Feedback Control Real-Time Scheduling in ORB Middleware

Chenyang Lu Xiaorui Wang Christopher Gill
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130-4899
{lu, wang, cdgill}@cse.wustl.edu

Abstract

Existing real-time ORB middleware standards such as RT-CORBA do not adequately address the challenges of 1) providing robust performance guarantees portably across different platforms, and 2) managing unpredictable workload. To overcome this limitation, we have developed software called FCS/nORB that integrates a Feedback Control real-time Scheduling (FCS) service with the nORB small-footprint real-time ORB designed for networked embedded systems. FCS/nORB features feedback control loops that provide real-time performance guarantees by automatically adjusting the rate of remote method invocations transparently to an application. FCS/nORB thus enables real-time applications to be truly portable in terms of real-time performance as well as functionality, without the need for hand tuning. This paper presents the design, implementation, and evaluation of FCS/nORB. Our extensive experiments on a Linux testbed demonstrate that FCS can provide deadline miss ratio and utilization guarantees in face of changes in the platform and task execution times, while introducing a small amount of overhead.

1. Introduction

Object Request Broker (ORB) middleware has shown promise in meeting the functional and real-time performance requirements of distributed real-time and embedded (DRE) systems built using common-off-the-shelf (COTS) hardware and software. DRE systems such as avionics mission computing, flight control systems, and autonomous aerial surveillance increasingly rely on real-time ORB middleware to meet challenging requirements such as communication and processing timeliness among distributed application components.

Several kinds of middleware are emerging as fundamental building blocks for these kinds of systems. Low-level frameworks such as ACE [20] provide portability across different operating systems and hardware platforms. Resource management frameworks such as Kokyu [6] use low-level elements to configure scheduling and dispatching mechanisms in higher-level middleware. Real-Time ORBs such as TAO [21] and nORB [23] are geared toward providing predictable timing of end-to-end method invocations. ORB services such as the TAO Real-Time Event Service [9] and TAO Scheduling Service [6] offer higher-level services for managing functional and real-time properties of interactions between application components. Finally, higher-level middleware services such as RTARM [10] and QuO [29] provide integration of real-time resource management in complex vertically layered DRE applications.

However, before it can fully deliver its promise, ORB middleware still faces two key challenges.

- *Provide real-time performance portability*: A key advantage of middleware is supporting portability on different OS and hardware platforms. However, although the *functionality* of applications running ORB middleware is readily portable, real-time *performance* can differ significantly across different platforms and ORBs. Consequently, an application that meets all of its timing constraints on a particular platform may violate the same constraints on another platform. Significant time and cost must then be incurred to test and re-tune an application for each platform on which it is deployed. Hence DRE applications are *not* strictly portable even when developed using today's ORB middleware. Thus, the lack of robust real-time performance portability detracts from the benefits of deploying DRE applications on current-generation ORB middleware.
- *Handle unpredictable workloads*: The task execution times and resource requirements of many DRE applications are unknown *a priori* or may vary significantly at run time - often because their executions are strongly influenced by the operating environment. For example, the execution time of a visual tracking task may vary dramatically as a function of the number of location of potential targets in a set of received camera images.

A key reason that existing ORB middleware cannot deal with the above challenges is that common sched-

uling approaches are based on *open-loop* algorithms (*e.g.*, Rate Monotonic Scheduling (RMS) or Earliest Deadline First (EDF) [13]) that depend on accurate knowledge of task execution times to provide real-time performance guarantees. However, when workloads and platforms are variable or simply not known *a priori*, open-loop scheduling algorithms either result in extremely underutilized systems based on pessimistic worst-case estimation, or in systems that fail when workloads or platform characteristics vary significantly from design-time expectations.

Our solution is to integrate a *Feedback Control real-time Scheduling* (FCS) framework [17] with ORB middleware, to provide portable real-time performance and robust handling of unpredictable workloads. In contrast to earlier research on real-time scheduling that was concerned with statically assured avoidance of undesirable effects such as overload and deadline misses, FCS algorithms are designed to handle such effects dynamically based on periodic performance feedback. More importantly, FCS algorithms offer an *analytic framework* to provide real-time performance guarantees without underutilizing the system, even when the task execution times are unknown or vary significantly at run-time. While FCS algorithms have been previously analyzed and evaluated through simulations, this paper presents 1) the first incarnation of FCS in a scheduling service for ORB middleware, and 2) the first performance evaluation of FCS on a physical Linux testbed.

FCS/nORB provides key scheduling support that makes DRE software performance portable across OS and hardware platforms and more robust against workload variations when tasks have negotiable QoS parameters that can be adjusted. The FCS service we have implemented in this work automatically adjusts the rates of method invocations on remote application objects, based on measured performance feedback. Our choice of this adaptation mechanism is motivated by the fact that in many DRE applications, *e.g.*, digital feedback control loops [5][22], sensor data display, and video streaming [3], task rates can be adjusted on-line without causing instability or system failure. Other QoS adaptation mechanisms such as online task admission control can also be incorporated easily into the FCS/nORB service.

Specifically, this paper makes three main contributions:

- Design documentation of a FCS service at the ORB middleware layer, that provides real-time performance portability and robust performance guarantees in face of workload variations,
- Implementation of a feedback control loop in an distributed ORB middleware that dynamically adjust the rates of remote method invocations (RMI) transparently to the application (subject to the limitations specified by the application), and
- Results of empirical performance evaluations on a physical testbed that demonstrate the efficiency, robustness and limitations of applying FCS at the ORB middleware layer.

The rest of this paper is structured as follows. We first review previous work on FCS in Section 2. Section 3 describes the design and implementation of our FCS service for nORB. We present results of our performance evaluation on a Linux testbed in Section 4. Section 5 surveys related work in the areas of real-time scheduling, software performance control, and adaptive resource management in middleware. Finally, Section 6 summarizes the contributions of this paper, and describes planned future work.

2. Feedback Control Real-Time Scheduling

Recent research has shown that FCS algorithms can provide performance guarantees in terms of deadline miss ratios and CPU utilization even when actual task execution times are unknown or vary at run time. In this section, we describe abstractly our instantiations of three existing FCS algorithms for the nORB middleware. Details of the algorithms, and their control analyses and simulation evaluations can be found in [17].

