics around their targets. A key intuition that affects the architecture of the feedback loops is that the dynamics of a distributed system manifest themselves on two different time-scales. Fast dynamics are observed on individual nodes. These dynamics arise from local load changes due to individual task arrivals and terminations. Slower dynamics are observed globally on the scale of the entire system. These dynamics arise from changes in aggregate load distribution. Hence, our feedback architecture necessarily involves two sets of loops, local and distributed ones, each tuned to the dynamics of the corresponding scale.

Each node in the distributed system (Figure 1) has a local feedback control system (LFC) and a distributed feedback control system (DFC). The distributed feedback controller is responsible for maintaining the appropriate QoS balance between nodes. The local feedback controller is responsible for tracking the global QoS set point set by distributed controller and ensuring that tasks that are admitted to this node have a minimum miss ratio (MR) and the node remains fully utilized. It is important to note that these two types of controllers form the main parts of the distributed scheduling in the system, but they are not the entire algorithm. In this paper we develop the distributed controller, rely on a local controller (developed earlier and presented in a previous paper [10][11][12]), and then embed these two controllers into the complete distributed scheduling algorithm (see section 3.7). We test the complete algorithm.

Now consider a few more details about the controllers' structure. The distributed controller commands the local controller via the QoS set point. The entire control system of the local node becomes an actuator for the distributed controller to control the state of one local node. The local controller manipulates its actuators in the LFC to achieve the target QoS set point. While the framework is general enough to accommodate different definitions of QoS, in this paper, we let the primary performance metric be the deadline miss ratio (MR). Since performance metrics of admitted tasks can be trivially satisfied if the admitted task set is empty, it is especially important to quantify the loss of QoS due to task rejection to avoid trivial solutions. For this reason, we use two different miss ratio measurements, MR and inNode-MR. The former is the global miss ratio of all submitted tasks (whether they are admitted or rejected). The latter is the miss ratio of admitted tasks only. The distributed controller is responsible for bounding the overall miss ratio, MR, of the system. The local controller is responsible for controlling the in-Node miss ratio of locally admitted tasks as dictated by the distributed controller. These design choices lead to the control system in Figure 1. Note that each of the local controller and the distributed controller has two similar parts, a miss ratio controller and a utilization controller. The miss ratio controller is active during overload. The utilization controller is active at underutilization when no deadline misses are observed. Its purpose is to keep the system sufficiently utilized. These are key parts of our model developed in the next section. The LFC also has a service level ratio controller (SLR) to address our secondary metric.

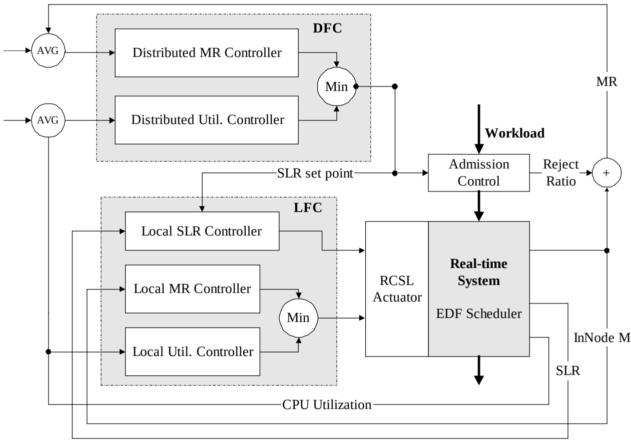


Figure 1: A Typical Node in DFCS.

Admission control (Fig. 1) is based on estimated CPU utilization and the global service level set point and decides to admit or reject tasks from the outside. If one task is rejected, it will be offloaded to another node based on a certain routing policy. Figure 1 also show a Reject Control and Service Level Actuator (RCSL): To track the service level set point from the DFC, the service level actuator modifies the service level of tasks in the system to the appropriate level, then the reject control actuator aborts tasks or notifies AC to admit more if necessary. Finally, the real-time system is the plant processing the request from the users. We can plug various scheduling algorithms into the RTS based on different requirements. Here, we use EDF in our design.

3. DFCS Modeling and Design

The design of DFC control policies requires two components: a task model and difference equations describing the dynamics of the DFC in underutilization and overload situations, respectively. The design process proceeds as follows:

First, we specify the task model. Second, we specify the desired dynamic behavior using both transient and steady-state performance metrics. This step requires a mapping from the performance metrics of adaptive real-time systems to the dynamic response metrics of control systems used in control theory. Third we establish a mathematical model of the system for the purposes of feedback control. We take a simple approach where our model aggregates the overall performance of the system in a single model. We show by our performance study that this model works well, in spite of its simplicity. Finally, based on the performance specifications and the system model from steps 2 and 3, we apply the mathematical techniques of control theory to design the controller that gives analytic guarantees on the desired transient and steady-state behavior at run-time. We map the resultant controller to various nodes in the system depending on the network structure being studied. This step is similar to the process that a control engineer uses to design a controller for a feedback-control system to achieve desired dynamic responses.

3.1. Task Model

For each task T_i, there are N QoS service levels (N > 1). Task T_i running at Service Level q ($0 \leq q < N$) has a deadline $D_i[q]$ and an execution-time $C_i[q]$ that is unknown to the scheduler. The requested CPU utilization, $J_i(q) = C_i[q]/D_i[q]$, of the task is a monotonically increasing function of the service level q, which means that a higher QoS will require more CPU utilization. Let the average CPU utilization needed for a task set at level 0 be U_b. Without loss of generality, the average CPU utilization for a task set at level q is f(q)U_b, where f(q) is a polynomial representing the Taylor's series expansion of the relation between CPU utilization and QoS level. Here we use the first order approximation of this relation to define average requested CPU utilization $J(q)$ of a task set:

$$J(q) = (Aq + 1)U_b$$

where $q \in$ [0, MaxLevel] and MaxLevl = N–1.

We make use of this approximation in the rest of the paper to derive the system model. Note that if this approximation is not appropriate, higher order ones can be used. The design process remains the same.

