Enhancing Adaptivity via Standard Dynamic Scheduling Middleware

Christopher Gill, Louis Mgeta, Yuanfang Zhang, and Stephen Torri[1]
Washington University, St. Louis, MO
{cdgill, lmm1, yfzhang, storri}@cse.wustl.edu

Yamuna Krishnamurthy and Irfan Pyarali
OOMWorks, Metuchen, NJ
{yamuna, irfan}@ oomworks.com

Douglas C. Schmidt
Vanderbilt University, Nashville, TN
d.schmidt@vanderbilt.edu

Abstract *This paper makes three contributions to research on QoS-enabled middleware for open distributed real-time embedded (DRE) systems. First, it describes the design and implementation of a dynamic scheduling framework based on the OMG Real-Time CORBA 1.2 specification (RTC1.2) that provides capabilities for (1) propagating QoS parameters and a locus of execution across endsystems via a distributable thread abstraction and (2) enforcing the scheduling of multiple distributable threads dynamically using standard CORBA middleware. Second, it examines the results of empirical studies that show how adaptive dynamic scheduling and management of distributable threads can be enforced efficiently in standard middleware for open DRE systems. Third, it presents results from case studies of multiple adaptive middleware QoS management technologies to monitor and control the quality, timeliness, and criticality of key operations adaptively in a representative DRE avionics system.*

Keywords: *Real-time CORBA, adaptive systems, dynamic scheduling.*

1 Introduction

Emerging trends and challenges for DRE systems and middleware. Developing distributed real-time and embedded (DRE) systems whose quality of service (QoS) can be assured even in the face of changes in available resources or in applications' QoS requirements is an important and challenging R&D problem. QoS-enabled middleware has been applied successfully to *closed* DRE systems where the set of tasks that will run in the system and their requirements for system resources are known in advance. For example, middleware based on the Real-time CORBA 1.0 (RTC1) standard [1] supports statically scheduled DRE systems (such as avionics mission computers and industrial process controllers) in which task eligibility can be mapped to a fixed set of priorities.

For an important emerging class of *open* DRE systems (such as adaptive audio/video streaming [2], collaborative mission replanning [3], and robotics applications designed for close interaction with their environments [4]), however, it is often not possible to know the entire set of application tasks that will run on the system, the loads they will impose on system resources in response to a dynamically changing environment, or the order in which the tasks will execute. This dynamism can occur because the number of combinations in which application tasks can be mapped to system resources is too large to compute efficiently or because task run-time behaviors are simply too variable to predict accurately. In either case, open DRE systems must be able to adapt *dynamically* to changes in resource availability and QoS requirements.

Assuring effective QoS support in the face of dynamically changing requirements and resources, while keeping overhead within reasonable bounds, requires a new generation of middleware mechanisms. In particular, the following are important limitations with applying RTC1

[1] This work was supported in part by the DARPA PCES program, contract F33615-03-C-4111

middleware capabilities to open DRE systems (e.g., by manipulating task priorities dynamically at run-time):

- Limits on the number of priorities supported by common-off-the-shelf (COTS) real-time operating systems can reduce the granularity at which dynamic variations in task eligibility can be enforced.
- Without middleware-mediated mechanisms for enforcing QoS, the application itself must provide priority manipulation mechanisms, which is tedious and error-prone for DRE system developers.
- Without open, well-defined, and replaceable scheduling mechanisms within the middleware itself, it is hard to integrate middleware features closely for better performance or to customize QoS enforcement policies so they meet the needs of each application.

Solution approach → Adaptive DRE middleware via dynamic scheduling. The OMG Real-Time CORBA 1.2 specification (RTC1.2) [5] addresses limitations with the fixed-priority mechanisms specified in RTC1 by - introducing two new concepts to Real-time CORBA: (1) *distributable threads* that are used to map end-to-end QoS requirements to distributed computations across the endsystems they traverse and (2) a *scheduling service architecture* that allows applications to choose which mechanisms enforce task eligibility. To facilitate the study of standards-based dynamic scheduling middleware, we have implemented a RTC1.2 framework that enhances on our prior work with The ACE ORB (TAO) [6] (a widely-used open-source implementation of Real-time CORBA 1.0 [1]) and its Real-time Scheduling Service (which supports both static [7] and dynamic scheduling [8]). This paper describes how we designed and optimized the performance of our RTC1.2 Dynamic Scheduling framework to address the following design challenges for adaptive DRE systems:

- Defining a means to install pluggable schedulers that support more adaptive scheduling policies and mechanisms for a wide range of DRE systems,
- Creating an interface that allows customization of interactions between an installed RTC1.2 dynamic scheduler and an application,
- Portable and efficient mechanisms for distinguishing between distributable vs. OS thread identities, and
- Safe, effective mechanisms for controlling distributed concurrency, *e.g.*, for canceling distributable threads.

The results of our efforts have been integrated with the TAO open-source software release and are available from `deuce.doc.wustl.edu/Download.html`.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes the RTC1.2 specification and explains the design of our RTC1.2 framework, which has been integrated with the TAO open-source Real-time CORBA object request broker (ORB); Section 3 presents micro-benchmarks we conducted to quantify the costs of dynamic scheduling of distributable threads in our RTC1.2 framework and also presents two broader case studies of applying key dynamic scheduling mechanisms to DRE avionics applications; Section 4 compares our work with related research; and Section 5 offers concluding remarks.

2 A Dynamic Scheduling Framework for Real-Time CORBA 1.2

This section outlines the RTC1.2 specification and describes how the RTC1.2 dynamic scheduling framework we integrated with TAO helps with the development of adaptive systems.

