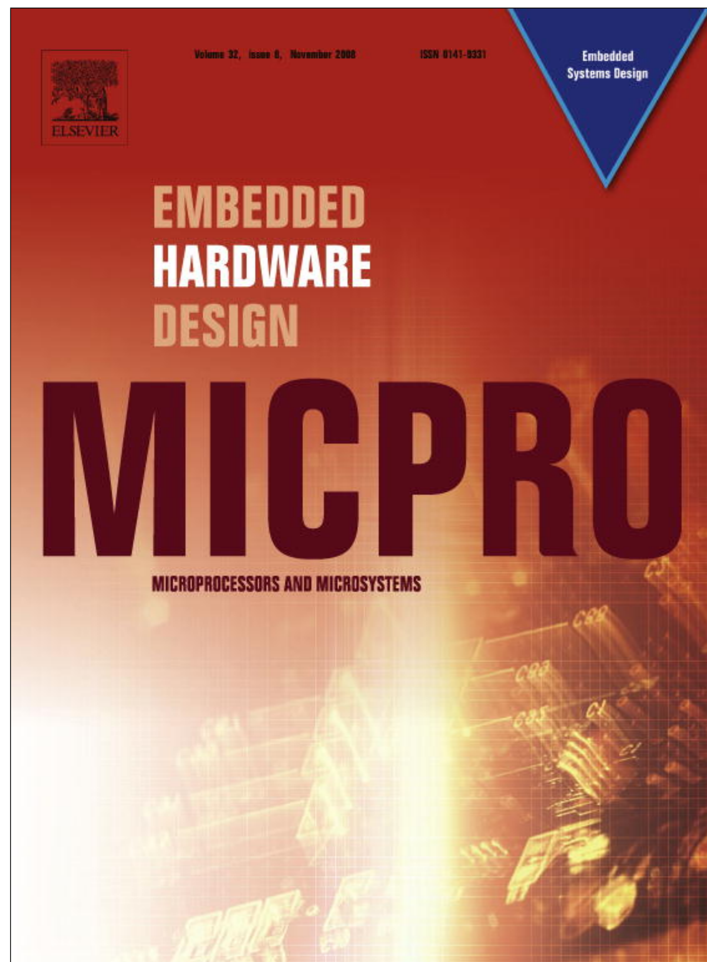
Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

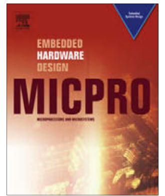
Author's personal copy

Microprocessors and Microsystems 32 (2008) 413–424

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro



FCS/nORB: A feedback control real-time scheduling service for embedded ORB middleware ☆

Xiaorui Wang [a,*], Chenyang Lu [b], Christopher Gill [b]

[a] *Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996, USA*
[b] *Department of Computer Science and Engineering, Washington University in St. Louis, MO 63117, USA*

ARTICLE INFO

Article history:
Available online 23 May 2008

Keywords:
Feedback control
Real-time scheduling
Object request broker
Performance portability
Middleware

ABSTRACT

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. However, existing real-time ORB middleware standards such as RT-CORBA do not adequately address the challenges of (1) managing unpredictable workload, and (2) providing robust performance guarantees portably across different platforms. 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 empirical evaluation of FCS/nORB. Our extensive experiments on a Linux testbed demonstrate that FCS/nORB can provide deadline miss ratio and utilization guarantees in the face of changes in platform and task execution times, while introducing only a small amount of overhead.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

Object Request Broker (ORB) middleware [31,33] 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 [8], unmanned flight control systems [2], and autonomous aerial surveillance [23] 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 [30] provide portability across different operating systems and hardware platforms. Resource management frameworks such as Kokyu [10] use low-level elements to configure scheduling and dispatching mechanisms in higher-level middleware. Real-Time ORBs such as TAO [31] and nORB [33] are geared toward providing predictable timing of end-to-end method invocations. ORB services such as the TAO Real-Time Event Service [12] and TAO Scheduling Service [10] offer higher-level services for managing functional and real-time properties of interactions between application components. Finally, higher-level middleware services [13,29,39,41] 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.

- *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 and location of potential targets in a set of camera images sent to it.
- *Provide real-time performance portability*: A key advantage of middleware is supporting portability across 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

☆ This paper is a substantially extended version of a conference paper [25].

* Corresponding author. Tel.: +1 865 974 0627; fax: +1 865 974 5483.
E-mail addresses: xwang@eecs.utk.edu (X. Wang), lu@cse.wustl.edu (C. Lu), cdgill@cse.wustl.edu (C. Gill).

0141-9331/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2008.05.002

Author's personal copy

not strictly portable even when developed using today's ORB middleware. The lack of robust real-time performance portability thus detracts from the benefits of deploying DRE applications on current-generation ORB middleware.

A key reason that existing ORB middleware cannot deal with the above challenges is that common scheduling approaches are based on *open-loop* algorithms, *e.g.*, Rate Monotonic Scheduling (RMS) or Earliest Deadline First (EDF) [20], which conduct schedulability analysis based on accurate knowledge of task execution times to provide real-time performance guarantees. However, when workloads and available platform resources 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 [24] with real-time embedded ORB middleware [33], 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 *soft* 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 just *simulations*, this paper presents:

- the design and implementation of an FCS feedback control loop in embedded ORB middleware, and
- the detailed *empirical* performance evaluation of FCS scheduling service with realistic workloads.

FCS/nORB provides key scheduling support that makes DRE software performance (1) more robust against workload variations when tasks have negotiable QoS parameters that can be adjusted and (2) portable across OS and hardware platforms. 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 [9,32], sensor data display, and video streaming [5], 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 middleware system.

Specifically, this paper makes three main contributions to research on DRE systems:

- design documentation of an 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 and all the components (e.g., monitor, controller, etc.) of the control loop in an embedded ORB middleware that dynamically adjusts the rates of remote method invocations transparently to the application (subject to application-specified constraints), 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 describe the design and implementation of the FCS/nORB middleware in Section 2. We then present results of our empirical performance evaluation on a Linux testbed with both synthetic workload and realistic workload in Section 3. Section 4 surveys related work in the areas of real-time scheduling, software performance control, and adaptive resource management in middleware. Finally, Section 5 summarizes the contributions of this paper.

2. FCS/nORB design

In this section, we first present the application model and architecture design of FCS/nORB. We then describe the feedback control loop instantiated in FCS/nORB, and introduce the implementation details of each component in the control loop. We also briefly review the FCS algorithms previously proposed in [24] as part of the controller component, and give the implementation information of the FCS/nORB system.