3.2. System Specification and Metrics

To design adaptive systems, it is necessary to devise specifications and performance metrics for the adaptation process itself. Following the successful practice of the control community in specifying and evaluating the performance of control loops, we have proposed a series of canonic benchmarks that test software adaptation capabilities. These benchmarks generate a set of simple load profiles adapted from control theory; namely, the step load and the ramp load. The step load represents a worst-case load variation: one that occurs in zero time. The ramp load represents a more gradual variation that features a slower rate of change. By experimenting with different rates of change, it is possible to assess the convergence of an adaptive system to the desired performance upon perturbations caused by changes in the environment. Effects of different "speeds" of environmental variation can be analyzed. If the rate of change of the environment is bounded, this analysis can yield guarantees on convergence time and worst-case performance deviation.

We measure system load in percentage of the system capacity. The load corresponding to the full system capacity is said to be 100%. An overload is a system load that is higher than 100%. A load profile *L(t)* is the system load as a function of time. In practice, this load is translated into system-specific parameters for evaluation purposes. For example, a 500% system load can be translated to the request rate of 8,000 request/sec in a specific web server (assuming a fixed requested file type/size distribution). We have shown in [10][12] that these load profiles can be used as benchmarks in adaptive systems to provide a common test-suite for quantifying and comparing the speed of adaptation to load changes of different adaptive systems.

Consider a time window *[(k-1)W,kW]*, where W is called the sampling *period* and k is called the *sampling instant*. During this window, let *M(k)* be the number of task instances that miss their deadline, let *T(k)* be the total number of task instances, and let *MR(k)* be the miss ratio *M(k)/T(k)*. The following metrics are used to describe the quality of adaptation:

- Overshoot M_o: the maximum amount by which *M(k)* exceeds its reference value, expressed as a percentage of the reference value.
- Settling time T_s: The time it takes the miss-ratio to enter a steady state after a load change occurs.
- Steady-state error E_s: It is the difference between *M(k)* and its set point when no disturbance happens and after system transients have decayed. It indicates the DFC's ability to regulate the controlled variable near the reference value in the long term.

Using these metrics, we can compare the effectiveness of feedback-control to other adaptive real-time scheduling policies. We can also specify the desired behavior of the adaptation process in terms of these metrics to guide the control loop design. The critical part towards the enforcement of these metrics is a good aggregate model of the system. This is described in the next section.

3.3. DFCS Modeling

Let the utilization *U(k)* be the fraction of time the CPU is busy in some sampling window *k*. Let *S(k)* be the service level ratio defined as:

$$S(k) = \frac{Avg.\ service\ \ level}{MaxLevel} = \frac{\sum_{q=0}^{MaxLevel} (Num.\ of\ \ Tasks\ \ completed\ \ at\ level\ \ q\ ) \times q}{(T(k) - M(k)) \times MaxLevel}$$

Before applying control theory to design a controller from specifications of adaptive behavior, it is necessary to model the controlled DFCS mathematically. By this we mean deriving the relation between utilization, service level ratio and the resulting miss ratio. Equivalently, we can relate utilization and service level ratio to the number of missed deadlines, which is the approach we take below. When the number of missed deadlines is known, the miss ratio can be trivially obtained. In each time window $[(k-1)W, kW]$, CPU utilization is proportional to the number of tasks that finish successfully. This relationship can be modeled as:

$$U(k) = c(k) \times (T(k) - M(k)) \times J(q) \qquad (1)$$

where *c(k)* is the percentage of the arrived tasks that finish in the same sampling window. For example if *c(k)=1*, all tasks arrive and finish in the same period. From the perspective of control theory, worst-case conditions for convergence stability are those when system gain is maximum, i.e., when the system is most sensitive to changes in its inputs. The maximum gain condition corresponds to *c(k)=1*, in which case equation (1) can be simplified as:

$$U(k) = (T(k) - M(k)) \times J(q) \qquad (2)$$

From definitions of $J(q)$ in section 3.1 and $S(k)$ above, we can derive the following formula:

$$J(q) = (1 + A \times S(k) \times MaxLevel) \times U_b \qquad (3)$$

Combining Equations (2) and (3), we get

$$U(k) = (T(k) - M(k)) \times (1 + A \times S(k) \times MaxLevel) \times U_b \quad (4)$$

Two important subcases arise in modeling the system: namely, overload and underutilization. They are modeled separately in the two subsequent subsections, respectively.

3.4. System Dynamics at Overload

When the DFCS is overloaded and tasks begin to miss their deadlines, there are two approaches to tackle the situation: Admission control and QoS adjustment. Admission control reduces a node's local miss-ratio by rejecting incoming requests. QoS adjustment tries to accommodate more tasks by degrading their service levels. In the DFCS design, we deem task rejection the same as missing the task's deadline. Hence, we favor QoS adjustment to adjust miss control. Here we get a difference equation that describes how QoS adjustment affects the number of misses when the system is overloaded $(M(k) > 0)$. From Equation (4), we obtain

$$M(k) = T(k) - \frac{U(k)}{U_b(1 + A \times S(k) \times MaxLevel)} \qquad (5)$$

Since we assume EDF scheduling, when deadline misses occur it must be that the CPU utilization *U(k)* is 100%. Substituting this in Equation (5) and obtaining its differential, we get the linearized small-signal model of the system:

$\Delta M(k) = G_M \times \Delta S(k) + \Delta T(k)$ where $\Delta M(k) = M(k) - M(k-1)$,

$\Delta S(k) = S(k) - S(k-1)$, $\Delta T(k) = T(k) - T(k-1)$

$$G_M = \frac{A \times MaxLevel}{U_b(1 + A \times MaxLevel \times S(k-1))(1 + A \times MaxLevel \times S(k))} \quad (6\ )$$

We use Equation (6) as the model for the purpose of controller design in overload situations.

3.5. System Dynamics at Under Utilization

When DFC is underutilized, we model the number of missed deadlines as zero. Hence, this is not an appropriate measurement for control purposes. Instead, we switch to utilization measurements. We can increase the QoS of the task set when the utilization is low to improve our service to the user. Here we obtain a difference equation that describes how QoS adjustment affects the CPU utilization when the system is underutilized (*U(k)* <100%). We set the number of misses *M(k) = 0* in Equation (4). The resulting equation is differentiated yielding the following model:

$$\Delta U(k) = G_U \times \Delta S(k) + G_t \times \Delta T(k) \quad (7)$$

where $G_U = (T(k-1) \times A \times MaxLevel \times U_b$

$G_t = (1 + A \times S(k-1) \times MaxLevel) \times U_b$, $\Delta U(k) = U(k) - U(k-1)$

3.6. DFC Loop Design

Having modeled the node dynamics, we now apply control design methods to the distributed feedback control loop. First, we define a set of performance specifications we want to achieve. Based on our knowledge of the DFC model and the performance specifications, we apply a control design method called Root Locus to tune the distributed controller, which is the crucial part of the DFC. Due to space limitations, we do not review local controller design here, which has been intensely studied in our previous work [10][12].