2.1. Task Model

We first describe the task model used by our FCS service. With ORB middleware, applications typically execute using remote method invocations on objects distributed across multiple endsystems. Invocation latency for remote methods includes latency on the client, the server, and the communication network. Furthermore, each method invocation may be subject to an end-to-end deadline. An established approach for handling timeliness of remote method invocations is through end-to-end scheduling [14]. In this approach, an end-to-end deadline is divided into intermediate deadlines on the server, client, and communication, and the problem of meeting the end-to-end deadline is transformed to the problem of meeting every intermediate deadline. In this paper, we focus on the problem of meeting intermediate deadlines on the server. We assume that the client and server are not collocated on the same processor. Such a configuration is common in networked digital control applications that run multiple control algorithms on a server processor that interacts with several other client processors attached to

sensors and actuators. Server delays often dominate in DRE systems that are equipped with high-speed network interfaces. Furthermore, our current solution makes progress toward an end-to-end scheduling service to be developed as future work.

In the rest of this paper, we use the term *task* to refer to the execution of a remote method on the server. Specifically, we assume the following task model on a server. Each task T_i is described by the following attributes:

- EE_i: the estimated execution time,
- $[R_{min,i}, R_{max,i}]$: the range of acceptable rates, and
- $R_i(k)$: the rate in the k^{th} sampling period.

We use $X(k)$ to represent the value of a variable X during a sampling period $(k\text{-}1)W$, kW) sec, where k>1, and W is the sampling period length. We assume all tasks are periodic, and each task T_i's relative deadline on the server, $D_i(k)$, equals the period[1], *i.e.*, $D_i(k) = 1/R_i(k)$.

A key property of our task model is that it does not require accurate knowledge of task execution times. The execution time of a task may be significantly different from its estimation and may vary at run time.

2.2. FCS Algorithms

The core of any FCS algorithm is a feedback control loop that periodically monitors and controls its *controlled variables* by adjusting a QoS parameter (*e.g.*, task rate). Candidate controlled variables include the total (CPU) utilization and the (deadline) miss ratio. The *utilization*, $U(k)$, is defined as the percentage of time when the CPU is busy in the k^{th} sampling period. The miss ratio, $M(k)$, is the number of deadline misses divided by the total number of completed tasks in the k^{th} sampling period. *Performance references* represent the desired values of the controlled variables, *i.e.*, the desired miss ratio M_s or the desired utilization U_s. For example, a particular system may require a miss ratio M_s = 1.5% or a utilization U_s = 70%. The goal of an FCS algorithm is to enforce the performance references specified by application, via run-time QoS adaptation.

Three FCS algorithms have been developed based on the choice of different sets of controlled variables. The FC-U and FC-M algorithms each control $U(k)$ or $M(k)$, respectively, and the FC-UM algorithm controls both $U(k)$ and $M(k)$ at the same time.

The feedback control loop in each FCS algorithm is composed of one or more Monitors, a Controller, and a QoS (Rate) Actuator. The Utilization and Miss Ratio Monitors measure the controlled variables, $U(k)$ and $M(k)$, respectively. At the end of each sampling period, the Controller compares the controlled variable with its corresponding performance reference (U_s or M_s), and computes $B(k+1)$, the *total estimated utilization* for the subsequent sampling period. To enforce the total estimated utilization on the server, i.e., $B(k+1) = \sum_i(EE_i * R_i(k+1))$, the Rate Actuator then computes a new set of task rates and instructs each client to adjust its invocation rate accordingly. It is important to note that $B(k)$ can be different from $U(k)$ due to the difference between the estimated and actual task execution times. We now briefly describe the three FCS algorithms, which were previously published in [17].

2.2.1. FC-U

FC-U embodies a feedback loop to enforce a specified utilization. Pseudo code for the FC-U algorithm is shown in Figure 1. FC-U is appropriate for systems with a known (schedulable) utilization bound. In such systems, FC-U can guarantee a zero miss ratio in steady state if U_s is lower than the utilization bound. However, FC-U is not applicable for systems whose utilization bounds are unknown or highly pessimistic.

Invoke in the end of each sampling period
U_s: utilization reference
K_u: utilization controller parameter
1) Get U(k) measured by Utilization Monitor.
2) Utilization Controller computes
   $B(k+1) = B(k) + K_u*(U_s - U(k))$
3) Rate Actuator adjusts task rates.
   a. Compute task rates so that
      $B(k+1) = \sum_i(EE_i * R_i(k+1))$.
   b. Inform clients of the new rates of their tasks.

Figure 1. Pseudo code of FC-U

2.2.2. FC-M

Unlike FC-U, which controls miss ratio indirectly through utilization control, FC-M utilizes a feedback loop *directly* to control the miss ratio[2]. Pseudo code for the FC-M algorithm is shown in Figure 2.

Invoke in the end of each sampling period
M_s: miss ratio reference
K_m: miss ratio controller parameter
1) Get M(k) measured by Miss Ratio Monitor.
2) Miss Ratio Controller computes B(k+1).
   $B(k+1) = B(k) + K_m*(M_s - M(k))$
3) Rate Actuator adjusts task rates (same as FC-U).

Figure 2. Pseudo code of FC-M

[1] The general FCS framework does not have these restrictions. For example, an application of FCS to aperiodic tasks was presented in [17].

[2] FC-M was called FC-EDF in [16] when run with EDF.

Compared with FC-U, the advantage of FC-M is that it does not depend on any knowledge about the utilization bound. It may also achieve a higher CPU utilization than FC-U, whose utilization reference (based on a theoretical utilization bound) is often pessimistic. However, as analyzed in [17], because miss ratio does not indicate the extent of underutilization when $M(k)$=0, FC-M must have a positive miss ratio reference (i.e., $M_s > 0$). Consequently, it will have a small but non-zero miss ratio even in steady state. FC-M is only applicable to soft real-time systems that can tolerate sporadic deadline misses in steady state.

2.2.3. FC-UM

FC-UM integrates miss-ratio control and utilization control to combine their advantages[3]. Pseudo code for the FC-UM algorithm is shown in Figure 3.

Invoke in the end of each sampling period
U_s: utilization reference
M_s: miss ratio reference
K_m: miss ratio controller parameter
K_u: utilization controller parameter
1) Get U(k) and M(k) from Utilization and Miss Ratio Monitors, respectively.
2) Miss Ratio Controller computes
$B(k+1) = B(k)+\min(K_u*(U_s-U(k)),K_m*(M_s-M(k)))$
3) Rate Actuator adjusts task rates (same as FC-U and FC-M).