2.1. Application model

We now describe the application model adopted by FCS/nORB. With ORB middleware, applications typically execute using method invocations on objects distributed across multiple end systems. Invocation latency for remote methods includes latency on the client, the server, and the communication network. 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 [21]. In this approach, an end-to-end deadline is divided into intermediate deadlines on the server, client, and communication network, and the problem of meeting the end-to-end deadline is thus transformed into the problem of meeting every intermediate deadline.

In this paper, we focus on the problem of meeting *intermediate deadlines* on a single processor, the server. To precisely measure the overhead of FCS on the client side and the server side, respectively, we run client and server on two different processors. 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. Communication delay is not the focus of this paper, although it is possible to treat a network similarly as a processor in an end-to-end scheduling model.

In the rest of this paper, we use the term *task* to refer to the execution of a remote method on the server. We assume that each task T_i has an *estimated* execution time EE_i known at design-time. However, the *actual* execution time of a task may be significantly different from EE_i and may vary at run time. We also assume that the rate of T_i can be dynamically adjusted within a range $[R_i^{\min}, R_i^{\max}]$. Earlier research has shown that task rates in many real-time applications (*e.g.*, digital feedback control [7], sensor data update, and multimedia [5,6]) can be adjusted without causing application failure. Specifically, each task T_i is described by the following three attributes:

- EE_i: the *estimated* execution time,
- $[R_i^{\min}, R_i^{\max}]$: the range of acceptable rates, and
- $R_i(k)$: the rate in the kth control period. $R_i^{\min} < R_i(k) < R_i^{\max}$.

We use $X(k)$ to represent the value of a variable X in a control period $[(k-1)W, (kW)$, where $k > 1$ and W is the control period length, which is selected so that multiple instances of each task may be released during a control period. We assume all tasks are periodic,[1] and each task T_i's (relative) deadline on the server, $D_i(k)$, is proportional to its period.

[1] The restrictions can be released to handle aperiodic tasks [24].

Author's personal copy

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 estimate and may vary at run-time.

2.2. FCS/nORB architecture design

FCS/nORB is developed based on open-source embedded middleware nORB [33], which is a light-weight real-time ORB designed to support networked embedded systems. Both FCS/nORB and the nORB are based on ACE [30], an open-source object-oriented (OO) framework that implements many core concurrent communication patterns for platform-independent distributed software.

2.2.1. Lane-based architecture

In FCS/nORB, for each priority level that is used for remote method invocation requests, a lane [29] is established between the server and a client. A lane is composed of three parts: the server side part, the client side part, and a separate TCP connection between the server and the client to avoid priority inversion at the communication layer.

As shown in Fig. 1, the client part of a lane has a pair of timer thread and connection thread that are connected through a FIFO queue. Each pair of timer/connection threads 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 timeouts, to initiate method invocation requests at a specified rate.

Similarly, the server part of a lane has a pair of worker and connection threads, connected through an intermediate FIFO queue. 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 multiple clients and then add the requests to the queue. The worker threads remove requests from the queue, invoke the corresponding methods, and send the results back to clients. The separation of worker threads and connection threads simplifies application programming by avoiding the need of handling asynchronous communication.

We apply the RMS policy [20] to assign task priorities to the thread pairs on the server. Each thread pair on the client shares the same priority as the thread pair on the server that it is connected to. A key contribution of this work is to show that feedback control can be realized in reduced-feature-set ORBs such as nORB that are tailored to fit within the space and power limitations seen in many networked embedded systems, without sacrificing real-time performance.

2.2.2. Priority management

The implementation of feedback control introduces new challenges to the design of scheduling mechanisms in ORB middleware. For instance, the rate adaptation mechanism adopted by FCS/nORB and several other projects [36,38] may dynamically change the rates of real-time tasks. When task priorities are determined by task rates (e.g. in the Rate Monotonic Scheduling (RMS) policy), this may cause the middleware to continuously change the priorities of all its tasks. To satisfy the special requirements posed by rate adaptation, FCS/nORB is configured with the *priority-per-lane*concurrency architecture.

Traditionally, a concurrency architecture called thread-per-priority has been adopted in most existing DRE middleware (e.g., [31]), include nORB. In that model, the same thread is responsible for executing all tasks with the same priority. This is because the workload is assumed to use only a limited number of fixed task rates. However, this concurrency architecture is not suitable for rate adaptation. Due to rate adaptation, the rates and thus the priorities of real-time tasks vary dynamically at run-time. In such a situation, the thread-per-priority architecture would require the ORB to dynamically move a subtask from one thread to another thread which may introduce significant overhead.

To avoid this problem FCS/nORB implements the priority-per-lane concurrency architecture that executes each task in a separate lane with a separate priority. FCS/nORB adjusts the priorities of the lanes, and thus the priorities of the threads in the lanes, only when the *order* of the task rates is changed. While the task rates may vary at every control period, the order of task rates often changes at a much lower frequency, especially when FCS/nORB adopts the proportional rate adjustment policy, which is introduced in Section 2.3. Therefore, the priority-per-lane architecture enables FCS/nORB to adapt task rates in a more flexible way, with less overhead.

A potential advantage of the thread-per-priority architecture is that it may need fewer threads to execute applications when multiple tasks share the same rate (i.e., priority). However, as FCS/nORB is targeted at memory-constrained networked embedded systems that commonly have limited number of tasks on a processor, each task can be easily mapped to a lane with a unique native thread priority even in a priority-per-lane architecture.

2.3. FCS/nORB feedback control loop

The core of FCS/nORB is a feedback control loop that periodically monitors and controls its *controlled variables* by adjusting QoS parameters (*e.g.*, task rates or service levels). Candidate controlled variables include the total (CPU) utilization and the

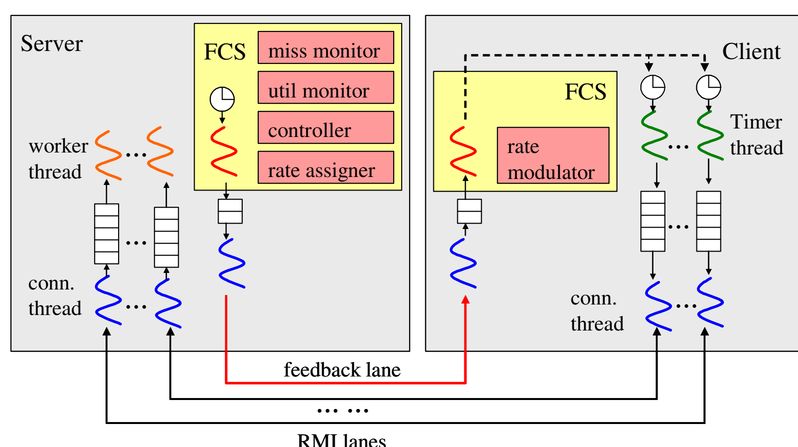


Fig. 1. The Architecture of FCS/nORB.