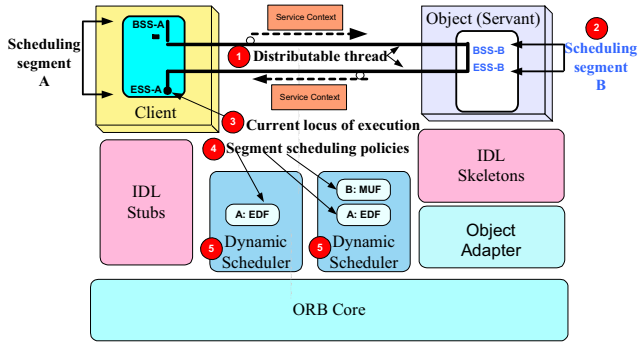

Figure 1: TAO's RTC1.2 Architecture

As shown in Figure 1, the key elements of TAO's RTC1.2 framework are:

1. *Distributable threads*, which applications use to traverse endsystems along paths that can be varied on-the-fly based on ORB- or application-level decisions,
2. *Scheduling segments*, which map policy parameters to distributable threads at specific points of execution so that new policies can be applied and existing policies can be adapted at finer granularity,
3. *Current execution locus*, which is the head of an active distributable thread that acts much like the head of an application or kernel thread, i.e., it can take alternative decision branches at run-time based on the state of the application or supporting system software,
4. *Scheduling policies*, which determine the eligibility of each thread based on parameters of the scheduling segment within which that thread is executing, and
5. *Dynamic schedulers*, which reconcile and adapt the scheduling policies for all segments and threads to determine which thread is active on each endsystem.

Distributable threads help enhance the adaptivity of DRE systems by providing an effective abstraction for managing the lifetime of sequential or branching distributed

operations dynamically. The remainder of this section explains the concepts of *distributable threads* and the adaptive management of their real-time properties through the *pluggable scheduling framework* specified by RTC1.2. This section also outlines the design of these concepts in TAO's RTC1.2 framework implementation. We describe *scheduling points*, which govern how and when scheduling parameter values can be mapped to distributable threads. We also discuss issues related to distributable thread identity and examine the interfaces and mechanisms needed to cancel distributable threads safely.

2.1 Distributable Threads

DRE systems must manage key resources, such as CPU cycles and network bandwidth, to ensure predictable behavior along each end-to-end path. In RTC1-based static DRE systems, end-to-end priorities can be acquired from clients, propagated with invocations, and used by servers to arbitrate access to endsystem resources. For dynamic DRE systems, the fixed-priority propagation model provided by RTC1 is insufficient because more information than priority is required, e.g., they may need deadline/execution time and the values of these parameters may vary during system execution. A more sophisticated abstraction is therefore needed to identify the most eligible schedulable entity, and additional scheduling parameters may need to be propagated with each entity so it can be scheduled properly.

The RTC1.2 specification defines a programming model abstraction called a *distributable thread*, which can span multiple endsystems and is the primary schedulable entity in RTC1.2-based DRE systems. Each distributable thread in RTC1.2 is identified by a unique system wide identifier called a Globally Unique Id (GUID) [10]. A distributable thread may also have one or more execution scheduling parameters, e.g., priority, deadline time-constraints, and importance. These parameters are used by RTC1.2 schedulers for resource arbitration, and also convey acceptable end-to-end timeliness bounds for completing sequences of operations in CORBA object instances that may reside on multiple endsystems.

On each endsystem, a distributable thread is mapped onto the execution of a local thread provided by the OS. At a given instant, each distributable thread has only one execution point in the whole system, i.e., a distributable thread does not execute simultaneously on multiple endsystems it spans. Instead, it executes a code sequence consisting of nested distributed and/or local operation invocations, similar to how a local thread makes a series of nested local operation invocations.

Below, we describe the key capabilities of distributable threads in the RTC1.2 specification, explain how we implement these capabilities in TAO, and identify their relevance to adaptive DRE systems.

Scheduling segment. A distributable thread comprises one or more scheduling segments, which is a code sequence whose execution is scheduled according to a set of application-specified scheduling parameters, e.g., worst-case execution time, deadline, and criticality of a real-time operation is used by the Maximum Urgency First (MUF) [4] scheduling strategy. These parameters can be associated with a segment encompassing that operation on a particular endsystem, *e.g.*, as shown for segment B in Figure 1. The code sequence that a scheduling segment comprises can include remote and/or local operation invocations. It is possible to adapt the values of parameters for the current scheduling policy within a given scheduling segment, but to adapt the policy itself, a new scheduling segment must be entered.

The `Current` interface. The `RTScheduling` module's `Current` interface defines operations on scheduling segments, including beginning scheduling segments (`begin_scheduling_segment()`), updating the values of their parameters (`update_scheduling_segment()`), and ending them (`end_scheduling_segment()`), as well as to create (`spawn()`) distributable threads [9]. Each scheduling segment keeps a unique instance of its `Current` object in local thread specific storage (TSS) [11] on each endsystem along the path of the distributable thread. Each nested scheduling segment keeps a reference to the `Current` instance of its enclosing scheduling segment.

Distributable thread location. Now that we have explained the terminology and interfaces for distributable threads, we can illustrate how the pieces fit together. A distributable thread may be entirely local to a host or it may span multiple hosts by making remote invocations. Figures 2 and 3 illustrate the different spans that are possible for distributable threads.

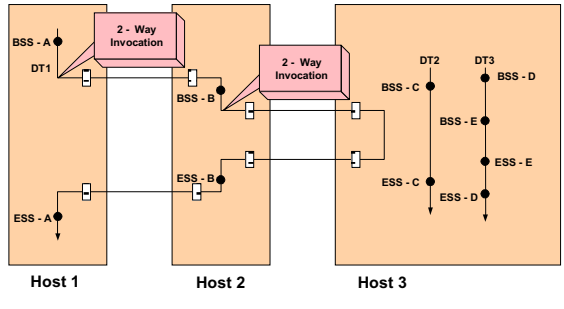

Figure 2: Distributable Threads and Hosts They Span

In these figures, calls made by the application are shown as solid dots, while calls made within the middleware (by *interceptors*, which are used to install upcalls to other services within mechanisms on the end-to-end method

invocation path [11]) are shown as shaded rectangles.