Figure 3. Pseudo code of FC-UM

The advantage of FC-U is its ability to meet all deadlines ($M(k)$=0) in steady state if the utilization reference is lower than the utilization bound. The advantage of FC-M is that it can achieve a low (but non-zero) miss ratio and higher utilization even when the utilization bound is unknown or pessimistic. Through integrated control, FC-UM aims to achieve the advantages of both the FC-U and FC-M algorithms. In a system with a FC-UM scheduler, the system administrator can simply set the utilization reference U_s to a value that causes no deadline misses in the *nominal* case (e.g., based on system profiling or experience), and set the miss ratio reference M_s according to the application's miss ratio requirement. FC-UM can guarantee zero deadline misses in the nominal case while also guaranteeing that the miss ratio stays close to M_s even if the utilization reference becomes lower than the (unknown) utilization bound of the system.

2.3. Control Analyses

Different from earlier heuristics-based adaptive scheduling techniques, the FCS framework is based on a control theoretic foundation. In [17], we established mathematical models of the scheduling system, and analytically designed and tuned the above FCS algorithms using a control methodology. Control analyses proved that the FCS algorithms can guarantee stability and achieve their performance references (miss ratio or utilization) in steady state even when task execution times are several times of their estimations and vary at run-time. Due to space limitations, the detailed analyses are not repeated in the paper. Readers are referred to [17] for details.

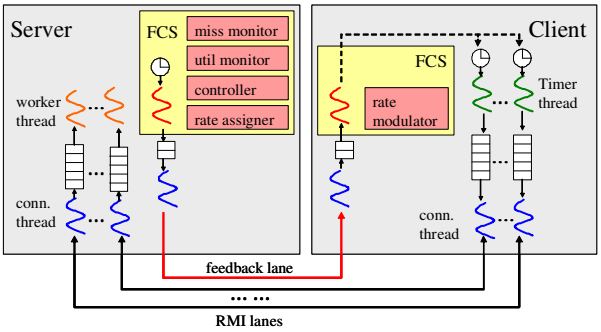


Figure 4: The Architecture of FCS on nORB

3. FCS Architecture on nORB

In this section, we present the architecture of an FCS service that instantiates the FCS algorithms described in Section 2.2, on an ORB called nORB, as illustrated in Figure 4. We first give an overview of our extensions to nORB for use with FCS, and then describe the design and implementation of the FCS service.

3.1. Extensions to nORB for FCS

nORB [23] is a light-weight ORB designed to support networked embedded systems. Both nORB and the new FCS service are based on ACE [20]. nORB currently only supports fixed priority scheduling. To avoid priority inversion at the communication layer, a separate TCP connection called a *lane* [19] is established between a server and a client for each priority level that is used for method invocation requests. The basic priority lane approach in nORB assumes preemptive thread priority enforcement by an underlying real-time operating system, so that RMS properties are preserved using FIFO queuing (*e.g.*, by TCP/IP) in each lane. Further details of nORB are presented in [23].

While FIFO queuing in each static priority lane is sufficient for many applications, to support the continuous adaptation of priorities needed by FCS we had to extend the basic nORB queuing capabilities as shown in Figure 4. An FCS/nORB client has a number of timer threads and connection threads. Each pair of

[3] FC-UM was called FC-EDF[2] in [15] when run with EDF.

timer/connection threads (connected through a buffer) is assigned a priority and submits method invocation requests to the server at this priority. Each timer thread is associated with a timer that generates periodic time-out interrupts to initiate method invocation requests at a specified rate. In this study, we apply the RMS policy [13] to assign task priorities. A basic FCS/nORB server has several worker threads and connection threads. Each pair of worker/connection threads is assigned a priority and is responsible for processing method invocation requests at that priority. Connection threads receive method invocation requests from clients, and worker threads invoke the corresponding methods and send the results back to clients.

3.2. Configuration Interface

Application developers can specify a set of scheduling parameters in a configuration file that is used to initialize the FCS service when the system is started. Configuration parameters include the specific FCS algorithm to run, the performance references, the sampling period, and two parameters, G_A and G_M. According to the analysis in [17], G_A and G_M determine the value of the control parameters and affect the range of platform and workload variability that FCS can handle.

Applications can register their tasks in a task description file. Each task T_i is described by a tuple (EE_i, $R_{min,i}$, $R_{max,i}$) as described in Section 2.1.

3.3. Feedback Control Loop

The FCS service on a server includes a *utilization monitor*, a *miss ratio monitor*, a *controller*, a *rate allocator*, and a pair of FCS/connection threads The FCS service on a client includes a *rate modulator* and a pair of FCS/connection threads. All FCS/connection threads in the FCS service are assigned the highest priority so that the feedback control loop can run in overload conditions, when it is needed most. The FCS/connection threads on the server are connected with each client connection thread through a TCP connection we call a *feedback lane*. We now present the details of each component.

Utilization Monitor: The utilization monitor uses the `/proc/stat` file in Linux to estimate the CPU utilization in each sampling period. The `/proc/stat` file records the number of *jiffies* (each 1/100 of a second) since the system start time, when the CPU is in user mode, user mode with low priority (nice), system mode, and when used by the idle task. At the end of each sampling period, the utilization monitor reads the counters, and estimates CPU utilization by dividing the number of jiffies used by the idle task in the last sampling period by the total number of jiffies in the same period. We note that the same technique is used by the benchmarking tool, NetPerf [18].

Deadline Miss Monitor: The deadline miss monitor measures the percentage of completed tasks that miss their deadlines on the server in each sampling period. FCS/nORB maintains two counters for each pair of connection/worker threads on the server. One counter records the number of completed tasks in the current sampling period, and the other records the number of tasks that missed their deadlines in the same period. Each connection thread timestamps every method invocation request when it arrives from its nORB lane. The worker thread checks whether a completed task has missed its deadline and updates the counters after it sends the invocation result to the client. At the end of each sampling period, the deadline miss monitor aggregates the counters of all worker/connection threads, and computes the deadline miss ratio in the sampling period. Note that FCS/nORB maintains separate counters for each pair of connection/worker threads instead of shared global counters, to reduce contention among threads updating the counters. This use of thread-specific storage is important because contention among worker threads could cause priority inversions.

Controller: The controller implements the control function presented in Section 2.2. Each time its periodically scheduled timer fires, it invokes the utilization and/or deadline miss monitors, computes the total estimated utilization for the next sampling period, and then invokes the rate assigner.