Author's personal copy

(deadline) miss ratio. The *utilization*, $U(k)$, is defined as the fraction of time when the CPU is busy in the kth control period. The *miss ratio*, $M(k)$, is the number of deadlines missed divided by the total number of completed tasks[2] in the kth control 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 the feedback control loop is to enforce the performance references specified by the application, via run-time QoS adaptation.

The feedback control loop of FCS/nORB on the server side includes a *utilization monitor*, a *miss ratio monitor*, a *controller*, a *rate allocator*, and a pair of FCS/connection threads. The control loop 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.

2.3.1. Utilization monitor

The utilization monitor uses the */proc/stat* file in Linux to estimate the CPU utilization in each control 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 control 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 control period by the total number of jiffies in the same period. We note that the same technique is used by the benchmarking tool, NetPerf [28].

2.3.2. Deadline miss monitor

The deadline miss monitor measures the percentage of completed tasks that miss their deadlines on the server in each control 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 control 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 control period, the deadline miss monitor aggregates the counters of all worker/connection threads, and computes the deadline miss ratio in the control 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 either allow priority inversions or introduce unnecessary overhead to prevent them.

2.3.3. Controller

The controller implements the three control algorithms presented in Section 2.4. Each time its periodically scheduled timer fires, the controller invokes the utilization and/or deadline miss monitors, computes the total estimated utilization for the next control period, and then invokes the rate assigner.

[2] When a task has a *firm* deadline, it may be aborted when it misses its deadline. An aborted task is counted as a completed one and a deadline miss for miss ratio calculation.

2.3.4. 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 $B(0) = \sum_i (EE_i R_i(0))$, and the total estimated utilization for the following (kth) control period is $B(k)$, the PRA policy assigns the new rate to task T_i as follows: $R_i(k) = (B(k)/B(0))R_i(0)$. If $R_i\ (k)$ falls outside its acceptable range $[R_i^{min}, R_i^{max}]$, it is rounded to the closer limit. It can be easily proven that PRA enforces the total estimated utilization, *i.e.*, $B(k) = \sum_i (EE_i R_i(k))$, 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, while 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 control period, it may introduce relatively high overhead for resetting all the timers on the clients. Fortunately, as shown in our measurement, such overhead is small when ACE timers are used.

Note that the PRA policy is based on the assumption that all tasks are "equally important". 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, existing optimization algorithms, *e.g.*, [17], could be used in the rate assigner to address this problem.

2.3.5. Rate modulator

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

2.4. FCS control algorithms

The FCS/nORB controller implements three FCS algorithms developed based on the choice of different sets of these controlled variables [24]. The FC-U and FC-M algorithms each control the utilization $U(k)$ and the miss ratio $M(k)$, respectively, and the FC-UM algorithm controls both $U(k)$ and $M(k)$ at the same time. The utilization and miss ratio monitors measure the controlled variables, $U(k)$ and $M(k)$, respectively. At the end of each control 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 control period. The QoS Actuators then adjust tasks' QoS parameters to enforce the total estimated utilization on the server. For example, a rate actuator assigns a new set of task rates such that, i.e., $B(k+1) = \sum_i (EE_i * R_i(k+1))$, and instructs each client to adjust its invocation rate accordingly. Other examples of QoS actuation mechanisms include admission control and adaptation techniques based on the imprecise computation model [22]. It is important to note that $B(k)$ may be different from $U(k)$ due to the difference between the estimated and actual task execution times.

We now briefly review the three FCS algorithms presented in our earlier work [24] to provide a complete understanding of the controller component. The key contributions of this paper lie in

Author's personal copy

Invoke in the end of each control period

U_s: utilization reference

K_u: utilization controller parameter

FC-U(U_s)

begin

Utilization Controller gets $U(k)$ value measured by the Utilization Monitor;

Utilization Controller computes $B(k+1) = B(k) + K_u*(U_s-U(k))$;

Rate Actuator computes task rates so that $B(k+1) = \sum_i(EE_i*R_i(k+1))$;

Rate Actuator informs clients of the new rates of their tasks.

end

Fig. 2. Pseudo-code of FC-U.

the architecture design and implementation of the whole feedback control loop in embedded ORB middleware and the extensive empirical evaluation to test its efficiency and limitations.

2.4.1. FC-U

FC-U embodies a feedback loop to enforce a specified utilization. Pseudo code for the FC-U algorithm is shown in Fig. 2. 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 a system whose utilization bound is unknown, or may underutilize the system when its utilization bound is highly pessimistic.

2.4.2. FC-M

Unlike FC-U, which controls the miss ratio indirectly through utilization control, FC-M utilizes a feedback loop to control the miss ratio *directly*. Pseudo-code for the FC-M algorithm is shown in Fig. 3.

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, because the 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 [24]. Therefore, FC-M is only applicable to soft real-time systems that can tolerate sporadic deadline misses in steady-state.

2.4.3. FC-UM

FC-UM integrates miss ratio control and utilization control to combine their advantages. Pseudo-code for the FC-UM algorithm is shown in Fig. 4.

The advantage of FC-U is its ability to meet all deadlines ($M(k) = 0$) in steady-states 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.

Based on control theory, the FCS algorithms can be proven to have the following control properties. First, the system stability is guaranteed as long as the actual task execution times within a specified range from (several times greater or smaller than) their estimated values. Second, the performance references (miss ratio or utilization) can be precisely achieved in the steady-state as long as the system is within its stability range, even when task execution times vary at run-time. Finally, the parameters K_u and K_m can be determined based on a trade-off between the stability range and the system settling time. Small values of K_u and K_m may cause

Invoke in the end of each control period

M_s: miss ratio reference

K_m: miss ratio controller parameter

FC-M(M_s)

begin

Miss Ratio Controller gets $M(k)$ value measured by the Miss Ratio Monitor;

Miss Ratio Controller computes $B(k+1) = B(k) + K_m*(M_s-M(k))$;

Rate Actuator adjusts task rates (same as in FC-U).

end

Fig. 3. Pseudo-code for FC-M.

Author's personal copy

Invoke in the end of each control period

U_s: *utilization reference*

M_s: *miss ratio reference*

K_m: *miss ratio controller parameter*

K_u: *utilization controller parameter*

FC-UM(U_s, M_s)

begin

Get $U(k)$ and $M(k)$ from the Utilization and Miss Ratio Monitors, respectively;

Miss Ratio Controller computes $B(k+1) = B(k)+min(K_u*(U_s–U(k)), K_m*(M_s–M(k)))$;

Rate Actuator adjusts task rates (same as in FC-U and FC-M).

end

Fig. 4. Pseudo-code for FC-UM.

long settling time for the system to achieve the desired performance while large values may affect the system stability. The detailed proofs are not shown due to space limitations but available in [24].