3.6.1. Design of the Control Loop

In the distributed case, we want each node in the DFCS to provide the same QoS to the user. This property is often preferred in many distributed applications. For example, in a web server farm, the QoS of each HTTP request should be independent of where this request is served in the farm. So the major goal of the distributed controller is to calculate the QoS set point for the system, based on the global miss number error $E_M(k)=M_S - M(k)$ and/or the global CPU utilization error $E_U(k)=U_S - U(k)$ as shown above, the DFC can be modeled with two difference equations. One describes the relation between the changes of the service level ratio and the changes of miss number when the whole system is overloaded; the other models the relation between the changes of the service level ratio and the changes of CPU utilization when the system is underutilized. Based on this knowledge, we design the distributed feedback control loop.

Because the external workload is not under our control, we deem *ΔT(k)* as the external disturbance. Let G_U be the gain from *ΔS(z)* to *ΔU(z)* when the system is underutilized and G_M be the gain from *ΔS(z)* to *ΔU(z*) when system is overloaded. We get:

$M(k) = M(k-1) + \Delta M(k) = M(k-1) + G_M \Delta S(k)$ when *M(k-1)>0* (8)

$U(k) = U(k-1) + \Delta U(k) = U(k-1) + G_U \Delta S(k)$ when *U(k-1)<1* (9)

where G_M and G_U are defined in (6) (7), respectively.

We can now draw the block diagrams of the feedback control system. When the system is overloaded, the distributed miss feedback control loop is invoked. It is shown in Fig 2. The components inside the dotted rectangle describe the dynamics of the controlled process with input *ΔS(z)* and output *M(z)*, where $C_M(z)$ is the miss controller to be designed and M_S is the miss set-point. Note that while the output of this figure is the number of missed deadlines, the miss ratio is simply that output divided by the total number of tasks. Either metric can be used depending on the chosen set point. More importantly, note that while the controlled system gain does change by a multiplicative factor when the metric used is miss ratio, the overall loop gain remains the same. This is because the designed controller gain in this case will be multiplied by the inverse of that factor. With the above observation in mind, the discussion below applies to both miss ratio and miss number control.

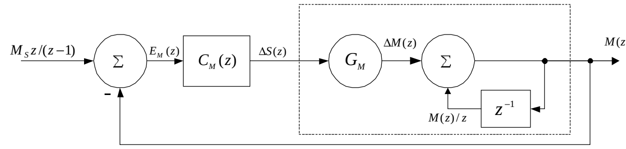


Fig. 2 Miss Feedback Control Loop

When the system is underutilized, we use the distributed utilization feedback control loop shown in Fig 3. $C_U(z)$ is the CPU utilization controller to be designed and U_S is the CPU utilization set point.

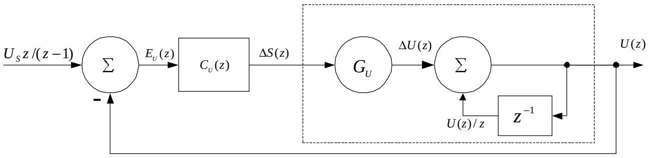


Fig. 3 CPU Utilization Feedback Control Loop

In z-transform notation we have:

$$M(z) = H_M(z)\frac{M_S z}{z-1} \text{ with } H_M(z) = \frac{C_M(z)G_M}{1 - z^{-1} + C_M(z)G_M} \quad (10)$$

$$U(z) = H_U(z)\frac{U_S z}{z-1} \text{ with } H_U(z) = \frac{C_U(z)G_U}{1 - z^{-1} + C_U(z)G_U} \quad (11)$$

Here the gains G_M and G_U are assumed to be set at some fixed values for nominal control design and analysis. Because our system intrinsically has an integral part, it is enough to use only a Proportional controller to design

$C_M(z)$ and $C_U(z)$ to guarantee the stability and zero steady state error. A general form of the digital Proportional controller in the time domain and z-domain is:

$\Delta S(k) = K_p E(k)$ (t-domain) $C(z) = K_P$ (z-domain) (12)

Here we denote K_M as the proportional term for the miss controller and K_U for the utilization controller. These values are substituted in Equations (10) and (11). Setting $C_M(z) = K_M$ and $C_M(z) = K_U$ we get:

$$M(z) = H_M(z)\frac{M_S z}{z-1} \text{ with } H_M(z) = \frac{K_M G_M z}{(1+K_M G_M)z-1} \quad (13)$$

$$U(z) = H_U(z)\frac{U_S z}{z-1} \text{ with } H_U(z) = \frac{K_U G_U z}{(1+K_U G_U)z-1} \quad (14)$$

3.6.2. Stability

According to control theory, system performance is determined by the poles of the closed loop transfer function. From Equations (13) and (14), we get $1/(1+K_M G_M)$ as the pole for $H_M(z)$ and $1/(1+K_U G_U)$ as the pole for $H_U(z)$, which are inside the unit cycle. Hence, for the DFC system, stability is ensured.

3.6.3. Steady State Error

Based on the Final-Value Theorem, the steady state values of $M(k)$ and $U(k)$ are:

$$\lim_{k\to\infty} M(k) = \lim_{z\to1}(z-1)M(z) = \lim_{z\to1}\left\{(z-1)\frac{K_M G_M z}{(1+K_M G_M)z-1}\times\frac{M_s z}{z-1}\right\} = M_S$$

$$\lim_{k\to\infty} U(k) = \lim_{z\to1}(z-1)U(z) = \lim_{z\to1}\left\{(z-1)\frac{K_U G_U z}{(1+K_U G_U)z-1}\times\frac{U_s z}{z-1}\right\} = U_S$$

This result theoretically proves that the DFC system can bring the miss number and CPU utilization to their set point in steady state with zero error. It can also be verified that for a constant external disturbance $\Delta T(k) = \Delta T$, this asymptotic property still holds.