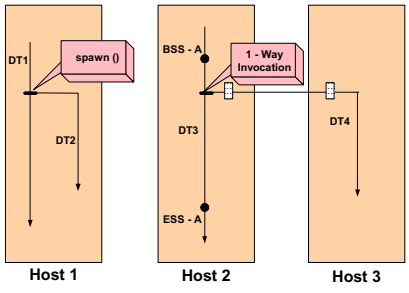

Figure 3: Ways to Spawn a Distributable Thread

In Figure 2, DT1 makes a twoway invocation on an object on a different host and also has a nested segment started on Host 2 (BSS-B to ESS-B within BSS-A to ESS-A). DT2 and DT3 are simple distributable threads that do not traverse host boundaries. DT2 has a single scheduling segment (BSS-C to ESS-C), while DT3 has a nested scheduling segment (BSS-E to ESS-E within BSS-D to ESS-D). In Figure 3, DT2 is created by the invocation of the `RTScheduling::Current::spawn()` operation within DT1, while DT4 is implicitly created on Host 2 to service a oneway invocation. DT4 is destroyed when the upcall completes on Host 2.

2.2 Pluggable Scheduling

Different distributable threads in a DRE system contend for shared resources, such as CPU cycles. To support the end-to-end QoS demands of open DRE systems, it is imperative that such contention be resolved predictably – yet the conditions under which that occurs may vary significantly at run-time. This tension between dynamic environments and predictable resource management necessitates scheduling and dispatching mechanisms for these entities that are (1) based on the real-time requirements of each individual system, and (2) sufficiently flexible to be applied adaptively in the face of varying application requirements and run-time conditions. The *pluggable scheduling* mechanisms in RTC1.2 help make DRE systems more adaptive since different scheduling strategies can be integrated corresponding to different application use cases and needs. In the RTC1.2 specification, a pluggable scheduling policy decides the sequence in which the distributable threads should be given access to endsystem resources and the dispatching mechanism grants the resources according to the sequence decided by the scheduling policy.

Various scheduling disciplines exist that require different scheduling parameters, such as MLF [4], EDF [12], MUF [4], or RMS+MLF [13]. One or more of these scheduling disciplines (or any other discipline the system developer chooses) may be used by an open DRE system to fulfill its scheduling requirements. Supporting this flexibility requires a mechanism by which different dynamic schedulers (each implementing one or more scheduling disciplines) can be plugged into an RTC1.2 implementation.

The RTC1.2 specification provides an IDL interface, `RTScheduling::Scheduler`, that has the semantics of an abstract class from which specific dynamic scheduler implementations can be derived. In RTC1.2, the dynamic scheduler is installed in the ORB and can be queried by passing the string `"RTScheduler"` to the standard `ORB::resolve_initial_references()` operation. The `RTScheduling::Manager` interface allows the application to install custom dynamic schedulers and obtain a reference to the one currently installed. The `RTScheduler_Manager` object can be obtained by passing the string `"RTScheduler_Manager"` to the `ORB:resolve_initial_references()` operation. The application then interacts with the installed RTC1.2 dynamic scheduler using operations defined in the `RTScheduling::Scheduler` interface described in Section 2.3.

2.3 Scheduling Points

To schedule distributable threads in a DRE system, an application and ORB interact with the RTC1.2 dynamic scheduler at well-defined points shown in Figure 4.

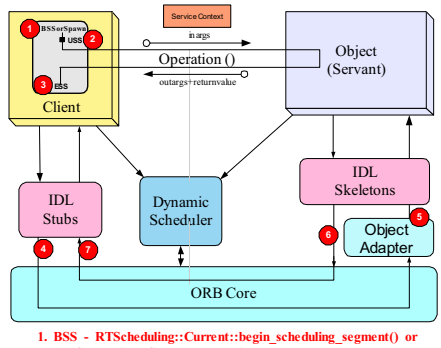

Figure 4: RTC1.2 Scheduling Points

These points can be defined *a priori* in more static systems, while in adaptive systems, the traversal of these points can be placed under adaptive control by the distributable threads, as described in Section 2.1. These scheduling points allow an application and ORB to provide the RTC1.2 dynamic scheduler up-to-the-instant information about the competing tasks in the system, so it can make scheduling decisions in a consistent, predictable, and adaptive manner. Scheduling points 1-3 in Figure 4 are points where an application interacts with the

RTC1.2 dynamic scheduler. The key application-level scheduling points and their characteristics are described below.

New distributable threads and segments. When a new scheduling segment or new distributable thread is created, the RTC1.2 dynamic scheduler must be informed so that it can schedule the new segment. The RTC1.2 dynamic scheduler schedules the new scheduling segment based on its parameters and those of the active scheduling segments for other distributable threads in the system. This occurs whenever code outside a distributable thread calls `begin_new_scheduling_segment()` to create a new distributable thread, or when code within a distributable thread calls `begin_nested_scheduling_segment()` to create a nested scheduling segment.

Changes to scheduling segment parameters. When `Current::update_scheduling_segment()` is invoked by a distributable thread to adapt its scheduling parameters, it updates scheduling parameters of the corresponding scheduling segment by calling `Scheduler::update_scheduling_segment()`.

Termination of a scheduling segment or distributable thread. The RTC1.2 dynamic scheduler should be called when `Current::end_scheduling_segment()` is invoked by a distributable thread to end a scheduling segment or when a distributable thread is cancelled, so it can reschedule the system accordingly. Hence, the `Current::end_scheduling_segment()` operation invokes the `end_scheduling_segment()` operation on the RTC1.2 dynamic scheduler to indicate when the outermost scheduling segment is terminated. The dynamic scheduler then reverts the thread to its original scheduling parameters. If a nested scheduling segment is terminated, the dynamic scheduler will invoke the `Scheduler::end_nested_scheduling_segment()` operation. The RTC1.2 dynamic scheduler then ends the scheduling segment and resets the distributable thread to the parameters of the enclosing scheduling segment scope.

A distributable thread can also be terminated by calling the `cancel()` operation on the distributable thread. When this call is made, `Scheduler::cancel()` is called automatically by the RTC1.2 framework, which allows the application to inform the RTC1.2 dynamic scheduler that a distributable thread has been cancelled.

Scheduling points 4-7 in Figure 4 are points where an ORB interacts with the RTC1.2 dynamic scheduler, i.e., when remote invocations are made between different hosts. Collocated invocations occur when the client and server reside in the same process. In collocated twoway invocations, the thread making the request also services the request. Unless a scheduling segment begins or ends at that point, therefore, the distributable thread need not be rescheduled by the RTC1.2 dynamic scheduler.