2.5. 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 compiler 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://www.ece.utk.edu/~xwang/RTES/FCS_nORB/

3. Empirical evaluations

In this section, we present the results of four sets of experiments we ran on a Linux testbed. Experiment I evaluated the performance portability of applications on FCS/nORB on two different server platforms. On both platforms, we first ran the same synthetic workload for which the actual task execution times significantly deviate from their estimated execution times (the same estimates were used in all experiments). Experiment II stress-tested FCS/nORB's ability to provide robust performance guarantees with a workload whose task execution times *varied* dramatically at run-time. Experiment III adopted an image matching workload that is representative of target location applications, to examine FCS/nORB's robust performance guarantees in *realistic* environment. Finally, Experiment IV measured the *overhead* introduced by the FCS control service from three different perspectives.

3.1. Experimental set-up

3.1.1. Platform

We performed our experiments on three PCs named *Server A*, *Server B*, and *client*. Server A and client were Dell 1.8 GHz 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 1.99 GHz Pentium4 PC with 256 MB of RAM. Server B and client were connected through our 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 host in all experiments.

3.1.2. Workload

To evaluate the robustness of FCS/nORB, we used both a synthetic workload and a more realistic one that simulated real applications in our experiments. Since we focused on unpredictable workload and platform portability, the estimated execution times were different from the actual execution times in all experiments. The same estimated execution times were used in all experiments despite the fact that they used different platforms and as a result had different actual task execution times. With FCS, re-profiling of task execution times was *not* needed to provide performance guarantees.

The *synthetic workload* comprised 12 tasks. Each task periodically 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 “min rate” column in Table 1.

The *realistic workload* comprised of an avionic task set and an additional target location task. The avionics task set is based on an F-16 simulator presented in [1]. It includes four separate tasks (guide, control, and slow and fast navigation) with different rate ranges and execution times as shown in Table 2. We chose these tasks in our workload because their rate ranges are available [1].[3]

The additional target location task is included because of its relatively high computing intensity and its potential execution time variation in the runtime. Those two properties make the simulated avionic system suffer a runtime performance variation, which provides a typical platform for FCS to apply.

A common solution for target location includes a series of steps including image restoration and enhancement, geometric correction, image matching, etc. For the sake of simplicity, we implemented only the most critical step, image matching [27], in our experiment. The goal of image matching is to search *input images* (periodically captured by camera equipments) for a target, which is represented by another smaller sized *template* image. Specifically, every pixel in the input image is potentially part of where the target is located so they all are *candidate points*. All those candidate points will be checked exhaustively for their similarity values with the target template (some advanced image matching algorithms only check a subset of all pixel positions). At each of

[3] We acknowledge that FCS/nORB may not be directly applicable to safety-critical real-time systems such as flight control for *manned* aircraft, which require hard performance guarantees.

Author's personal copy

Table 1
Methods invoked by the workload

Method	Estimated execution time (ms)	Min rate	Max rate	Number of invoking tasks
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

Table 2
Task sets in real image matching workload

Task	Est. execution time (ms)	Min rate	Max rate
Guide	100	0.2	1.0
Control	80	1.0	5.0
Slow navigation	100	0.2	1.0
Fast navigation	60	1.0	5.0
Target location	150	0.2	5.0

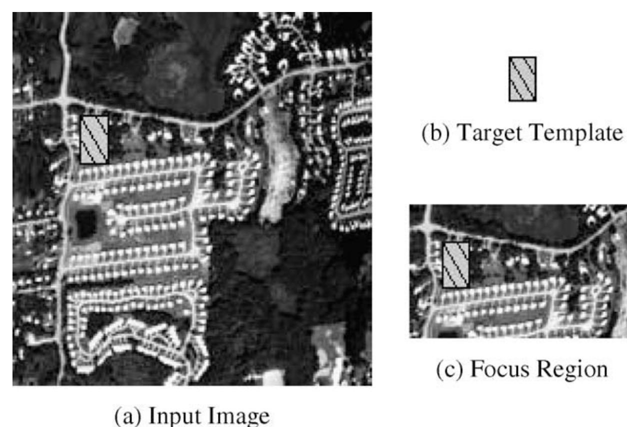


Fig. 5. Images used in Experiment III.

the candidate points, a *candidate region* with the same size as target template is extracted from the input image to compare with target template pixel by pixel. All the individual similarity values from each of the corresponding pixel pairs are summed up as an overall similarity value for this candidate point. The candidate point with largest similarity will be identified as the *match place*, so long as its similarity value is larger than a pre-defined threshold. A target is considered to have been located when its match place is found. In our experiment, absolute difference (AD) [27] is used to compute the similarity.

The application scenario for our experiment is as follows. Before the target object of interest is located, the image matching task searches the full input image for a match with the template. After an object is found at a particular location, a *focus region* is shrunk from the full image to a small region that is centered at the known location of the object in subsequent images to save CPU cycles for other tasks. However, in some cases a fast moving object may escape from the focus region between two consecutive invocations of the task, resulting in the loss of the object template. In this situation, the full image must be searched again to relocate the target. Therefore, in our scenario the execution time for the image matching task starts at a high level in the beginning, then drops to a low-level when the target has been detected. After this target is lost, the execution time returns again to its initial high level. The variation in the execution time is *unknown a priori* because it depends whether the target is being found on the input image or the focus region.

Fig. 5 shows those images used in our experiment. Since the execution time of the exhaustive image matching with AD algorithm depends only on the sizes of those images and is insensitive to their contents, a same input image (Fig. 5a) can be used in every invocation of the image matching task without affecting the task workload, which is the main concern of FCS/nORB. Similarly, a same focus region (Fig. 5b) can also be used. The switch between the full input image and the focus region is forced to simulate the target capture and loss. In the sequence described in above scenario, we first use the full input image to search for the target template. At a certain time the target is found and we then start to search the focus region image for a while. Finally we change back to the input image by assuming the target is lost from the focus region. Although the images used in our experiments are simplified compared to real world scenarios, they are sufficient for the purpose of causing realistic variations in the task execution time.