Rate Assigner: The rate assigner on the server and the rate modulator on its clients together serve as actuators in the feedback control loop. The rate assigner computes the new task rates to enforce the total estimated utilization computed by the controller. Different policies can be applied to assign task rates. Our rate assigner currently implements a simple policy that is called *Proportional Rate Adjustment* (PRA) in this paper. Assuming that the initial rate of task T_i is $R_i(0)$, the initial total estimated utilization $B(0) = \sum_i(EE_iR_i(0))$, and the total estimated utilization for the following (k^{th}) sampling period is $B(k)$, the PRA policy assigns the new rate to task T_i as follows:

1) $R_i(k+1) = (B(k+1)/B(0))R_i(0)$
2) if $(R_i(k+1)<R_{min,i})$ $R_i(k+1)=R_{min,i}$
3) else if $(R_i(k+1)>R_{max,i})$ $R_i(k+1)=R_{max,i}$

It can be proven that PRA enforces the total estimated utilization, *i.e.*, $B(k)=\sum_i(EE_iR_i(k))$, in every sampling period if no task rates reach their lower or upper limits.

The PRA policy treats all the tasks "fairly" in the sense that the relative rates among tasks always remain the

same if no tasks reach their rate limits. When an application runs on a faster platform, the rates of all tasks will be increased proportionally. Similarly, when the application runs on a slower platform, the rates of all tasks will be decreased proportionally. A side effect of the PRA policy is that priorities of tasks will not change at run-time under RMS because the relative order of task rates remains the same. This reduces overhead on the clients because they do not need to change task deadlines on the fly. However, since PRA potentially changes the rate of every task in each sampling period, it can introduce relatively high overhead for resetting all the timers on the clients. We are currently developing a light-weight timer management technique to reduce such overhead. This technique has not been integrated with FCS/nORB.

Note that the PRA policy is based on the assumption that all tasks are "equally important". More precisely, it assumes that all tasks' *values* to the application are uniformly proportional to their execution times. When this assumption is not true, the rate assigner needs to optimize the total system value under the constraint of the total estimated utilization. Although the value optimization problem is not a focus of this study, we note that several existing algorithms, *e.g.*, [12], could be used in the rate assigner to address this problem.

Rate Modulator: A Rate Modulator is located on each client. It receives the new rates for its remote method invocation requests from the rate assigner on the server through the feedback lane, and resets the interval of the timer threads whose request rates have been changed.

3.4. Implementation

FCS/nORB 1.0 is implemented in C++ using ACE 5.2.7 on Linux. The entire FCS/nORB middleware (excluding the code in the ACE library and IDL library) is implemented in 7898 lines of C++ code - compared to 4586 lines of code in the original nORB. Both nORB and FCS/nORB are open-source software and can be downloaded from:

- nORB: http://deuce.doc.wustl.edu/nORB/
- FCS/nORB: http://deuce.doc.wustl.edu/FCS_nORB/

4. Empirical Evaluations

In this section, we present the results of three sets of experiments run on a Linux testbed. Experiment I measured the overhead introduced by FCS. Experiment II evaluated the performance portability of applications on FCS/nORB on two different server platforms. On both platforms, we ran the same steady workload whose task execution times were significantly different from their estimations (the same estimations were used in all experiments). Finally, Experiment III stress-tested FCS/nORB's ability to provide robust performance guarantees with a workload whose task execution times varied dramatically at run-time.

4.1. Experimental Set-up

Platform: We performed our experiments on three PCs named **Server A**, **Server B**, and **Client**. Server A and Client were Dell 1.8G Celeron PCs each with 512 MB of RAM. Server A and Client were directly connected with a 100 Mbps crossover Ethernet cable. They both ran Red Hat Linux release 7.3 (Kernel 2.4.19). Server B was a Dell 1.99G Pentium4 PC with 256 MB of RAM. Server B and Client were connected through the CSE departmental 100 Mbps LAN. Server B ran Red Hat Linux release 7.3 (Kernel 2.4.18). Server A and Server B served as servers in separate experiments, while Client served as the only client PC in all experiments.

Workload: The workload used in all experiments comprised 12 tasks on the client that periodically submitted method invocation requests to the server. Each task invoked one of three methods (shown in Table 1) of an application object. All the tasks invoking the same method shared the same maximum rate, but their minimum rates were randomly chosen from a range listed in the "minimum rate" column in Table 1. Since we focus on unpredictable workload and platform portability, the estimated execution times were different from the actual execution times in each experiment. Note that the same estimated execution times are used in all experiments despite the fact that they used different platforms and had different actual task execution times. With FCS, re-profiling of task execution times was *not* needed to provide performance guarantees.

Table 1. Methods invoked by the workload

method	est. exe. time (ms)	min rate	max rate	#task
1	8.4	[1.1,2.1]	35	6
2	1.2	[1.3,1.9]	50	2
3	7.0	[1.2,2.2]	40	4

FCS Configuration: The configuration parameters of FCS are shown in Table 2. To demonstrate the robustness of feedback control, the *same* configuration was used in all experiments even though they were performed on different platforms and tested with different task execution times. The Controller parameters were computed using control theory based on G_A and G_M, which determine the robustness of FCS [17]. The utilization reference of FC-U is chosen to be 70%, slightly lower than the RMS schedulable utilization bound for 12 tasks: $12(2^{1/12}-1) = 71\%$. FC-UM had a higher utilization reference (75%) because of the existence of miss ratio control as we discussed in Section 2.3. As a

ratio control as we discussed in Section 2.3. As a baseline, we also ran experiments under open-loop scheduling (RMS) by turning off the feedback loop. For simplicity, the open-loop baseline is called **OPEN** in the rest of the paper.

Table 2. FCS Configuration in all Experiments

	FC-U	FC-M	FC-UM
reference	U_s=70%	M_s=1.5%	M_s=1.5% U_s=75%
G_A, G_M	G_A=2	G_A=2, G_M=0.447	
sampling period	4 second		

4.2. Experiment I: Overhead Measurement

The feedback control loop for each FCS algorithm introduces overhead. The overhead is caused by several factors including the timer associated with FCS, the utilization and miss ratio monitoring, the control computation in the controller, and the rate calculation and communication overhead with the clients in the rate assigner. FCS is a viable middleware service only if the overhead it introduces is sufficiently low.