The ORB interacts with the RTC1.2 dynamic scheduler at points where the remote operation invocations are sent and received. Client- and server-side interceptors are therefore installed to allow interception requests as they are sent and received. These interceptors are required to (1) intercept where a new distributable thread is spawned in oneway operation invocations and create a new GUID for that thread on the server, (2) populate the service contexts, sent with the invocation with the GUID and required scheduling parameters of the distributable thread, (3) recreate distributable threads on the server, (4) perform cleanup operations for the distributable thread on the server when replies are sent back to a client for twoway operations, and (5) perform cleanup operations on the client when the replies from twoway operations are received. These interception points interact with the RTC1.2 dynamic scheduler so it can make appropriate scheduling decisions. The key RTC1.2 ORB-level scheduling points and their characteristics are described below.

Send request. When a remote operation invocation is made, the RTC1.2 dynamic scheduler must be informed so it can (1) populate the service context of the request to embed the appropriate scheduling parameters of the distributable thread and (2) potentially re-map the local thread associated with the distributable thread to service another distributable thread. As discussed in Section 2.4, when the distributable thread returns to that same ORB, it may be mapped to a different local thread than the one with which it was associated previously. The client request interceptor's `send_request()` operation is invoked automatically just before a request is sent, which in turn invokes `Scheduler::send_request()` with the scheduling parameters of the distributable thread that is making the request. The scheduling information in the service context of the invocation enables the RTC1.2 dynamic scheduler on the remote host to schedule the incoming request appropriately.

Receive request. When a request is received, the server request interceptor's `receive_request()` operation is invoked automatically by the RTC1.2 framework before the upcall to the servant is made, which in turn calls `Scheduler::receive_request()`, passing it the received service context that contains the GUID and scheduling parameters for the corresponding distributable thread. The RTC1.2 dynamic scheduler is responsible for unmarshaling the scheduling information in the service context that is received. The RTC1.2 dynamic scheduler uses this information to schedule the thread servicing the request and the ORB requires it to reconstruct a `RTScheduling::Current`, and hence a distributable thread, on the server.

Send reply. When a distributable thread returns via a

twoway reply to a host from which it migrated, the RTC1.2 framework calls the `send_reply()` operation on the server request interceptor just before the reply is sent. This operation in turn calls the `Scheduler::send_reply()` operation on the server-side RTC1.2 dynamic scheduler so it can perform any scheduling of the thread making the upcall as required by the scheduling discipline used so the next eligible distributable thread in the system is executed.

Receive reply. Distributable threads migrate across hosts via twoway calls. The distributable thread returns to the previous host, from where it migrated, through the reply of the twoway request. When the reply is received, the RTC1.2 framework calls the client request interceptor's `receive_reply()` operation, which in turn invokes `Scheduler::receive_reply()` on the client-side RTC1.2 dynamic scheduler to perform any scheduling related decisions required by the scheduling discipline.

2.4 RTC1.2 Implementation Challenges

To manage the behavior of distributable threads correctly, an RTC1.2 framework must resolve a number of design challenges, including (1) transferring ownership of storage, locks and other reserved system resources between local threads, (2) switching execution between distributable threads efficiently on an endsystem when one becomes more eligible than the other previously executing one, (3) dynamically scheduling and canceling non-critical operations when an endsystem is overloaded, and (4) adjusting operation invocation rates adaptively by coordinating multiple resource management services to avoid overload and maximize application benefit. We now summarize how TAO's RTC1.2 framework addresses these challenges. Section 3 then presents benchmarks and case studies that evaluate our solutions.

Distributable vs. OS thread identity. A key design issue with the RTC1.2 specification is that a distributable thread may be mapped to several different OS threads on each endsystem over its lifetime. Figure 5 illustrates how a distributable thread can use thread-specific storage (TSS), lock endsystem resources recursively so that they can be re-acquired later by that same distributable thread, or perform other operations that are sensitive to the identity of the distributable thread performing them. In this figure distributable thread DT1 (mapped to OS thread 1) writes data into TSS on Host 1 and then migrates to Host 2. Before DT1 migrates back to Host 1, DT2 migrates from Host 2 to Host 1. For efficiency, flexible concurrency strategies, such as thread pools [15], may map distributable threads to whatever local threads are available. For example, Figure 5 shows DT2 mapped to OS thread 1 and when DT1 migrates back to Host 1 it is mapped to OS thread 2.

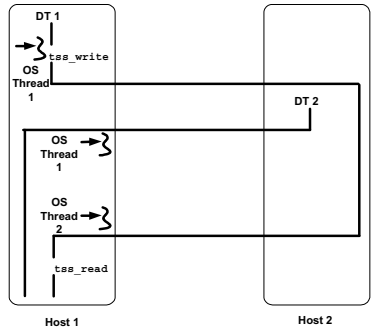

Figure 5: TSS with Distributable Threads

Problems can arise when DT1 wants to obtain the information it previously stored in TSS. If OS-level TSS was used, OS thread 2 cannot access the TSS for OS thread 1, so DT1's call to `tss_read()` in Figure 5 will fail. Moreover, OS-level TSS does not offer a way to substitute the OS thread identity used for a TSS call, even temporarily. To resolve these problems, a notion of distributable thread identity is needed that is separate from the identities of OS threads. Likewise, mechanisms are needed that use distributable thread GUIDs rather than OS thread IDs, resulting in an emulation of OS mechanisms in middleware that can incur additional overhead, which we quantify in Section 3.1.

Efficient and portable distributable thread dispatching. Two competing design forces – efficiency and portability – impact the decision of which dispatching mechanisms are the most suitable for distributable threads. Dispatching using OS thread priorities can offer low overhead for each individual context switch between threads. If a large number of thread schedule reordering decisions must be made, however, the OS thread priority approach may incur more calls from user space to the kernel (e.g., to manipulate thread priorities) and result in more context switches overall. Conversely, middleware-based reordering of distributable threads (e.g., in dispatching queues using condition variables to signal execution of a thread) may reduce the number of expensive calls from user space to the kernel, but may not offer as fine-grained control over dispatching as the kernel. Since some operating systems limit the number of thread priorities that can be assigned, middleware-based dispatching can also be more portable, especially for systems with very fine-grained adaptation requirements [20]. We compare the relative costs of middleware and OS dispatching approaches in Section 3.1.