3.1.3. FCS configuration

The configuration parameters for FCS are shown in Table 3. 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 workloads and execution times. The controller parameters K_u and K_M were computed using control theory based on a trade-off between stability range and system settling time [24]. 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 it uses miss ratio control as we discussed in Section 3. The control period used in Experiments I and II is 4 s. Since in the real image matching workload task 1 and task 3 both have 5 s as their maximum period, we set the control period in Experiment III to 10 s to decrease the sampling jitter caused by rate tuning. As a baseline, we also ran these 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 3
FCS configuration in all experiments

	FC-U	FC-M	FC-UM
Reference	$U_s = 70\%$	$M_s = 1.5\%$	$M_s = 1.5\%$ $U_s = 75\%$
K_u, K_M	$K_u = 0.185$, $K_M = 0.414$		
Control period	4 s (10 s in Experiment III)		

3.2. Experiment I: performance portability

In Experiment I, the execution time of each task on Server A remained approximately twice its estimated value 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 significant inaccuracy in task profiling.

Our first experiment emulates 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 was 70%. However, when we ran the tasks at the rates according to the predicted rates, the server locked up. This is not surprising: since the estimated execution times were inaccurate, the actual total requested utilization by all nORB threads reached approximately 140%. This caused the Linux kernel to freeze because all nORB threads were run (with the *root* privilege) at real-time scheduling priorities that are higher than kernel priorities on Linux. When the CPU utilization requested by nORB threads reached 100%, no kernel activities were able to execute. To avoid

Author's personal copy

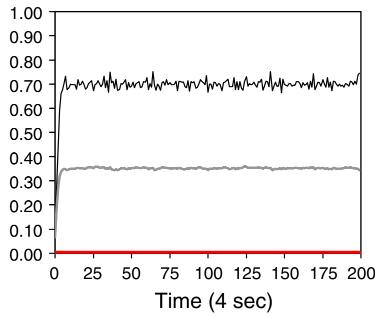


Fig. 6. A typical run of FC-U on server A.

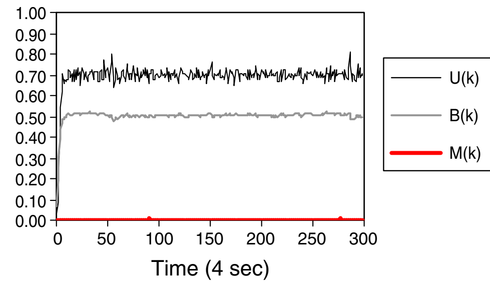


Fig. 8. A typical run of FC-U on Server B.

this problem using common real-time engineering techniques, 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 performance 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 system development time.

We now examine the experimental results for the FCS algorithms themselves. As an example, Fig. 6 illustrates the sampled utilization $U(k)$, the miss ratio $M(k)$, and the total estimated utilization $B(k)$ computed by the controller in a typical run under FC-U. All tasks started from their lowest rates. The feedback control loop rapidly 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 shows 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 are consistent with the control analysis presented in [24].

The performance results for FC-U, FC-M, and FC-UM on Server A are summarized in Fig. 7a–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 also be viewed as the duration of the self-tuning period after an application is ported to a new platform. It is usually 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 $0.99U_s$ in the last control period of the experiment. Each data point in Fig. 7a–c is the mean of three repeated runs, and each run took 800 s. The standard deviations in miss ratio, utilization, and settling time are below 0.01%, 0.03%, and 6.11 s (i.e., a 1.53 control period), respectively.

From Fig. 7a, 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 of 1.5%. At the same time, the steady-state utilizations of FC-U and FC-UM are respectively 70.01% and 74.97%, compared to respective utilization references of 70.00% and 75%. The result for FC-UM occurred 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 Fig. 7c, 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 setting the initial task rates closer to the desired rates. For example, we may choose the initial rates to be the same as the desired rates on the slowest platform in a product line.

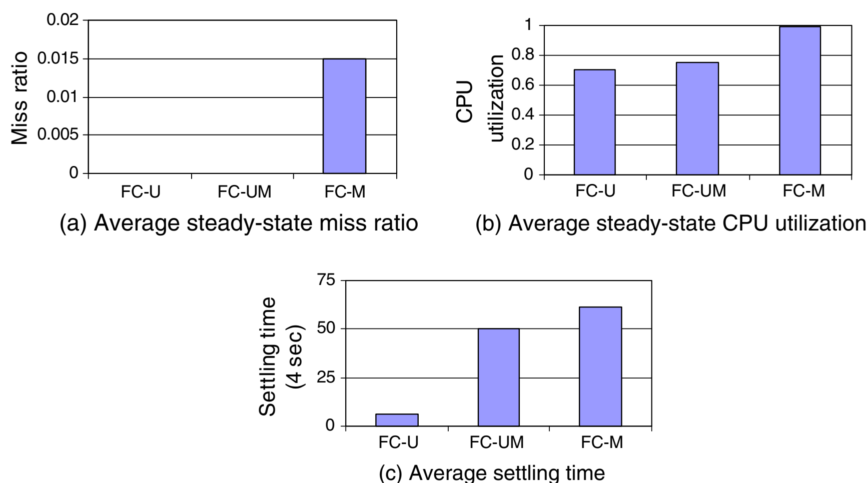


Fig. 7. Performance results of FCS algorithms on server A in Experiment II.

Author's personal copy

To further evaluate the performance portability of FCS/nORB, we re-ran the same experiments on Server B. Typical runs of FC-U, FC-UM, and FC-M are shown in Figs. 8–10, respectively. Each run takes 1200 s. As was the case on Server A, all the algorithms successfully enforced their utilization and/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 [24].

In summary, Experiment I demonstrated that FCS/nORB can provide a 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 represents 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 can be scheduled initially 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.

3.3. Experiment II: varying realistic workload

In Experiment II, we examined FCS/nORB's performance with the *realistic* workload in which the execution time of the target location task vary dynamically.

Fig. 11 shows a typical run of OPEN. In the beginning of the run, the target location task had a long execution time while it searched the whole input images for the interested object. Consequently, the CPU utilization was close to 95% and a number of task invocations missed their deadlines. At around 160th control period, the target was assumed to have been found, so the focus region was shrunk to locate the target. CPU utilization dropped significantly as we can observe in Fig. 11. This drop switched the system from an overloaded to an underutilized status. We continued by assuming the target escaped from the focus region at around the 265th control period, so the execution time of the target location task then returned to its original level. At that point utilization again returned to an overload condition. Hence in this scenario, the OPEN system just switched back and forth between overload with deadline misses, and underutilization with unnecessarily low task rates, neither of which leads to satisfactory performance.

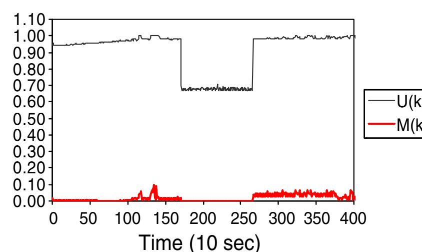


Fig. 11. A typical run of OPEN under realistic workload.