To quantify overhead of the FCS algorithms, we compared the average CPU utilization under different scheduling algorithms when the same workload is applied to the system running on Server A. To limit the overhead caused by the utilization monitoring for OPEN and FC-M, average CPU utilizations were measured by setting the sampling period of the utilization monitor to the duration of the entire run, *i.e.*, the utilization monitor is only invoked twice for each run with FC-M and OPEN – once in the beginning of the run, and once at the end of the run. The average CPU utilization of FC-U and FC-UM was measured by averaging the utilization of each sampling period, since they need to execute the utilization monitor periodically. To keep the application workload constant, we disabled the rate modulator on the clients so that all tasks always ran at constant rates.

The results of the overhead measurements are shown in Table 3. Each value in the first row is the mean of average utilization in 8 repeated runs and its 90% confidence interval. Each run lasted for 800 seconds, a total of 200 sampling periods. The second row shows the overhead of each FCS algorithm in terms of CPU utilization (computed by subtracting OPEN's utilization from each FCS algorithm's utilization). The 90% confidence interval of the most efficient algorithm, FC-U, actually overlapped with that of OPEN, which meant that there were no statistical evidence with 90% confidence that FC-U had a higher utilization than OPEN. FC-M and FC-UM, however, had statistically higher utilizations than OPEN. Over a 4 second sampling period, all three FCS algorithms introduced overhead of less than 1% of CPU utilization. We also estimated the execution time of each FCS algorithm by multiplying its utilization overhead with the sampling period (shown in the bottom row of Table 3). The overhead of all FCS algorithms was less than 36 msec per sampling period. FC-U introduced less overhead indicating that the utilization monitor was more efficient than the miss ratio monitor. While the utilization monitor only needs to read the `/proc/stat` file once every sampling period, the miss ratio monitor requires time-stamping every method invocation twice. FC-UM's overhead is slightly less than the summation of the overheads by FC-M and FC-U because FC-UM ran both monitors. These experiments showed that FCS overhead is sufficiently small for a broad range of applications.

Table 3. Results of Overhead Measurement

	OPEN	FC-U	FC-M	FC-UM
util (%)	74.15 ±0.30	74.55 ±0.42	74.70 ±0.10	75.05 ±0.16
overhead (%)		0.40	0.54	0.90
overhead (ms)		15.9	21.7	35.9

4.3. Experiment II: Performance Portability

In Experiment II, the execution time of each task on Server A remained approximately twice of its estimation throughout each run. The purpose of this set of experiments was to evaluate the performance of the FCS algorithms and OPEN when task execution times vary significantly from estimated values, either due to the difference between a new deployment platform and the original platform on which the tasks were profiled, or to inaccuracy in task profiling.

Our first experiment emulates the common engineering practice based on open loop scheduling. We first tuned task rates based on the *estimated* execution times so that the total estimated utilization is 70%. However, when we ran the tasks at the rates according to the predicted rates, the server crashed. This is not surprising: since the estimated execution times were inaccurate, the actual total requested utilization by all nORB threads reached approximately 140% and caused the Linux kernel to freeze. This was because all nORB threads ran at real-time scheduling priorities that are higher than kernel priorities on Linux. When the utilization of nORB threads exceeded 100%, no kernel activities were be able to execute. To avoid this problem, all the tasks would need to be re-profiled for each platform on which the application is deployed. Hence, the open loop approach can cost developers significant time to tune the workload to achieve the same perform-

ance on different platforms. This lack of performance portability is an especially serious problem when there is a large number of potential platforms (*e.g.*, in a product line) or if a potential platform is unknown at development time.

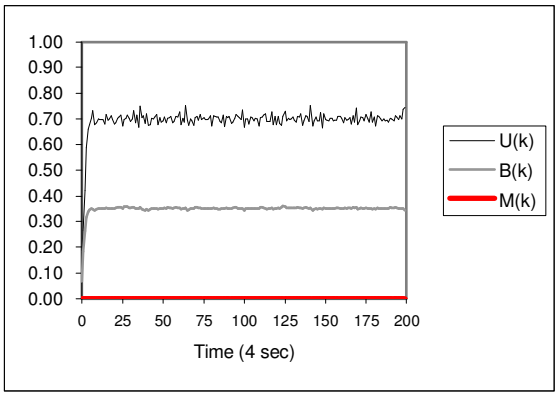
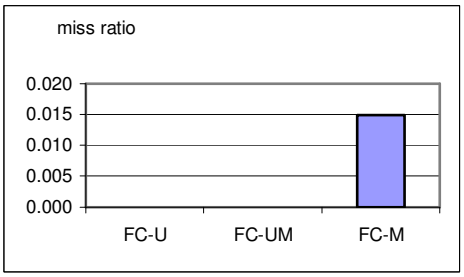


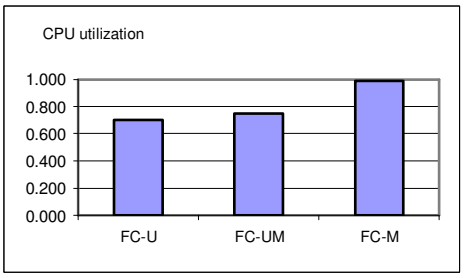
Figure 5: A typical run of FC-U on Server A (note the time unit is the sampling period)

We now examine the experimental result for the FCS algorithms themselves. As an example, the sampled utilization $U(k)$, miss ratio $M(k)$, and the total estimated utilization $B(k)$ computed by the controller in a typical run under FC-U is illustrated in Figure 5. All tasks started from their lowest rates. The feedback control loop gradually increased $U(k)$ by raising task rates (proportional to $B(k)$). At the 5th sampling point, the $U(k)$ reached 67.7% and settled in a steady state around 70%. This result showed that FC-U can *self-tune* task rates to achieve the specified CPU utilization even when task execution times were significantly different from estimated values. The results were consistent with the control analysis presented in [17].