Cancellation of operations and distributable threads. In addition to the issue of canceling entire distributable threads [9], the issue of canceling (and then possibly retrying) individual operations performed by distributable threads is also essential to consider in adaptive systems using dynamic scheduling techniques. For example, if an

individual operation in an end-to-end invocation chain will miss an interim local deadline on a particular endsystem, then dynamic schedulers in the RTC1.2 framework must be able to decide whether to cancel (and possibly retry) the operation or to allow it to execute anyway. If the operation is cancelled and retried, the distributable thread may miss its end-to-end deadline, but can free up the CPU so that other distributable threads may still meet their deadlines. We present results of our empirical studies of this issue for an avionics application in Section 3.2.

Efficient adaptive rescheduling of operation rates. In addition to mechanisms that support cancellation of individual distributable thread operations, adaptation of parameters (such as the invocation rate of each distributable thread) can increase a DRE system's ability to adapt to changing application QoS requirements or resource availability. This adaptivity is especially important in DRE systems using multiple layers of QoS management, where dynamic schedulers in the RTC1.2 framework can give higher QoS management layers a wider range of options for manipulating scheduling behavior at run-time. Section 3.3 describes such a multi-layer QoS management architecture and present results of studies of the interactions between dynamic schedulers and QoS managers.

3 Empirical Evaluation of RTC1.2 Dynamic Scheduling Issues in TAO

The studies in this section quantify the overhead of TAO's RTC1.2 dynamic scheduling framework and the effectiveness of key dynamic scheduling mechanisms. We first present micro-benchmarks of our RTC1.2 implementation described in Section 2. We also describe two case studies that apply our dynamic scheduling middleware in the context of real-time avionics computing systems running on production computing, communication, and avionics hardware. The first case study examines the effects of applying cancellation mechanisms to non-critical real-time operations and the second case study examines the performance of adaptive operation rate re-scheduling in a multi-layered resource management architecture for real-time image transfer.

3.1 Micro-benchmark Results

We conducted two micro-benchmarks to asses the performance of our RTC1.2 framework for different classes of DRE systems. These studies examined two primary concerns: the overhead of thread ID management and the responsiveness of distributable thread scheduling.

Overhead of thread ID management. Section 2.4 describes the challenges associated with emulating distributable thread identity via thread-specific storage (TSS) in middleware, rather than using OS-level TSS support. To quantify the additional overhead of TSS support in middleware, we conducted several experiments to compare and contrast the cost of creating TSS keys, and writing and reading TSS data on a single endsystem.

The experiments were conducted on a single-CPU 2.8 GHz Pentium 4 machine with 512KB cache and 512Mb RAM, running Red Hat Linux 9.0 (2.4.18 Kernel) with the KURT-Linux patches and using ACE version 5.3.2. The experiments were run as root, in the real-time scheduling class, and the experimental data were collected using the ACE high resolution timer. Experiments to assess the cost of TSS key creation were run by iteratively creating 500 different keys and measuring the time it took to create each one. The experiment to assess the cost of TSS write and read operations repeatedly wrote and then read from a storage location associated with one TSS key.

	slope (nsec/key)	**y-intercept (nsec)**
Emulated	2.53	2438
Native OS	1.61	815

Table 3: Cost of OS vs. Emulated TSS Key Creation

Table 3 summarizes the cost of creating the TSS keys, which our results showed was a linear function of the number of keys previously created. The key creation cost in middleware is higher than for creating TSS keys in the OS. Moreover, the slope at which the cost of key creation in middleware increased with each additional key was higher than the slope at which the cost of each additional key increased with OS-level TSS support. Similarly, Table 4 summarizes our results for read and write operations, which showed that the costs of `these` operations in middleware TSS emulation were again higher than in OS-level TSS, but did not increase with additional calls.

	read (nsec) mean/min/max	**write (nsec) mean/min/max**
Emulated	686.2 / 609 / 1203	1292.4 / 1203 / 1893
Native OS	70.8 / 68 / 101	77.6 / 71 / 130

Table 4: Cost of OS vs. Emulated TSS Read/Write Operations

Dynamic scheduling performance. To examine the ability of different scheduling mechanisms to respond adaptively to changes in parameters like task importance, we plugged two different implementations of the RTC1.2 `Scheduler` interface – a Thread Priority scheduler and a Most Important First scheduler – into the TAO ORB to test the behavior of its RTC1.2 dynamic scheduling framework with different scheduling strategies. Both implementations enforce a scheduling policy that prioritizes distributable threads according to their importance.

The *OS Thread Priority (TP) Scheduler* is a RTC1.2 implementation that schedules the distributable threads by mapping each one's dynamic importance to native OS priorities. The onus of dispatching the distributable threads is thus delegated to the OS-level thread scheduler, according to the native OS priorities assigned to the local threads to which the distributable threads are mapped. Conversely, the *Most Important First (MIF) Scheduler* uses a ready queue that stores distributable threads in order of their importance, with the most important distributable thread that is ready for execution at the head of the queue. The local OS thread to which each distributable thread in the queue is mapped then waits on a condition variable. When a distributable thread reaches the head of the queue (*i.e.*, is the next to be executed), the MIF scheduler awakens the corresponding local thread by signaling the condition variable on which it is waiting.

The experimental configuration we used to examine both the TP and MIF schedulers is identical. The test consisted of a set of local and distributed (spanning two hosts) distributable threads. The hosts were both running RedHat Linux 7.1 in the real-time scheduling class. The local distributable threads consisted of threads performing CPU bound work for a given execution time. The distributed distributable threads (1) performed local CPU bound work on the local host, (2) made a remote invocation performing CPU bound work on the remote host for a given execution time, and (3) came back to the local host to perform the remaining local CPU bound work.