Figs. 12 and 13 show that both FC-U and FC-UM maintained specified utilization levels in their steady-states, which was over most of the entire run. The performance of FC-U is illustrated in Fig. 12. The CPU utilization was decreased to the set point (70%) at the 15th period, so deadline misses were avoided. At around 160th period, when the system found the object, FC-U drove the utilization back up to the set point by increasing the rates of all current tasks to utilize the CPU better. The faster rates also improved the system utility. Particularly for our image matching task, more frequent invocation of the image matching task improves tracking precision while reducing the chance that the tracked object may escape from the window image. At the 165th control period, when the target did escape from the focus region, the full image was searched again to relocate the target. FC-U only had a very transient spike of deadline misses, which is highly preferable compared with OPEN. The performance of FC-UM shown in Fig. 13 had similar results to FC-U. The only difference is that the settling time was longer when the system recovered from underutilized status, for the same reasons explained in previous section.

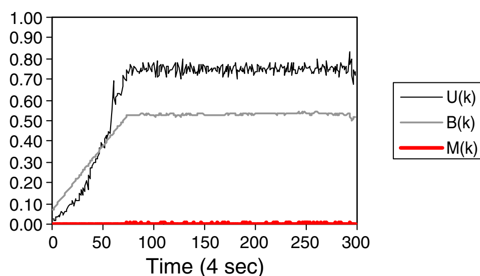


Fig. 9. A typical run of FC-UM on Server B.

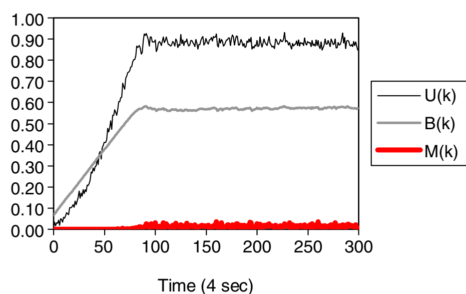


Fig. 10. A typical run of FC-M on Server B.

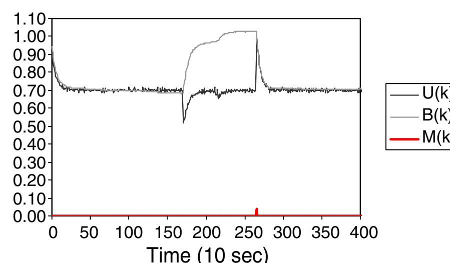


Fig. 12. A typical run of FC-U under varying workload.

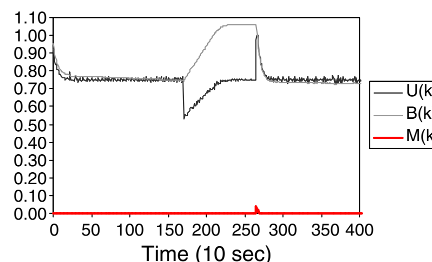


Fig. 13. A typical run of FC-UM under varying workload.

Author's personal copy

Interestingly, FC-M caused the system to lock up when the execution times increased. This is because FC-M achieved a high utilization (more than 90%) before the execution time increased at around 160th control period. The utilization then increased to 100% due to the increase in execution times, and the system again locked up due to kernel starvation. In contrast, previous simulation results [24] showed that FC-M could handle such varying workload because the impact of CPU over-utilization by the middleware on kernel activities was not modeled in the simulator, which was design to simulate a scheduler in the OS kernel.

In general, FCS/nORB cannot handle varying workload that even transiently increases the utilization to 100% due to the starvation of the kernel under such conditions. This result shows a limitation of middleware implementations 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 the FCS algorithms 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 the face of varying workload. Since FC-M usually achieves a high utilization and, more importantly, does not have control over its safety margin, a middleware implementation of FC-M is less appropriate for time varying workloads.

Experiments II 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.

3.4. Experiment III: overhead measurement

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

3.4.1. Coarse-grained overhead measurement

To quantify the overhead imposed by 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 utilization monitoring for OPEN and FC-M, average CPU utilizations were measured by setting the control 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 at 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 control 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 summarized in Table 4. The first row shows the mean of the average utilizations in 8 repeated runs, along with its 90% confidence interval. Each run lasted for 800 s, a total of 200 control periods. The second row shows the overhead of each FCS algorithm in terms of CPU utilization, which is 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 FC-U showed no statistically significant overhead based on our measurement. FC-M and FC-UM, however, showed statistically significant

Table 4
Results of coarse-grained 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 compared to OPEN. Over a 4 s control period, all three FCS algorithms introduced overhead of less than 1% of the total CPU utilization. FC-U introduced the least overhead among all FCS algorithms, 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 control period, the miss ratio monitor requires time stamping every method invocation twice. FC-UM's overhead is slightly less than the sum of the overheads from FC-M and FC-U. This is because, while FC-UM ran both monitors, it only execute the controller and actuator once per invocation.

3.4.2. Fine-grained overhead measurement

Although the above overhead measurement shows satisfactory results based on utilization comparison between OPEN and FCS algorithms, we noticed two limitations of the above measurement approach. The first one is that the Linux system file /proc/stat records the number of *jiffies*. Since each jiffy is 10 ms (1/100 of a second), the granularity of above measurement is too coarse for precise measurements on the overhead. The second problem is that CPU utilization may suffer interference from the operating system itself even though we minimized the number of system processes.

To measure overhead more accurately, we adopted a time stamping approach. Firstly, we differentiated all FCS related code from the original nORB code. Then two time stamps were taken at the starting point and finishing point of each segment of FCS code to get the execution time of FCS. Fortunately, since most FCS code is within feedback lane which is running with highest Linux real-time priority, the code segment between two timestamps will not be preempted during its execution. Hence, the time-stamped result accurately reflects the real execution overhead.

To achieve fine-grained measurements, we needed an accurate time stamping function. The commonly used *gettimeofday* system call can not be used here since this function is also based on a 10 ms scale. Instead we adopted a nanosecond scale time measuring function called *gethrtime*. This function uses an OS-specific high-resolution timer, which can be found on Solaris, AIX, Win32/Pentium, Linux/Pentium and VxWorks, to return the number of clock cycles since the CPU was powered up or reset. The *gethrtime* function has a low overhead and is based on a 64 bit clock cycle counter. With the clock counter number divided by the CPU speed, we can get reasonably precise and accurate time measurements. Since *gethrtime* is supported on Pentium processor, we performed our fine-grained overhead measurements on Server B, a Dell Pentium4 PC.