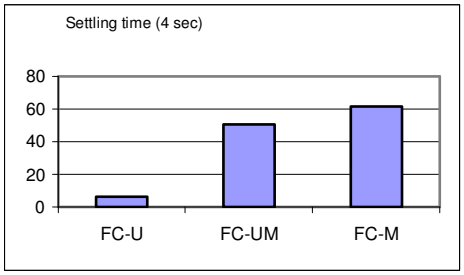
The performance results of FC-U, FC-M, and FC-UM on Server A are summarized in Figures 6(a-c). The performance metrics we used included the miss ratio and utilization in steady state, and the settling time. The *steady-state miss ratio* is defined as the average miss ratio in a steady state. The *steady-state utilization* is similarly defined as the average utilization in a steady state. Both metrics measure the performance of a system after its adaptation process settles down to a steady state. Settling time represents the time it takes the system to settle down to a steady state. The settling time can be viewed as the duration of the self-tuning period after an application is ported to a new platform. It is difficult to determine the precise settling time on a noisy, real system. As an approximation, we considered that FC-U and FC-M entered a steady state at the first sampling instant when $U(k)$ reached $0.99U_s$, and FC-M entered a steady state at the first sampling instant when $U(k)$ reached 99% of the utilization in the last sampling period of the experiment. Each data point in Figures 6(a-c) is the mean of three repeated runs, and each run took 800 seconds. The standard deviations in miss ratio, utilization, and settling time are below 0.01%, 0.03%, and 6.11 sec (i.e., 1.53 sampling period), respectively.



(a) Average steady-state miss ratio



(b) Average steady-state CPU utilization



(c) Average settling time

Figure 6: Performance results of FCS algorithms on Server A in Experiment II

From Figure 6(a), we can see that both FC-U and FC-UM caused no deadline misses in steady-states. FC-M's steady-state miss ratio is 1.49% - compared to the miss ratio reference (1.5%). At the same time, the steady-state utilizations of FC-U and FC-UM are 70.01% (compared to a utilization reference of 70.00%) and 74.97% (compared to a utilization reference of 75%), respectively. The result of FC-UM is because the utilization control dominated in steady state due to the fact that its steady state utilization is lower than the miss ratio control. In contrast, FC-M achieved a higher utilization (98.93%) in the steady state at the cost of a slightly higher miss ratio.

As shown in Figure 6(C), FC-M and FC-UM both had significantly longer settling times than FC-U due to the saturation of miss ratio control in underutilization. This means that FC-M and FC-UM need more self-tuning time before they can reach steady states. Note that the settling times of FC-M and FC-UM are related to the initial task rates. In our experiments, all tasks started from their lowest possible rates in the beginning of the self-tuning phase. The settling times can be reduced by increasing the initial task rates. For example, we may choose the initial rates as the desired rates on the slowest platform in a product line.

To further evaluate the performance portability of FCS/nORB, we re-ran the same experiments on Server B. A typical run of FC-U, FC-UM, and FC-M are shown in Figures 7, 8, and 9, respectively. Each run takes 1200 seconds. As the case on Server A, all the algorithms successfully enforced their utilization or miss ratio references in steady state. The difference is that all tasks ran at a higher rate (proportionally to $B(k)$) on Server B than Server A because Server B is faster than Server A. In addition, all algorithms had longer settling times on Server B than Server A. This is consistent with our control analyses in [17].

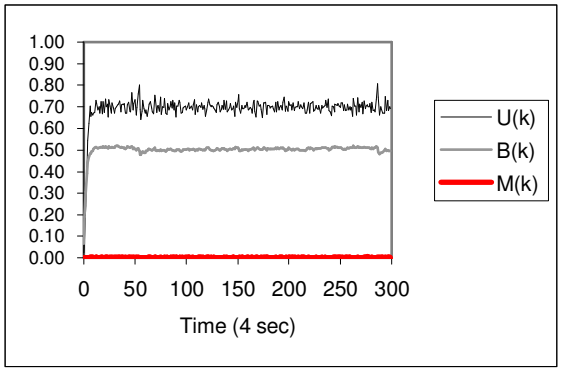


Figure 7: A typical run of FC-U on Server B

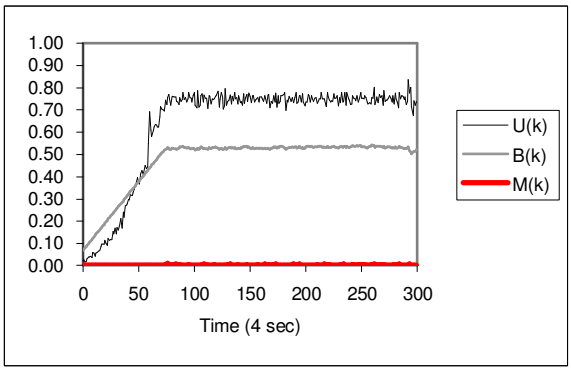


Figure 8: A typical run of FC-UM on Server B

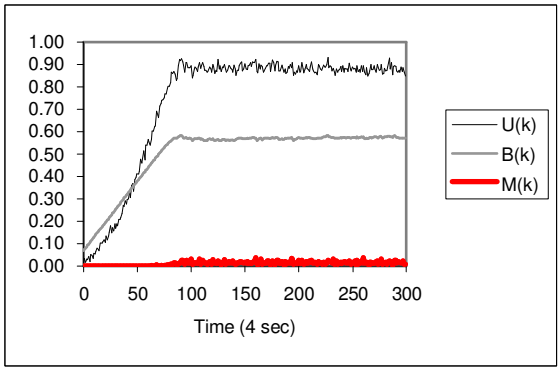


Figure 9: A typical run of FC-M on Server B

In summary, Experiment II demonstrated that FCS/nORB can provide desired utilization or miss ratio even when 1) applications were ported to different platforms and 2) task execution times were significantly different from their estimations. Therefore, FCS/nORB represented a way to perform automatic performance tuning on a new platform.

In addition, we note that a combination of FCS and open-loop scheduling can be used to achieve both self-tuning and run-time efficiency for applications with steady workloads. When an application is ported to a new platform, it is initially scheduled using the FCS algorithm to converge to a steady state with desired performance. Then the feedback control loop can be turned off and the applications can continue to run at the correct rates under open-loop scheduling.

4.4. Experiment III: Varying Workload

In this set of experiments, we evaluate the performance of FCS/nORB and OPEN on Server A when task execution times vary significantly at run-time.

We first study the performance of OPEN. A typical run of OPEN is illustrated in Figure 10. We initially hand tuned the application rates to achieve a utilization of approximately 75%. In the beginning of the 50th sampling period (200 sec), however, the execution times of method 1 (invoked by 6 tasks – see Table 1) suddenly increased, which caused the CPU utilization to reach almost 100% and deadline misses. Note that the system would have crashed (as in Experiment II) should the execution time of method 1 increased slightly more. At 900 sec, the execution time of method 1 suddenly decreased causing the CPU utilization to drop to approximately 40%. This experiment shows that even fine-tuned applications can not always achieve acceptable performance under OPEN. When the actual execution times exceed the initial execution time used for tuning, the system can be overloaded and even crash. On the other hand, if the actual execution times become lower than those used at tuning time, the CPU is

underutilized when task could have run at higher rates (QoS).