Figure 6 shows how distributable threads are scheduled dynamically by the MIF scheduler (a nearly identical plot was seen for the TP scheduler) as they enter and leave the system across multiple hosts by the TP Scheduler and the MIF Scheduler respectively. The start times of the distributable threads are offset from T=0 by less than 1 sec, which is the time taken to initialize the experiments and start the distributable threads. On host 1, DT1 and DT2 start at time T= 0 and DT3 starts at T=12. Since DT1 is of higher importance than DT2 it is scheduled to run first. On host 2, DT4 starts at T=0 and is scheduled for execution as DT5 is not ready to run till T=9.

After executing for 3 secs DT1 makes a twoway operation invocation on host 2 and waits for a reply. DT4 is suspended to allow DT1 to execute on host 2 as DT1 is of higher importance. When DT1 is executing on host 2 the respective RTC1.2 dynamic scheduler on host 1 continues to schedule DT2 on host 1 as it has the highest importance on host1. DT1 completes execution on host 2 after T=3 and returns to host 1 and resumes execution, for T=3, after pre-empting DT2. DT4 resumes on host 2. DT1 completes its execution cycle of T=9 on host 1. Hence, DT2 is scheduled on host 1 for the next 3 secs. On host 2 DT5 enters the system at T=9. DT5 pre-empts DT4 as it is of higher importance and executes for 3 secs.

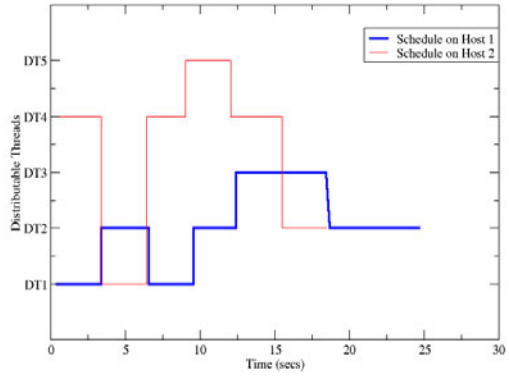

Figure 6: MIF Scheduler Dispatching Graph

At T=12 DT2 has completed 6 secs of local execution and makes a twoway invocation to host 2. DT3 enters the system at T=12 and is scheduled for execution for 3secs. On host 2, DT5 completes its cycle of execution and DT4 is scheduled. DT2 does not get to execute immediately on host 2 as DT5 is of higher importance. After DT4 completes execution on host 2, DT2 is scheduled on host 2 for the next 3 secs. DT3 continues execution on host 1. DT2 completes execution on host 2 and returns to host 1. DT3 completes its cycle of execution and DT2 is scheduled till its cycle of local execution is complete. Both the TP and MIF schedulers use the same scheduling *policy* based on the importance of the distributable threads. The graphs are therefore nearly identical, with one important distinction: in the very last scheduling decision on Host 1 (which switched from DT3 back to DT2), a larger delay occurred in the MIF scheduler for that switch, as reflected by the slope of the line in the graph at T==18 seconds which in the TP scheduler (and in all other transitions in both schedulers) was essentially vertical on that time scale.

The experimental results of the TP and MIF schedulers show that dynamic scheduling can be achieved with TAO's RTC1.2 framework when distributable threads migrate from endsystem to endsystem dynamically. Since both the TP and MIF schedulers schedule the distributable threads based on their importance, both graphs are nearly identical. The one small but important difference is in the times at which the threads are suspended and resumed, due to the context switch time for the MIF scheduler (which is at the application level) compared to the TP scheduler (which is at the OS level). These results validate our hypothesis that dynamic schedulers implementing different scheduling disciplines *and even using different scheduling mechanisms* can be plugged into TAO's RTC1.2 framework to schedule the distributable threads in the system according to a variety of requirements,

while maintaining reasonable efficiency.

3.2 Case Study 1 → Effects of Cancellation of non-Critical Operations

Overview and configuration. Many complex DRE systems perform a mixture of critical and non-critical real-time operations, for which it is desirable to maximize the ability of non-critical operations to meet their deadlines, while ensuring that *all* critical operations also meet their deadlines. When the CPU is overloaded (which can happen all too readily in open systems in dynamic operating environments), canceling some operations so that others are more likely to meet their deadlines is an important strategy for ensuring best use of CPU resources. In our first case study, we used an Operational Flight Program (OFP) system architecture based upon commercial hardware, software, standards, and practices that supports reuse of application components across multiple client platforms. The OFP is primarily concerned with integrating sensors and actuators throughout the aircraft with the cockpit information displays and controls used by the pilot and other aircraft personnel.

The system architecture for our first case study included an OFP consisting of approximately 70 operations, the Boeing Bold Stroke avionics domain-specific middleware layer [16] built upon The ACE ORB (TAO) [6], the TAO Reconfigurable Scheduling Service [7,8], and the TAO Real-Time Event Service [17], configured for various scheduling strategies described in Sections 2 and 3. This middleware isolates applications from the underlying hardware and OS, enabling hardware or OS advances to be integrated more easily with the avionics application.

We conducted measurements of two key areas of resource management: *cancellation of non-critical operations* that are at risk of missing their deadlines, and *protecting critical operations*. The analysis below features a comparison of two canonical scheduling strategies, the hybrid static/dynamic Maximum Urgency First (MUF) [4] strategy, which assigns operations to strict priority lanes according to their criticality and then schedules them dynamically within each lane according to laxity, and the static Rate Monotonic Scheduling (RMS) [12] strategy, which assigns operations to priority lanes according to their rates of invocation, and schedules each lane in FIFO order. Measurements were made on 200 MHz Power PC Single Board Computers running VxWorks 5.3.

Operation cancellation. Figure 7 shows the effects of canceling non-critical operations in the MUF hybrid static/dynamic scheduling strategy in conditions of CPU overload.