In Table 5 we list all FCS related operations and their overheads for the three FCS algorithms respectively. All results in that table are averaged values of 10 runs and each run's result is an average over 300 continuous control periods. Operations 1–4, respectively give the overhead of the utilization monitor, the miss ratio monitor, the controller, and the rate assigner, all of which ran in a feedback lane at highest priority. Operation 5 ran in the remote method invocation lane and was used to time stamp each remote method invocation from the client side *twice* to check whether it meets the deadline as Section 2.2 explains. The overheads of operations 1–4 are relatively fixed for each control period, while the overhead of operation 5 depends on how many invocations come from client side in a given control period. The measured overhead for one sin-

Author's personal copy

Table 5
Results of differentiated fine-grained overhead measurement

#	Name	Description	FC-U (μs)	FC-M (μs)	FC-UM (μs)
1	Utilization monitor	*/proc/stat* system file reading	160.90	N/A	263.22
2	Miss ratio monitor	Deadline miss ratio reading	N/A	181.29	
3	Controller	Control analysis	40.64	49.84	43.27
4	Rate assigner	Calculating new rate; transmitting new rate to client side	659.90	633.73	637.74
5	Time stamp	Time stamping each remote method invocation twice to check deadline	N/A	0.1246	0.1246
	Total	(Assuming 1000 remote method invocations in each control period)	**861.44**	**989.46**	**1068.82**

gle *gethrtime* call is 0.0623*us*. With n invocations in one control period, the overhead for time stamping is 0.1246n *us*. In the total value row, we assume a common application model which has 10 tasks running at a rate of 100 invocations per second. If the control period is 1 s, we get 1000 remote method invocations per control period.

From Table 5 it is easy to see that FC-U has the lowest overhead and FC-UM has the highest overhead. That observation is consistent with our coarse-grained overhead measurements. It is also interesting to find that fine-grained overhead result for FCS algorithms is actually much less than the result of the coarse-grained measurement. The reason is coarse-grained measurement is based on 10 ms measurement accuracy so it is unable to gauge this overhead precisely.

Fig. 14 illustrates the overheads for the monitor, controller and rate assigner in the three FCS algorithms while the overhead of time stamping is not included. Rate assigner has the dominant overhead because it involves relatively more complicated internal data structure access, modification and socket handling while there are just several lines of code for monitor and controller. Overall, the server overhead of all FCS algorithms in our experiments is around 1 ms per control period, which is clearly acceptable in a wide range of real-time and embedded applications.

3.4.3. Memory footprint measurement

Besides execution time, memory overhead is also a significant factor for overall system performance. For embedded systems, however, code size is a major part of the memory footprint because all code of a system is typically loaded into the memory before the system starts to execute. Hence, it is useful to measure the code size increase after we plugged in FCS related code. The code size of FCS/nORB is 680 KB for client and 801 KB for server, while the size of nORB is 602 KB for client and 723 KB for server. We see that adding FCS only resulted in an increase of 78 KB on both client and server. The ratio of increase is only 12% and 10% for the client side and server side, respectively. This minor increase is acceptable considering the system performance improvements that were seen in the previous experiments. We note that the combined static footprint on each given endsystem, of both FCS/nORB and the client or server application, is well below the static footprint of a full-featured ORB such as TAO alone (detailed footprint results of nORB and TAO are available in [33]).

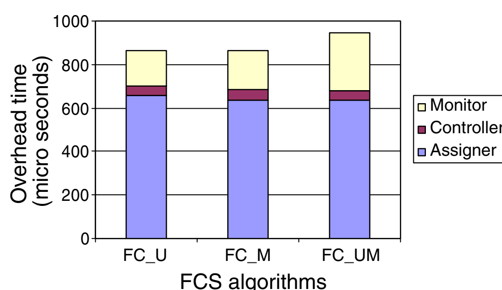


Fig. 14. Detailed overhead measurement.

4. Related work

Control theoretic approaches have been applied to various computing and networking systems. A survey of feedback performance control for software services is presented in [3]. 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 [35] that coordinated allocation of the CPU to consumer and supplier threads, to guarantee the fill level of buffers. Goel et al. developed feedback-based schedulers [11] that guarantee desired progress rates for real-time applications. Abeni et al., presented analysis of a reservation-based feedback scheduler in [4]. In our previous work [24,25,38], feedback control 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 [26,34]. For systems requiring discrete control adaptation strategies, hybrid control theory has been adopted to control state transitions among different system configurations [1,16]. Feedback control has also been successfully applied to power control [18] and digital control applications [7]. A key 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 simulations [1,16,24–26,34,38] or kernel level implementation [4,7,35].

Adaptive middleware is emerging as a core building block for DRE systems. For example, TAO [31], dynamicTAO [15], ZEN [14], and nORB [33] are adaptive middleware frameworks that can (re)configure various properties of Object Request Broker (ORB) middleware at design- and run-time. Higher-level adaptive resource management frameworks, such as QuO [41], Kokyu [10] and RTARM [13], leverage lower-level mechanisms provided by ORB middleware to (re)configure scheduling, dispatching, and other QoS mechanisms in higher-level middleware. ORB services such as the TAO Real-Time Event Service [11] and TAO Scheduling Service [10] offer high level services for managing functional and real-time properties of interactions between application components. The difference between the work presented in this paper and earlier work at adaptive ORB middleware is that our work integrated a unified control theoretic framework with a reduced-feature-set ORB (nORB). As a result, our work provides adaptive middleware service, with real-time performance guaranteed by control theories, to networked embedded systems with storage space and power limitations. Agilos [19] was an earlier effort on control-based middleware framework for QoS adaptation in distributed multimedia applications. Our work provides a general framework which is applicable to various real-time applications, whereas Agilos only supports adaptation strategies (e.g., image operations) specific to multimedia applications (e.g., visual tracking).

Another project that is closely related to FCS/nORB is ControlWare [40], 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

Author's personal copy

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 method invocation mechanisms for real-time embedded systems. In our previous work, we have developed FC-ORB [36], feedback controlled middleware for distributed real-time systems. FC-ORB only controls processor utilization and is designed to handle end-to-end tasks while FCS/nORB controls both processor utilization and deadline miss ratio and focuses on the server side.

WSOA [8] gave a large-scale demonstration of adaptive resource management at multiple architectural levels in a realistic distributed avionics mission computing environment. The WSOA image transmission application is in essence a networked ad hoc control system, with adaptation of image tile compression to meet download deadlines. Based on the WSOA application, a real-time system computing model and theoretical controller has been developed in [37]. Similar to the WSOA project, in this paper, we also seek to complement existing middleware projects for DRE systems, and increase the capabilities offered by DRE middleware as a whole.

5. Conclusions

In summary, we have designed and implemented a Feedback Control real-time scheduling service atop 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 levels even when task execution times deviate significantly from their estimated values or change significantly at run-time; (2) FCS/nORB can provide similar 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 client and 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.