In contrast, both FC-U and FC-UM maintained specified CPU utilizations (70% and 75%, respectively) in steady states despite the variations in task execution times (as illustrated in Figures 11 and 12, respectively). Both algorithms effectively adapted task rates (proportionally to $B(k)$) in response to changes of system load. FC-UM had a long settling time in the underutilized condition. However, its settling time is significantly shorter in the overload condition. The short settling time under overload is important because adaptation is much more important in overload conditions than that in underload conditions.

Interestingly, FC-M caused the system to crash when the execution times increased. This is inconsistent with previous OS-level simulation results [17] that showed FC-M could handle such varying workload. This is because that FC-M achieved a high utilization (more than 90%) before the execution time increased at time 200 sec. The utilization then increased to 100% due to the increase in execution times and the system crashed due to kernel starvation.

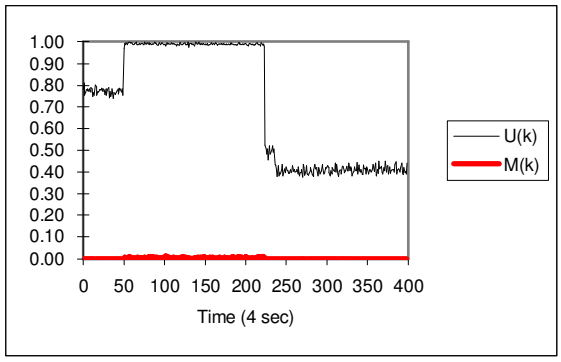


Figure 10: A typical run of OPEN under varying workload

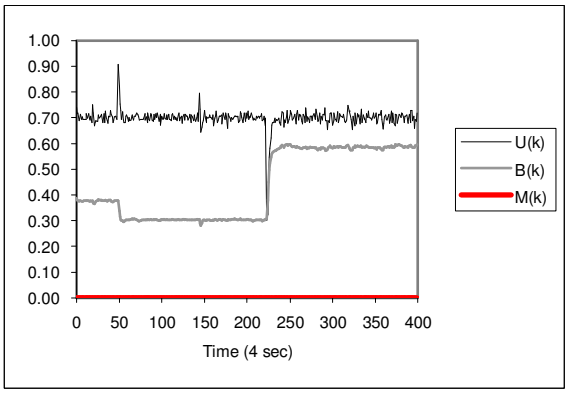


Figure 11: A typical run of FC-U under varying workload

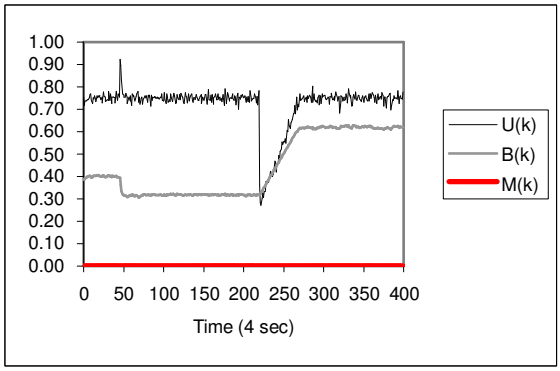


Figure 12: A typical run of FC-UM under varying workload

In general, FCS/nORB cannot handle varying workload that transiently increased the utilization to 100% due to the starvation of the kernel under such conditions. This result shows that limitation of middleware implementation on top of common general purpose operating systems (e.g., Linux, Windows, and Solaris) in which real-time scheduling priorities are higher than kernel priorities. On such platforms, the range of variation in utilization that a FCS algorithm can handle is limited by its steady-state utilization before the variation occurs. For example, with a utilization reference of U_s, FC-U can only handle a utilization increase of no more than (1-U_s) in order to provide robust utilization guarantees. Therefore, the utilization reference of FC-U and FC-UM should consider this *safety margin* in face of varying workload. Since FC-M usually achieves a high utilization and does not have control over its (already small) safety margins, a middleware implementation of FC-M is less appropriate for time varying workload.

In summary, Experiment III demonstrated that, FC-U and FC-UM can provide robust performance guarantees even when task execution times vary (within the aforementioned safety margin) at run-time.

5. Related Work

The control theoretic approach has been applied to various computing and networking systems. A survey of feedback performance control for software services is presented in [1]. Recent research on applying control theory to real-time scheduling is directly related to this paper. For example, Steere, *et al.*, developed a feedback based CPU scheduler [25] that coordinated the CPU allocation to consumer and supplier threads, to guarantee the fill level of buffers. Abeni, *et al.*, presented analysis of a reservation-based feedback scheduler in [2]. In our previous work [15][16][17], a Feedback Control real-time Scheduling framework and algorithms were developed to provide deadline miss ratio and utilization guarantees for real-time applications

with unknown task execution times. Feedback control real-time scheduling has also been extended to handle distributed systems [24]. The difference between the work presented in this paper and the related work is that we describe the design and evaluation of a FCS service at the ORB middleware layer, while the related work is based either on simulation or kernel level implementation. ORB middleware is an attractive vehicle to implement FCS because of its ability to support portable real-time applications on COTS platforms.

Another project that is closely related to FCS/nORB is ControlWare [28], which is also an incarnation of software performance control at the middleware layer. The difference is that ControlWare embodies adaptation mechanisms (such as server process allocation in the Apache server) that are tailored for Quality of Service provisioning on Internet servers, while FCS/nORB integrates Feedback Control real-time Scheduling with remote method invocation mechanisms for distributed real-time embedded systems.

WSOA [4] is a large-scale demonstration of adaptive resource management at multiple architectural levels in a realistic distributed avionics mission computing environment. The WSOA application is in essence a networked ad hoc control system, with adaptation of image tile compression to meet download deadlines. Our work integrating FCS with nORB dispatching seeks to add rigor to middleware-based resource management by applying control theory *within the middleware itself*. In doing so, we seek to complement other middleware projects for DRE systems, and increase the capabilities offered by DRE middleware as a whole.