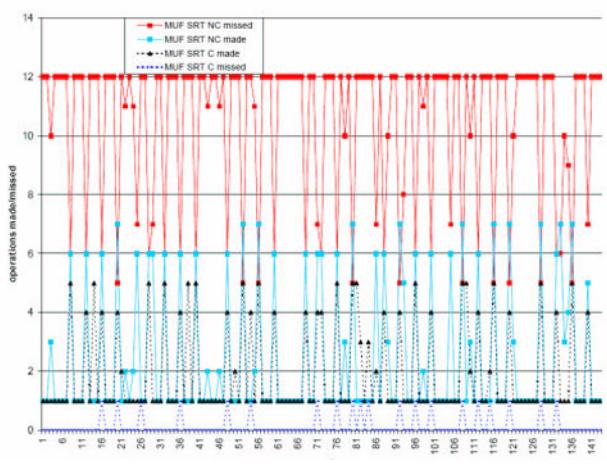

Figure 7: Effects of non-Critical Operation Cancellation

Operation cancellation can potentially help reduce the amount of wasted work performed in operations that miss their deadlines. This wasted time increases the amount of unusable overhead. We observed that while the MUF strategy with operation cancellation was more effective in limiting the number of operations that were dispatched and then missed their deadlines, the number of operations that made their deadlines in each case was comparable. We attribute this observation to the short execution times of several of the non-critical operations. In fact, the variation with cancellation had slightly lower numbers of non-critical operations that were successfully dispatched, as operation cancellation is necessarily pessimistic.

Protecting critical operations. We also compared the effects of non-critical operation cancellation on critical and non-critical operations under overload in the hybrid static/dynamic MUF and static RMS scheduling strategies. Figure 8 shows the number of deadlines made and missed for each strategy.

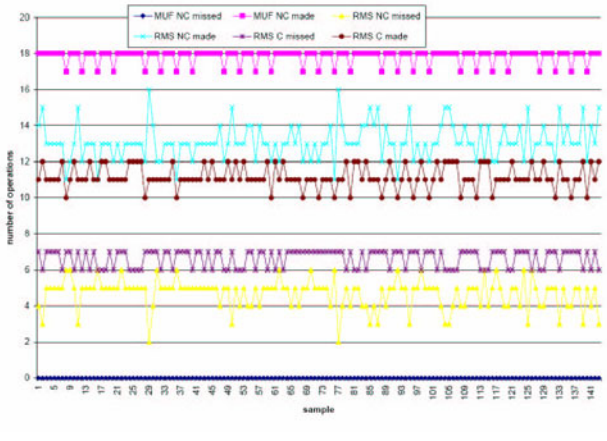

Figure 8: Effects of Cancellation under Overload Conditions

With no operation cancellation, MUF met all of its deadlines, while RMS missed between 2 and 6 critical operations per sample. Moreover, MUF successfully dispatched additional non-critical operations. We investigated whether adding operation cancellation might reduce the number of missed deadlines for critical operations with RMS, by reducing the amount of wasted work. It appears, however, that the overhead of operation cancellation actually makes matters worse, with between 6 and 7 misses per sample, which we interpret to mean that there were few opportunities for effective non-critical operation cancellation in RMS under the experimental conditions.

Case study analysis. The results from our first case study offer the following insights about the use of adaptive middleware to support DRE systems more effectively. First, operation cancellation can be an effective way to shed tasks that cannot meet their deadlines during resource overload. Second, these results indicate that hybrid static/dynamic scheduling strategies are more likely to benefit from operation cancellation than purely static ones since (1) hybrid static-dynamic scheduling strategies prioritize critical tasks as a whole over non-critical ones and (2) the availability of the CPU to non-critical tasks is more variable and more sparse so that more non-critical tasks are likely potential candidates for cancellation. Moreover, because operation cancellation is necessarily pessimistic, it is essential to avoid overestimating the risk of operations missing their deadlines, which can result in overly aggressive cancellation degrading – rather than improving – overall system performance. As usual, the more accurate the information that the cancellation mechanism has about deadline failure risks, the more accurate its cancellation decisions and the better its effect on overall system performance.

3.3 Case Study 2 → Adaptive Scheduling in Multi-level Resource Management

Overview and configuration. Our second case study examines the performance of adaptive rescheduling of operation rates within the context of multi-layered resource management [20]. We have applied the layered resource management architecture shown in Figure 9 to provide an open systems "bridge" between legacy on-board embedded avionics systems and off-board information sources and systems. The foundation of this bridge is the interaction of two Real-time CORBA [1] ORBs (TAO and ORBExpress) using a pluggable protocol to communicate over a very low (and variable) bandwidth Link-16 data network. Higher-level middleware technologies then manage key resources and ensure the timely exchange and processing of mission critical information. In combination, these techniques support browser-like connectivity between server and client nodes, with the added assurance of real-time performance in a resource-constrained and dynamic environment.

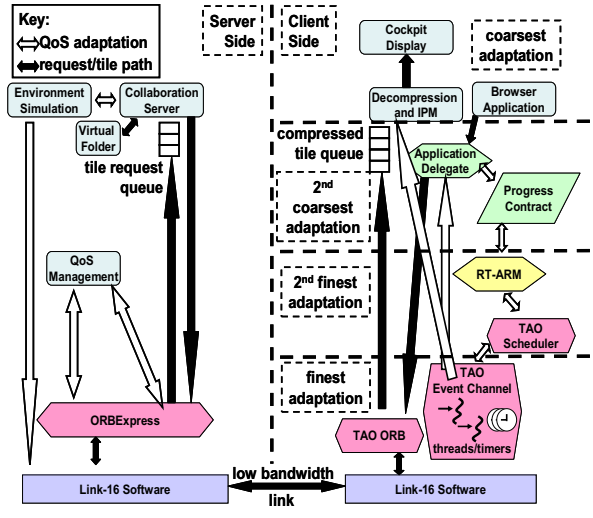

Figure 9: Multi-Level Resource Management Model

System resource management model. When a client operator requests an image, that request is sent from the *browser application* to an *application delegate* [20], which then sends a series of requests for individual tiles via TAO over a variable low-bandwidth Link-16 connection to the server. The delegate initially sends a burst of requests to fill the server request queue; it then sends a new request each time a tile is received. For each request, the delegate sends the tile's desired compression ratio, determined by the progress of the overall image download when the request is made.