3.6.4. Settling time

Settling time can be determined by the poles inside the unit cycle. The closer the pole is to the origin, the shorter the settling time. Theoretically, we want $1/(1+K_M G_M)$ and $1/(1+K_U G_U)$ to be as small as possible to reduce settling time. However, there are limitations on how large the gains K_M and K_U can be. For example when system is in overload, if we reduce SLR too much, the system will become underutilized. The limitation can be described as a maximum rate of change:

$$\Delta M(z) \le E_U(z) \text{ and } \Delta U(z) \le E_U(z) \quad (15)$$

From the block diagram we know that $\Delta M(z) = K_M G_M E_M(z)$ and $\Delta U(z) = K_U G_U E_U(z)$. Thus, the following should be satisfied:

$$K_M G_M \le 1 \text{ and } K_U G_U \le 1 \quad (16)$$

While the previous sections establish the theoretic foundation for DFC design, in reality G_M and G_U are time-varying, and the proportional term of the distributed controller is taken in (17) to deal with a worst case situation:

$$K_M = \frac{1}{Max\{G_M\}} \text{ and } K_U = \frac{1}{Max\{G_U\}} \quad (17)$$

If we let $K_M G_M = 1$ and $K_U G_U = 1$, the poles of $H_m(z)$ and $H_U(z)$ will be 0.5. According to control theory, the settling time is determined by the distance of the pole from the origin of the root locus plot. With radius of 0.5, the theoretical settling time is about 8 sampling periods, which guarantees the specification in Table 1. In the experiment, based on the model, the calculated controller settings are 0.82 and 1.22 for the miss and utilization controllers, respectively.

3.6.5. Performance specification of DFC

As an example, here we assume the desired DFCS has the performance specifications listed in table 1. The sampling period is W = 1 sec. The transient and steady state performance requirements are the following: (1) the miss ratio and CPU utilization of DFCS should be stable after a step load of 160%. (2) DFCS should settle down to steady state within 10-sec. (3) Both the miss ratio and CPU utilization of DFCS should remain at their set-point in the steady state.

Load Profile	SL(0,160%)
Sampling Period W	1 sec
Settle Time T_s	< 10 sec
Steady-state miss number M_s	< 1% *T(k)*
Steady-state CPU utilization	>99%

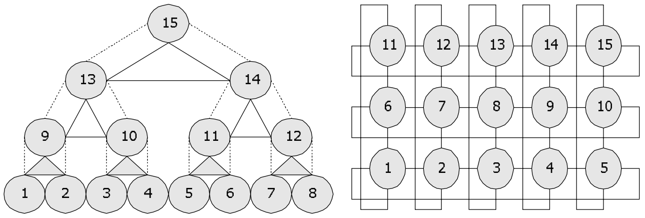
Table 1. Performance Specification

3.7. Network Structures

In a distributed system, the interactions between nodes must be considered. These interactions depend on the logical network structure. We investigate two logical network structures (hierarchical and neighborhood) and design distributed real-time scheduling algorithms based on each network structure. The neighborhood structure (a mesh) has good scaling potential and hence is used for comparison with other well-known distributed scheduling algorithms. The hierarchical structure does not scale well, although there are some situations where a hierarchy is valuable.

Hierarchical feedback control scheduling (H-DFCS) allows a large distributed systems to be broken down into a multiple levels, as illustrated in Figure 4.A. By doing so, only the information that is required to coordinate subsystems needs to be exchanged at higher levels. In the H-

DFCS system, any node that has sub-nodes can be considered a coordinator. The full scheduling algorithm for this system operates every sampling period in the following manner. Each node contains the LFC control system, with the exception of the top node in the hierarchy. This node contains the LFC control system along with the DFC control system. The top node will receive the MR and CPU utilization averaged for the entire system and use this as inputs to its distributed controller to determine the new service level set point for the entire system.



A. Hierarchical B. Neighborhood

Figure 4 Network structures

Load balancing is achieved by migrating tasks between nodes through the network. Each node determines the route by comparing the MR values from the its children, parent and siblings. The MR values are weighted, to describe the number of nodes that they represent. This information is then used to assign a percentage to each entry in the route table, specifying the ratio of the offloaded tasks to be sent along each route. In the hierarchical case that implements a binary tree, there are 4 possible routes from any given node --- two to its two children, one to its parent, and one to its sibling. Routes are unidirectional, and are assigned only if the miss ratio of the other node in question is lower than that of the current node.

In neighborhood feedback control scheduling (N-DFCS) (Figure 4.B) every node contains both a local and distributed controller. This means that a node shares its state information with its direct neighbors and receives state information from these same neighbors. This information that is exchanged is averaged and used to determine the service-level set-point for the node as well as the routes from the node. This prevents a node from being isolated from other nodes, keeps information sharing to local neighbors, and overall creates a more decentralized scheduling system.

Basically, the DFCS is controlling the node based on the state of the node as well as the state of its neighbor nodes. One main difference between the neighborhood solution and the hierarchical solution is that the former works within individual subnets instead of the whole distributed system. The advantage here is that state information is shared in a more decentralized manner.

4. Performance Evaluation

In the first part of our performance evaluation we compare the performance of both hierarchical (HCLOSE) and neighborhood (NCLOSE) feedback control scheduling algorithms with two baseline algorithms: no cooperation and neighborhood open loop.

No Cooperation: Nodes in the system do not interact in any fashion. When a task arrives at a node even if it cannot be scheduled at that node, it remains at that node. This can be considered as a global open loop system without any load balance support. Note: each node still uses local feedback control, but there is no global feedback support.