Acknowledgement

This work is funded, in part, by the US National Science Foundation under Grant CNS-0720663 and by DARPA under Grant NBCHC030140.

References

[1] S. Abdelwahed, S. Neema, J. Loyall, R. Shapiro, A hybrid control design for QoS management, in: IEEE Real-Time Systems Symposium (RTSS), December 2003.
[2] T.F. Abdelzaher, E.M. Atkins, K.G. Shin, QoS negotiation in real-time systems and its application to automated flight control, IEEE Transactions on Computers 49 (11) (2000).
[3] T.F. Abdelzaher, J.A. Stankovic, C. Lu, R. Zhang, Y. Lu, Feedback performance control in software services, IEEE Control Systems 23 (3) (2003).
[4] L. Abeni, L. Palopoli, G. Lipari, J. Walpole, Analysis of a reservation-based feedback scheduler, in: IEEE Real-Time Systems Symposium (RTSS), December 2002.
[5] S. Brandt, G. Nutt, A dynamic quality of service middleware agent for mediating application resource usage, in: IEEE Real-Time Systems Symposium (RTSS), December 1998.
[6] Giorgio Buttazzo, Luca Abeni, Adaptive workload management through elastic scheduling, Real-Time Systems 23 (1–2) (2002).
[7] A. Cervin, J. Eker, B. Bernhardsson, K.-E. Årzén, Feedback-feedforward scheduling of LQG-control tasks, Real-time Systems Journal 23 (1–2) (2002).
[8] D. Corman, WSOA – weapon systems open architecture demonstration – using emerging open system architecture standards to enable innovative techniques for time critical target prosecution, in: IEEE/AIAA Digital Avionics Systems Conference (DASC), October 2001.
[9] J. Eker, Flexible embedded control systems-design and implementation. PhD-thesis, Lund Institute of Technology, December 1999.
[10] C.D. Gill, D.L. Levine, D.C. Schmidt, The design and performance of a real-time CORBA scheduling service, Real-Time Systems Journal 20 (2) (2001).
[11] A. Goel, J. Walpole, M. Shor, Real-rate scheduling, in IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2004.
[12] T.H. Harrison, D.L. Levine, D.C. Schmidt, The design and performance of a real-time CORBA event service, in ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 1997.
[13] J. Huang, Y. Wang, 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) (1999).
[14] R. Klefstad, D.C. Schmidt, C. O'Ryan, Towards highly configurable real-time object request brokers, in: the IEEE/IFIP International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), March 2002.
[15] F. Kon, F. Costa, G. Blair, R. Campbell, The case for reflective middleware, Communications of the ACM 45 (6) (2002).
[16] X. Koutsoukos, R. Tekumalla, B. Natarajan, C. Lu, Hybrid supervisory control of real-time systems, in: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), March 2005.
[17] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, J. Hansen, A scalable solution to the multi-resource QoS problem, in: IEEE Real-Time Systems Symposium (RTSS), December 1999.
[18] C. Lefurgy, X. Wang, M. Ware, Server-level power control, in: IEEE International Conference on Autonomic Computing (ICAC), June 2007.
[19] B. Li, K. Nahrstedt, A control-based middleware framework for quality of service adaptations, IEEE Journal on Selected Areas in Communications, Special Issue on Service Enabling Platforms 17 (9) (1999).
[20] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of ACM 20 (1) (1973).
[21] J.W.S. Liu, Real-Time Systems, Prentice Hall, 2000.
[22] J.W.S. Liu et al., Algorithms for scheduling imprecise computations, IEEE Computer 24 (5) (1991).
[23] J. Loyall et al., Comparing and contrasting adaptive middleware support in wide-area and embedded distributed object applications, in: IEEE International Conference on Distributed Computing Systems, April 2001.
[24] C. Lu, J.A. Stankovic, G. Tao, 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) (2002).
[25] C. Lu, X. Wang, C.D. Gill, Feedback control real-time scheduling in ORB middleware, in: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2003.
[26] C. Lu, X. Wang, X.K. Koutsoukos, End-to-end utilization control in distributed real-time systems, in: International Conference on Distributed Computing Systems (ICDCS), March 2004.
[27] W.K. Pratt, Digital Image Processing, John Wiley & Sons, New York, 1978.
[28] Public Netperf Homepage, <http://www.netperf.org>.
[29] D. Rosu, K. Schwan, S. Yalamanchili, R. Jha, On adaptive resource allocation for complex real-time applications, in: IEEE Real-Time Systems Symposium, December 1997.
[30] D.C. Schmidt, The ADAPTIVE communication environment: an object-oriented network programming toolkit for developing communication software, in: 12th Annual Sun Users Group Conference, December 1993.
[31] 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.
[32] D. Seto, J.P. Lehoczky, L. Sha, K.G. Shin, On task schedulability in real-time control systems, in: IEEE Real-Time Systems Symposium (RTSS), December 1996.
[33] V. Subramonian, G. Xing, C.D. Gill, C. Lu, R. Cytron, Middleware specialization for memory-constrained networked embedded systems, in: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 2004.
[34] J.A. Stankovic et al., Feedback control real-time scheduling in distributed real-time systems, in: IEEE Real-Time Systems Symposium (RTSS), December 2001.
[35] D.C. Steere et al., A feedback-driven proportion allocator for real-rate scheduling, in: Symposium on Operating Systems Design and Implementation (OSDI), February 1999.
[36] X. Wang, Y. Chen, C. Lu, X. Koutsoukos, FC-ORB: a robust distributed real-time embedded middleware with end-to-end utilization control, Elsevier Journal of Systems and Software 80 (7) (2007).
[37] X. Wang, M. Chen, H. Huang, V. Subramonian, C. Lu, C. Gill, CAMRIT: control-based adaptive middleware for real-time image transmission, IEEE Transactions on Parallel and Distributed Systems 19 (6) (2008).
[38] X. Wang, D. Jia, C. Lu, X. Koutsoukos, DEUCON: decentralized end-to-end utilization control for distributed real-time systems, IEEE Transactions on Parallel and Distributed Systems 18 (7) (2007).
[39] L.R. Welch, B. Shirazi, B. Ravindran, Adaptive resource management for scalable, dependable real-time systems: middleware services and applications to shipboard computing systems, in: IEEE Real-time Technology and Applications Symposium (RTAS), June 1998.
[40] R. Zhang, C. Lu, T.F. Abdelzaher, J.A. Stankovic, ControlWare: a middleware architecture for feedback control of software performance, in: International Conference on Distributed Computing Systems (ICDCS), July 2002.
[41] J.A. Zinky, D.E. Bakken, R. Schantz, Architectural support for quality of service for CORBA objects, Theory and Practice of Object Systems 3 (1) (1997).