6. Conclusions and Future Work

In summary, we have designed and implemented a Feedback Control real-time Scheduling service on an ORB middleware for distributed real-time embedded systems. Performance evaluation on a physical testbed has shown that 1) FCS/nORB can guarantee specified miss ratio and CPU utilization even when task execution times deviate significantly from the estimation and change at run-time; 2) FCS/nORB can provide the same performance guarantees on platforms with different processing capabilities; and 3) the middleware layer instantiation of performance control loops only introduces a small amount of processing overhead on the server. These results demonstrate that a combination of FCS and ORB middleware is a promising approach to achieve robust real-time performance guarantees and performance portability for DRE applications. In the future, we will develop a distributed FCS service on an ORB middleware that provide end-to-end delay guarantees. We will also investigate and integrate other adaptation mechanisms such as task reallocation into the FCS service, and will integrate our work on FCS with our other research on multi-level scheduling aspects [7] and QoS-aware component middleware [26].

7. References

[1] T.F. Abdelzaher, J.A. Stankovic, C. Lu, R. Zhang, and Y. Lu, "Feedback Performance Control in Software Services," *IEEE Control Systems*, to appear in 2003.

[2] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, "Analysis of a Reservation-Based Feedback Scheduler," *IEEE Real-Time Systems Symposium (RTSS 2002)*, Dec 2002.

[3] S. Brandt and G. Nutt, "A Dynamic Quality of Service Middleware Agent for Mediating Application Resource Usage," *IEEE Real-Time Systems Symposium*, December 1998.

[4] D. Corman, "WSOA - Weapon Systems Open Architecture Demonstration - Using Emerging Open System Architecture Standards to Enable Innovative Techniques for Time Critical Target Prosecution", *IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Daytona Beach, FL, October 2001.

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

[6] C.D. Gill, D.L. Levine and D.C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems Journal, Special Issues on Real-Time Middleware*, Vol. 20, No. 2, March 2001

[7] C.D. Gill, D. Niehaus, L. DiPippo, V.F. Wolfe, and V. Subramonian, "Resource Rationalizer: A Pattern Language for Multi-Level Scheduling", 8th Pattern Languages of Programming Conference, Allerton Park, Illinois, USA, 8-12 September 2002.

[8] C.D. Gill and V. Subramonian, J. Parsons H.-M. Huang, S. Torri, D. Niehaus and D. Stuart, "ORB Middleware Evolution for Networked Embedded Systems", *8th International Workshop on Object Oriented Real-time Dependable Systems (WORDS)*, January 2003.

[9] T.H. Harrison, D.L. Levine, and D.C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service", *OOPSLA '97*, Atlanta, GA, October 1997.

[10] J. Huang, Y. Wang, and F. Cao, "On developing distributed middleware services for QoS- and criticality-based resource negotiation and adaptation", *Real-Time Systems Journal*, *Special Issue on Operating Systems and Services*, 16(2): 187-221; May 1999.

[11] D.A. Karr, C. Rodrigues, Y. Krishnamurthy, I. Pyarali, and D.C. Schmidt, "Application of the QuO Quality-of-Service Framework to a Distributed Video Application", *3rd International Symposium on Distributed Objects and Applications*, September 2001, Rome, Italy.

[12] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar and J. Hansen, "A Scalable Solution to the Multi-Resource QoS Problem," *IEEE Real-Time Systems Symposium*, December 1999.

[13] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," Journal of ACM, Vol. 20, No. 1, pp. 46-61, 1973.

[14] J.W.S Liu, *Real-Time Systems*, Prentice Hall, 2000.

[15] C. Lu, J.A. Stankovic, T.F. Abdelzaher, G. Tao, S.H. Son, and M. Marley, "Performance Specifications and Metrics for Adaptive Real-Time Systems," IEEE Real-Time Systems Symposium (RTSS 2000), December 2000.

[16] C. Lu, J.A. Stankovic, G. Tao, and S.H. Son, "Design and Evaluation of a Feedback Control EDF Scheduling Algorithm," *20th IEEE Real-Time Systems Symposium (RTSS 1999)*, Phoenix, AZ, December 1999.

[17] C. Lu, J.A. Stankovic, G. Tao, and S.H. Son, "Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms," *Real-Time Systems Journal*, *Special Issue on Control-theoretical Approaches to Real-Time Computing*, 23(1/2): 85-126, July/September 2002.

[18] Public Netperf Homepage, http://www.netperf.org.

[19] I. Pyarali, D. C. Schmidt, and R. Cytron, "Techniques for Enhancing Real-time CORBA Quality of Service", *IEEE Proceedings Special Issue on Real-time Systems*, May 2003.

[20] D.C. Schmidt, "The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software", *12th Annual Sun Users Group Conference*, December 1993

[21] D.C. Schmidt *et. al*, "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems", *IEEE Distributed Systems Online*, 3(2), February 2002. http://dsonline.computer.org/middleware

[22] D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin, "On Task Schedulability in Real-Time Control Systems", *IEEE Real-Time Systems Symposium*, December 1996.

[23] V. Subramonian, G. Xing, C.D. Gill, and R. Cytron, "The Design and Performance of Special Purpose Middleware: A Sensor Networks Case Study," *Tech. Report WUCSE-2003-6*, Washington University in St. Louis.

[24] J.A. Stankovic et. al., "Feedback Control Real-Time Scheduling in Distributed Real-Time Systems," *IEEE Real-Time Systems Symposium (RTSS 2001)*, Dec 2001.

[25] D.C. Steere, et. al., "A Feedback-driven Proportion Allocator for Real-Rate Scheduling," *Symposium on Operating Systems Design and Implementation*, Feb 1999.

[26] N. Wang, D.C. Schmidt, A. Gokhale, C.D. Gill, B. Natarajan, C. Rodrigues, J.P. Loyall, and R.E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications", *Elsevier Journal of Microprocessors and Microsystems*, Vol. 26, No. 9-10, January 2003.

[27] L. Wills, S. Kannan, S. Sander, M. Guler, B. Heck, J.V.R. Prasad, D. Schrage, and G. Vachtsevanos, "An Open Platform for Reconfigurable Control", IEEE Control Systems Magazine, June 2001, pp. 49-64

[28] R. Zhang, C. Lu, T.F. Abdelzaher, and J.A. Stankovic, "ControlWare: a Middleware Architecture for Feedback Control of Software performance," *International Conference on Distributed Computing Systems (ICDCS 2002)*, Vienna, Austria, July 2002.

[29] J.A. Zinky, D.E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects", *Theory and Practice of Object Systems*, 3(1): 1-20, 1997