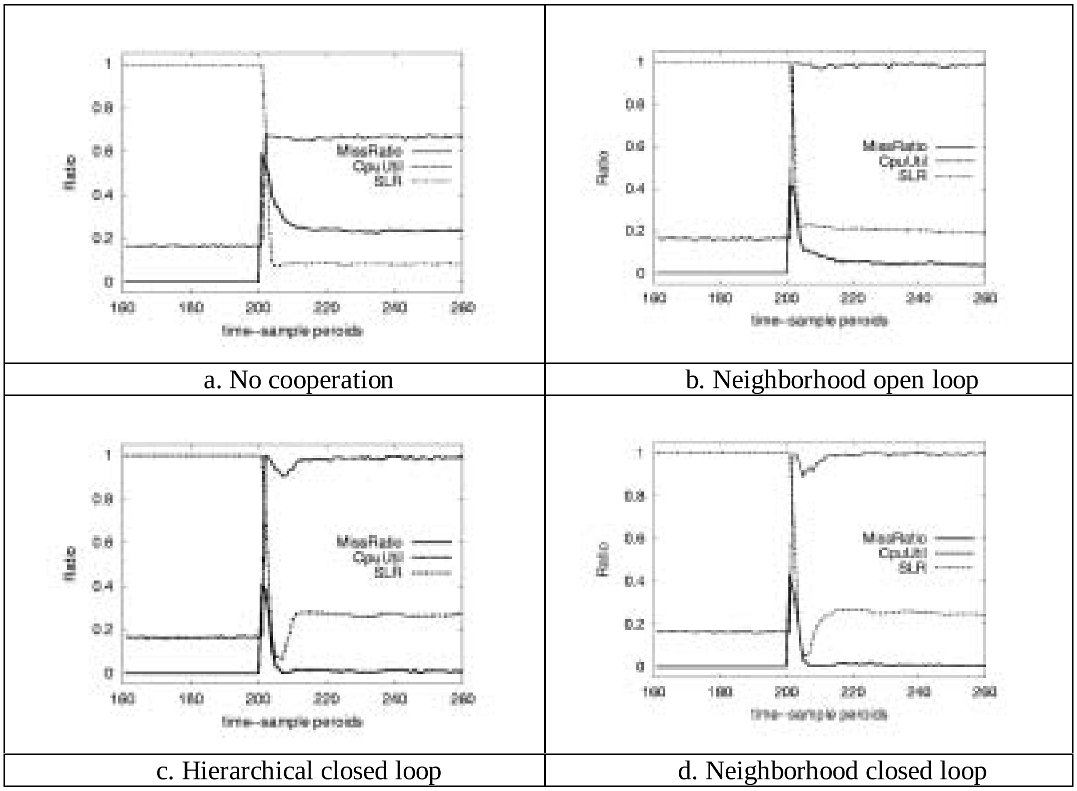
Neighborhood Open Loop: Neighborhood Open Loop (NOPEN) is a second baseline algorithm, in which a node schedules a task using local feedback, if it can. If it cannot be scheduled, it will select a neighbor with a lower miss ratio. If more than one neighbor has a lower miss ratio, the probability of one neighbor to be selected is proportional to the miss ratio difference between the node and this neighbor. Then the task is offloaded to the selected node. If the task cannot be scheduled at the receiving node, it will be again sent to another selected neighbor, until its deadline expires. It can be considered as a global open loop system with load balance support. The load balance method used in NOPEN is the same one used in the feedback control algorithms so that performance gains are not coming from different load balancing strategies, but from the distributed controllers.

4.1. Experimental Setup

For the experiments, we set up 15 nodes as shown previously in Figure 4 with the network delay modeled as one fifth of the average task period. Each task has two different service levels (best and half), which indicates its CPU requirement. The system is initialized with 100 tasks. This represents ~16% load if all tasks use their highest (best) service level (it represents a ~8% if all task use half of their service request). At time 200, a step load is applied with 1000 tasks, which represent ~160% load if all tasks use their highest service level (80% if half). To test the system reaction to unbalanced workloads, the extra workload is applied uniformly to the first 8 nodes (0~7) and the remaining 7 nodes (8~14) don't accept any extra workload directly from the outside. Of course they can accept tasks offloaded by other busy nodes throughout the network.

Workload Model

In our experiments, each task has just 2 different logical versions. A task is described by a tuple *(P,$WCET_i$, $BCET_i$,EET_i ,AET_i)*, where i is either 1 or 0 corresponding to the higher or lower service levels, respectively. *P* is the period (the deadline of each task instance equals its period), *WCET* is the worst-case execution time, *BCET* is the best-case execution time, *EET* is the estimated execution time and *AET* is the average execution time.



a. No cooperation

b. Neighborhood open loop

c. Hierarchical closed loop

d. Neighborhood closed loop

Figure 6 Comparison with baseline algorithms using step workload

The actual execution time is computed as a uniform random variable in the interval [AET_i , $WCET_i$] or [$BCET_i$, AET_i], depending on a random *Bernoulli trial* with probability *(AET_i-$BCET_i$)/($WCET_i$-$BCET_i$).* The following equations describe each tuple value as they are used in this experiment:

$$BCET_i = uniform(1{,}10) \qquad WCET_i = 4 \times BCET_i$$

$$EET_i = (WCET_i + BCET_i) \times 0.5 \qquad WCET_1 = 2 \times WCET_0$$

$$P = WCET_1 \times uniform(5{,}50) \qquad AET_i = EET_i \times ETF$$

This workload model is a very general one. For example, task execution times vary from 1 to 40 time units; task periods vary from 20 to 2000 time units. Here ETF is the execution time factor and can be tuned to vary the accuracy of the estimation of the average execution time. For example, an ETF of 1.0 means that the average execution time that is reported to the scheduler will be the average execution time of the task. If the ETF is less than 1.0, this means that the estimation is less than the actual average execution time of the task. If the ETF is greater than 1.0, then the estimation is greater than the actual average execution time of the task. By varying this factor we can determine how the system will react to a pessimistic or optimistic estimation of the required execution time for the tasks in the system. It is important to note that while the workload is described with a $WCET$, the local and distributed controllers have no knowledge of their values! In these experiments, all tests were run 20 times. However, the graphs represent the trace of single run since it shows how the system acts over time. The long term average for CPU utilization and miss ratio were extremely tight (within 1% of the mean) using a 90% confidence interval.

4.2. Performance Comparison

From the results shown in Figure 6, at time 200 all algorithms react to the incoming step load very quickly, because each node has its own local feedback control system, which was previously shown to be very effective in unpredictable situations [10][12]. But we can see that without global feedback both baseline algorithms do not perform well in terms of transient and steady state response. For example, when the step load occurs at time 200, the miss ratio peaks at almost 60% in no cooperation case (Figure 6.a), compared to only 40% for the other three algorithms. Also, for the no cooperation system while it does adapt to the step load, the local controllers do not reach a zero miss ratio after the step load. This is because there is no load sharing between nodes and the tasks have to be executed at the node where it was originally sent. Furthermore, since the loads are not shared between nodes, the CPU resource is not effectively utilized. For example, as shown in Figure 6.a, it turns out that the average CPU utilization is relatively low (around 67%) even when there is a non-zero miss rate (around 22%). The second baseline, the neighborhood open loop, provides a way to share load between nodes in the sense that the tasks that cannot be scheduled will migrate to other nodes. However, each node does not adjust its service level based on the load of other nodes. The node with small number of local tasks will set its service level relatively high, and accept offloaded tasks from overloaded nodes until it reaches 100% utilization. After that, without global feedback control, this node doesn't lower its service level to further accept offloaded tasks. As a result, although NOPEN exploits the computational power of all nodes, NOPEN can't balance the service level between each node, which leads to a higher miss rate (about 8% in Figure 6.b) than found in the closed loop solutions (Figures 6.c and 6.d) where the miss ratio is between 0-1%. Also, the graphs show that the reaction time to the sudden overload (step load) is sluggish in the open loop case in that it takes a relatively long time (about 12 sampling periods) for the system to reach steady state. On the other hand, both closed loop algorithms show satisfactory response in terms of CPU utilization and miss ratio. After the step load at time 200, the local closed loop controller adjusts its output to reduce the miss ratio according to the difference between the actual miss ratio and the miss ratio set point. The global closed loop controller adjusts the global service level set point according to the busyness of the whole system in the HCLOSE case and the busyness of the subsystem in the NCLOSE case. Through the synergy between the local and distributed controllers, the miss ratio drops to zero and the CPU utilization quickly approaches 100% to provide the best service to the tasks. Also, because of the global service level set point, the whole distributed system behaves as a single node to the user, which is preferred in many practical applications.

The above experiments were performed with $ETF = 1$, which means the average execution time of each task is the same as the estimated execution time. Of course, the actual execution time of each task could vary over time. In reality, it is usually difficult to get the accurate estimation of the execution time because of the uncertainty and unpredictability of the environment. Therefore, we have also run the above comparisons for different ETF values. The result further dramatizes the performance difference between the global closed loop case and the two baselines. For example if $ETF = 1.25$ it takes NOPEN about 40 sampling period to settle down, but it only takes NCLOSE about 12 sampling periods. Due to space limitations, we only provide the *ETF*=1 case here. We will show the impact of imprecise execution time estimation (i.e., varying *ETF*) in the next section when comparing our solution to other algorithms in the literature.

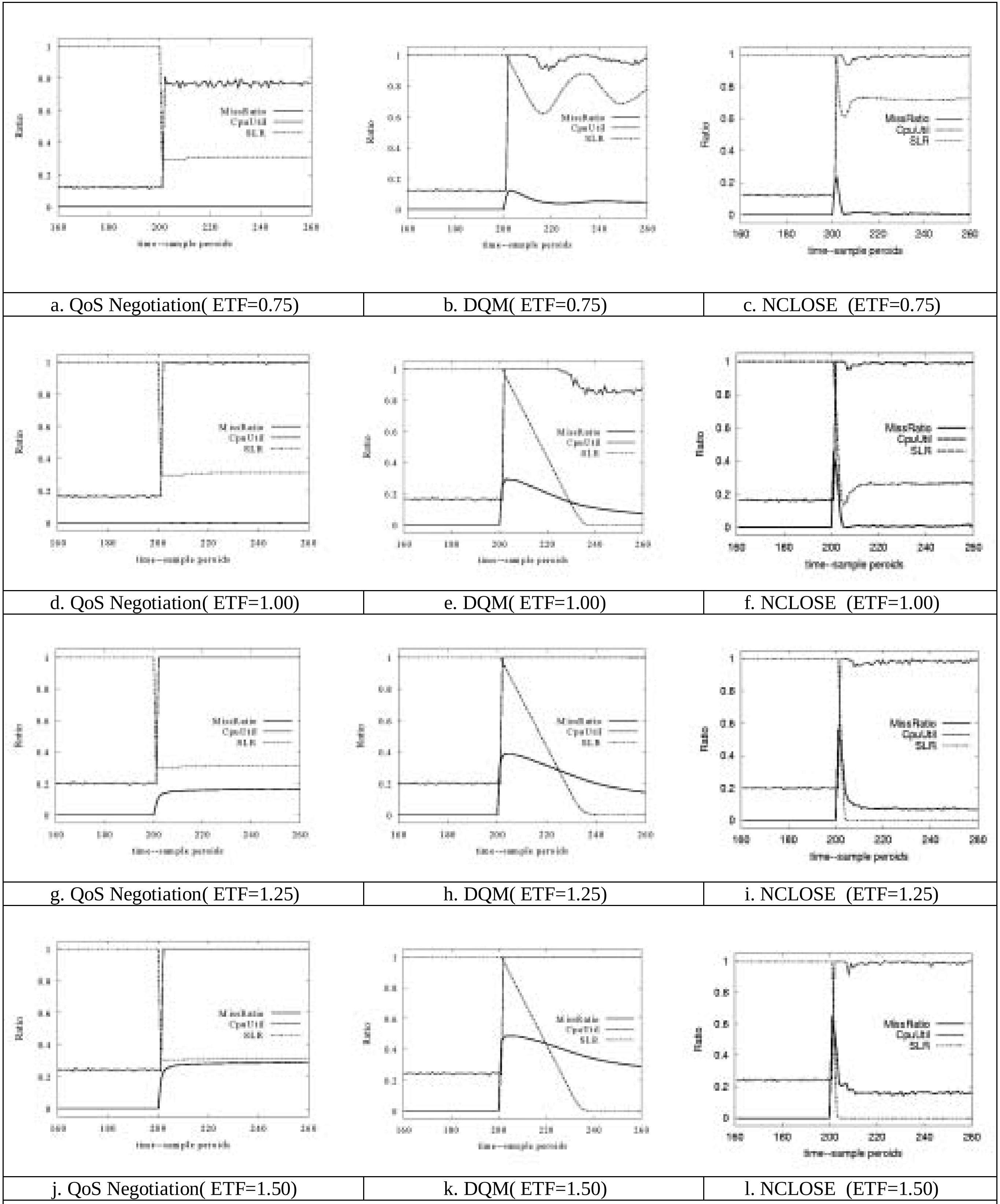
In summary, these experiments demonstrate that feedback control scheduling is a feasible solution for distributed real-time systems. Compared with other baseline algorithms, it can reach zero miss ratio at steady state, effectively share the load between nodes, yield high CPU utilization, and promptly react to load change.

5. Performance Evaluation with Previous Known Algorithms

To further evaluate the performance of DFCS, we compare NCLOSE with two other well-known scheduling algorithms, QoS negotiation [1] and Dynamic QoS Management (DQM)[4]. We first present a brief summary of these two algorithms.

QoS Negotiation: The task model in QoS negotiation is similar to our current task model. A distributed system is assumed in which tasks can arrive at any node. There are several QoS levels for each task. These levels are specified upon task arrival as alternative acceptable performance levels for the task. Each QoS level has different resource requirements and a corresponding benefit (called reward) of executing the task at that QoS level. A node can accept a task in which case a QoS contract is said to be signed which promise to execute the task at one of its specified QoS levels. Alternatively, the task may be rejected in which case no contract is signed. There is a penalty if the task is rejected, and a different (higher) penalty if the QoS contract is violated. The latter allows a node to break the contract unilaterally and offer a compensation. The overall objective of the service is to schedule the tasks to yield a maximum global reward taking into account rejection and QoS violation penalties. At each node, there is a local scheduler, which uses a greedy hill-climbing heuristic to schedule the tasks for maximum reward. The heuristic upgrades the task with the largest reward differential between QoS levels when the system is underutilized, and downgrades the task with the smallest reward differential when the system is overloaded. Underutilization is measured by idle time, while overload is measured by deadline misses. When tasks execute at a level that is lower than the maximum QoS level declared in their contract, the system is said to have an unfulfilled potential reward (UPR). A global scheduler runs periodically to migrate tasks from nodes with high UPR to nodes with low UPR to balance the load and gain higher global reward. A hysteresis is used to prevent oscillations in the load balancing processes. The hysteresis reflects the cost of task migration.

Distributed Dynamic QoS Management: In DQM, each task is associated with a different level of QoS and there are benefits and estimated CPU utilizations for each level. The aim of DQM is to adjust the QoS level of tasks in order to reduce miss ratio and maintain high CPU utilization. More specifically, in the situation of system overload (which is tested by the miss ratio, i.e., if the miss ratio is high, then the system is overloaded), DQM will lower the level of a task's QoS, which has the highest CPU utilization/benefit. In the case of underutilization (which is tested by the percentage of CPU idle cycles), DQM will raise the level of a task's QoS, which has the lowest CPU utilization/benefit. There are two thresholds associated with the algorithm: one for miss ratio and the other for



a. QoS Negotiation(ETF=0.75)

b. DQM(ETF=0.75)

c. NCLOSE (ETF=0.75)

d. QoS Negotiation(ETF=1.00)

e. DQM(ETF=1.00)

f. NCLOSE (ETF=1.00)

g. QoS Negotiation(ETF=1.25)

h. DQM(ETF=1.25)

i. NCLOSE (ETF=1.25)

j. QoS Negotiation(ETF=1.50)

k. DQM(ETF=1.50)

l. NCLOSE (ETF=1.50)

Figure 7 System Performance Comparison

CPU utilization. Once the miss ratio is higher than the threshold, the DQM will lower the service level. If the CPU utilization is lower than the threshold, DQM will increase the service level of the system. This is actually a kind of heuristic feedback. The initial algorithm of DQM was designed for a centralized system. To neutralize this limitation, we optimally distributed the total incoming load to N distributed DQM nodes. Because of the way we distribute the load, each DQM node can act independently. Experimental Setup

In most cases, the same experimental setup as that in Section 4 is used. One important variation in this experiment is to use different values of *ETF* to cover different cases of execution time estimation. As we discussed before, it is usually difficult to provide an accurate estimation of the actual execution time beforehand because of many unknown factors. To make a more realistic comparison, the experiments have evaluated the performance with different estimations:

$$Avg.\ Execution\ time = ETF \times Estimated\ Execution\ time$$

In these experiments, we run the simulation for each algorithm with ETF = 0.75, 1.00, 1.25 and 1.50, to cover the cases of underestimation and overestimation. In the experiment, all tests were run 20 times. However, the graphs represent the trace of single run since it shows how the system acts over time. The long term average for CPU utilization and miss ratio were extremely tight (within 1% of the mean) using a 90% confidence interval.

As shown in Figures 7.d and 7.f, QoS negotiation outperforms NCLOSE by 10% when the estimation is accurate. This should be the case because QoS negotiation is a well-tuned algorithm under the assumption that task execution time are known exactly. Note that its performance depends heavily on the accuracy of the estimation of the execution time. In many real systems it is usually difficult to provide the accurate estimation beforehand.

When the actual execution time is overestimated ($ETF = 0.75$ in Figure 7.a), the QoS negotiation algorithm makes conservative decisions, resulting in lower CPU utilization (76%) and provides low average service level (0.31), compared to 100% CPU utilization and 0.71 average service level of our NCLOSE algorithm ($ETF = 0.75$ in Fig 7.c).

When the actual execution time is underestimated ($ETF > 1$), the performance of QoS negotiation becomes even worse. This is because the scheduling decision is based on inaccurate estimations. In the underestimated situation, the estimated execution time is less that the actual execution time. In that case, the scheduler makes poor decisions by requiring less resource than actually needed by the tasks, causing them to miss their deadlines. The deadline miss ratio reaches 18% for *ETF = 1.25* in Fig 7.g and 28% for *ETF = 1.50* in Fig 7.j, respectively. This is mainly because the algorithm is an open loop algorithm and hence it lacks the mechanism to dynamically adjust its decision according to the actual outcome from the system.

The results also show several deficiencies of DQM. DQM cannot effectively eliminate the error, (i.e., reaching zero miss ratio error) at steady state (Fig 7.b, 7.e, 7.h, 7.k). The problem is that unlike the feedback control in NCLOSE, there is no effective way for DQM to minimize the miss ratio. DQM will lower the service level once the miss ratio is higher than the threshold. One way to lower the error is to choose a smaller threshold. The problem is that if the threshold is very small, the system becomes excessively sensitive to disturbances and becomes unstable. A similar problem exists with the CPU utilization threshold. Furthermore, DQM cannot effectively handle the step load and its reaction to the load change is sluggish. For example Figure 7.e, after 60 sampling periods, it still has 8% miss ratio. Unlike feedback control, which can output a larger control signal to eliminate the error more quickly when the error is large, DQM responds to high miss ratio according to the heuristic, i.e., lower the service level with the highest CPU utilization/benefit. This leads to the slow response, especially when the miss ratio is high. When the estimation is inaccurate, the performance of DQM is even worse, both in transient and steady-state response. The source of these problems is that DQM is a heuristic feedback control algorithm, which adjust the system heuristically based on the output. Without a theoretical basis, such an approach can be inefficient and difficult to tune.

The results show that the feedback-control based approach is very effective. Although the scheduler depends on the estimation to make a decision, the controller can adapt to the inaccuracy of the estimation by monitoring the actual output and adjusting the system according to the difference between the set point and the actual output. When the miss ratio is higher than the set point after the step load, it can adjust service level and therefore reduce the miss ratio. This is also true for the CPU utilization control. This gives NCLOSE the advantage of working in uncertain and unpredictable situations. Furthermore, as we showed in Section 4, based on system modeling, we performed systematic design based on control theory. Given the performance specification such as steady-state error and settling time in Table 1, we designed the controller that satisfies the specification. Through this approach, we have theoretically and experimentally proved the effectiveness of our DFCS framework.

6. Related work

Recently, a new generation of adaptive architectures has emerged. This architecture differs in two respects from early adaptive approaches. First, the performance of the adaptive system is modeled in some coarse-grained manner that represents the relation between aggregate QoS and aggregate resource consumption. This is as opposed to fine-grained models that require knowledge of individual task execution times. Second, feedback was used as a primary mechanism to adjust resource allocation in the absence of *a priori* knowledge of resource supply and demand. This is in contrast to early optimization-based QoS adaptation techniques that assumed accurate models of application resource requirements. Examples of such approaches include [4][7]. In [7], a transaction scheduler called AED monitors the system deadline miss ratio and adjusts task priorities to improve the performance of EDF in overload. The DQM algorithm [4] features a feedback mechanism that changes task QoS levels according to the sampled CPU utilization or deadline misses. These algorithms are based on heuristics rather than theoretical foundations. A particularly interesting approach that belongs in this category is one that uses feedback control theory as the underlying analytical foundation for building feedback-based adaptive systems. This approach uses difference equations to model the aggregate behavior of the system. These equations are then used to design feedback loops with guaranteed properties on both steady state and transient behavior such as speed of convergence. This approach has been adopted by several researchers. For example, Li and Nahrstedt presented a fuzzy-controller on a sender node that dynamically adjusts the sending rate in a distributed visual tracking system [9]. In [5], Buttazzo et.al. presents an elastic task model for adaptive rate control. In [13] integrated control analysis is presented. Hollot et. al. [8] presents control analysis of a congestion control algorithm in Internet routers. In [7], Gandhi et. al. designs a feedback controller to control the queue length of a Locus e-mail server. Eker [6] proposed to integrate on-line CPU scheduling with control performance in embedded control systems. Abdelzaher et. al. [2][3] developed a web server architecture that guarantees desired server utilization under HTTP 1.0. Lu et. al.[10] presented a new feedback architecture to provide relative and absolute delay guarantees in HTTP 1.1 servers. In [11][12], feedback control real-time scheduling algorithms are developed to achieve deadline miss ratio guarantees in uni-processor systems. This paper generalizes the control-theoretical QoS-adaptation approach to distributed systems. Unlike the above solutions that deal with a single controller on a single node, we develop models and algorithms for a group of decentralized controllers that controls the aggregate performance of a distributed system (such as networked embedded systems in smart spaces).

7. Conclusions

We have developed an effective distributed real-time scheduling approach based on feedback control. We have shown that the algorithm is stable and meets transient performance requirements. The algorithm is based both on a novel analytical model and employing standard feedback control design techniques using that model. A performance evaluation study has demonstrated that the algorithm outperforms both baselines and important previous algorithms from the literature in uncertain environments.

8. References

[1] T. F. Adbelzaher, E. Atkins and K. Shin. QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control. *IEEE transaction on Computers*, 49 (11), Nov. 2000.

[2] T. F. Abdelzaher and N. Bhatti, Adaptive Content Delivery for Web Server QoS. *International Workshop on Quality of Service*, London, UK, June 1999.

[3] T. F. Abdelzaher and K.G. Shin. QoS Provisioning with qContracts in Web and Multimedia Servers. IEEE Real-Time Systems Symposium, Phoenix, Arizona, December 1999.

[4] S.Brandt and G.Nutt. A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage. In Real-Time Systems Symposium, pages 307--317, Madrid, Spain,December 1998.

[5] G. Buttazzo, G. Lipari, and L. Abeni, Elastic Task Model for Adaptive Rate Control. IEEE Real-Time Systems Symposium, Madrid, Spain, pp. 286-295, December 1998.

[6] J. Eker. Flexible Embedded Control Systems-Design and Implementation. *PhD-thesis*, Lund Institute of Technology, Dec 1999.

[7] J. R. Haritsa, M. Livny and M. J. Carey. Earliest Deadline Scheduling for Real-Time Database Systems. IEEE Real-Time Systems Symposium, 1991.

[8] C. V. Hollot, V. Misra, D. Towsley, and W. Gong. A Control Theoretic Analysis of RED. IEEE INFOCOM, Anchorage, Alaska, April 2001.

[9] B. Li and K. Nahrstedt. A Control-based Middleware Framework for Quality of Service Adaptations. IEEE Journal of Selected Areas in Communication, Special Issue on Service Enabling Platforms, 17(9), Sept. 1999.

[10] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son.. A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers. IEEE Real-Time Technology and Applications Symposium, Taipei, Taiwan, June 2001.

[11] C.Lu, J.A. Stankovic, G.Tao, and S.H. Son. The Design and Evaluation of a Feedback Control EDF Scheduling Algorithm. In IEEE Real-Time Systems Symposium, Phoenix, Arizona, December 1999.

[12] C.Lu, J.A. Stankovic, T.Abdelzaher, G.Tao, S.H. Son, and M. Marley. Performance Specification and Metrics for Adaptive Real-time Systems. In IEEE Real-Time Systems Symposium, Orlando, Florida, November 2000.

[13] L. Palopoli, L. Abeni, F. Conticelli, M. D. Natale, and G.Buttazzo. Real-Time Control System Analysis: An Integrated Approach. IEEE Real-Time Systems Symposium, Orlando, FL, Dec 2000.

[14] J.Stankovic. Decentralized Decision Making for Task Reallocation in a Hard Real-time System. *IEEE Transactions on Computers*, 38(3):341--355, March 1989.