On the server, an *ORBExpress/RT* (`www.ois.com`) Ada ORB receives each request from the Link-16 connection, and from there each tile goes into a *queue of pending tile requests*. A collaboration server pulls each request from that queue, fetches the tile from the server's *virtual folder* containing the image, and compresses the tile at the ratio specified in the request. The collaboration server then sends the compressed tile back through ORBExpress and across Link-16 to the client.

Server-side environmental simulation services emulate additional workloads that would be seen on the command and control (C2) server under realistic operating conditions. Back on the client, each compressed tile is received from Link-16 by TAO and delivered to a servant that places the tile in a queue where it waits to be processed. The tile is removed from the queue, decompressed, and then delivered by client-side operations to Image Presentation Module (IPM) hardware which renders the tile on the cockpit display. The decompression and IPM delivery operations are dispatched by a TAO Event Service [17] at rates selected in concert by the RT-ARM [18] and the TAO Reconfigurable Scheduler [7,8].

Schedule re-computation latency. We measured schedule recomputation overhead resulting from priority and rate reassignment by the TAO Reconfigurable Scheduler. Figure 10 plots schedule recomputations while the system is performing adaptation of both image tile compression and decompression and IPM operation rates, at deadlines for downloading the entire image of 48, 42, and 38 seconds. The key insight from these results is that the number and duration of re-scheduling computations is both (1) reduced overall compared to our earlier results [19] and (2) proportional to the degree of rate adaptation that is useful and necessary for each deadline.

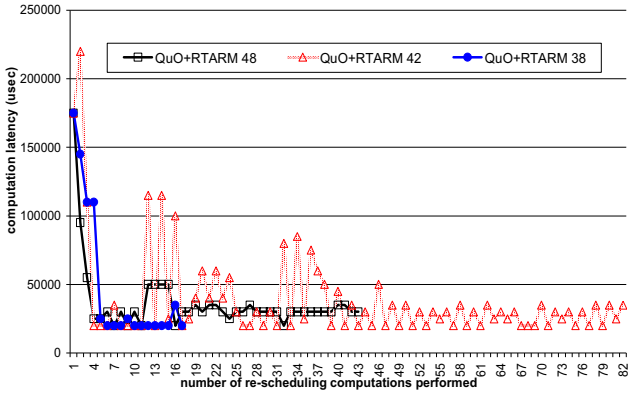

Figure 10: Adaptive Schedule Computation Latency

The main feature of interest in Figure 10 is the downward settling of schedule computation times, as ranges of available rates are narrowed toward a steady-state solution and the input set over which the scheduler performs its computation is thus reduced. We also observed a *phase transition* in the number of re-computations between the infeasible and barely feasible deadlines. If we arrange trials in descending order according to the number of re-computations in each, we get 42, 46, 48, 50, 52, 54, and then 58 seconds, and then finally 38 second and 1 second deadlines showed the same minimal number of computations. The duration of the experiment for the 42 second deadline was comparable to that for other deadlines.

Case study analysis. This case study demonstrates that adaptive rescheduling techniques can be applied to adjust the rates of operation invocation at run-time in response to dynamically varying environments. The convergence of the scheduling behavior toward lower latencies and smaller input sets is a good example of desirable adaptation performance in open DRE systems.

4 Related Work

The *Quality Objects* (QuO) distributed object middleware is developed at BBN Technologies [20]. QuO is based on CORBA and provides (1) *run-time performance tuning and configuration* and (2) *feedback* based on a control loop in which client applications and server objects request levels of service and are notified of changes in service. We have integrated our earlier dynamic scheduling service (Kokyu [23]) with the QuO framework in the context of the case study described in Section 3.3.

The RTC1.2 specification allows pluggable dynamic schedulers. However, this means that endsystems, or even segments within an endsystem, along an end-to-end path could be applying differing scheduling disciplines and scheduling parameters. For example, one endsystem could order the eligibility of distributable threads per the EDF scheduling discipline using deadlines, and another per the MUF scheduling discipline using criticality, deadlines, and execution times. RTC1.2 does not address the issue of interoperability of schedulers on the endsystems that a distributable thread spans. Juno [22], a meta-programming architecture for heterogeneous middleware interoperability, addresses the above issues. It formalizes the above problems, defines formalisms to express different instances of the problem and maps the formalisms to a software architecture based on Real-time CORBA.

5 Concluding Remarks

The OMG Real-time CORBA 1.2 (RTC1.2) specification defines a dynamic scheduling framework that enhances the development of open DRE systems that possess dynamic QoS requirements. The RTC1.2 framework provides a *distributable thread* capability that can support execution sequences requiring dynamic scheduling and enforce their QoS requirements based on scheduling parameters associated with them. RTC1.2 distributable threads can extend over as many hosts as the execution sequence may span. Flexible scheduling is achieved by plugging in dynamic schedulers that implement different scheduling strategies, such as MUF, or RMS+MLF, as well as the TP and MIF strategies described in Section 3.

TAO's RTC1.2 implementation has addressed broader issues than the standard covers, including mapping distributable and local thread identities, supporting static and dynamic scheduling, and defining efficient mechanisms for enforcing a variety of scheduling policies. We learned the following lessons from our experience developing and evaluating TAO's RTC1.2 framework:

- RTC1.2 is a good beginning towards addressing the dynamic scheduling issues in DRE systems. By integrating our earlier work on middleware scheduling frameworks [7,8,23] within the RTC1.2 standard, we have provided an even wider range of scheduling policies and mechanisms.
- Some features that are implemented for the efficiency of thread and other resource management can hinder the correct working of the RTC1.2 framework, e.g., managing distributable threads is more costly and

complicated due to the sensitivity of key mechanisms to their identities, as discussed in Section 2.4.

- System-wide dynamic scheduling is not yet as pervasive as fixed-priority static scheduling, which has limited the scope of the RTC1.2 specification, e.g., it does not yet address interoperability of dynamic schedulers on different hosts, but only ensures propagation of scheduling parameters across the hosts it spans so it can be scheduled on each host.
- Empirical case studies based on actual DRE systems (such as those presented in Section 4) are essential to (1) understand how techniques such as cancellation and adaptive rescheduling can be applied effectively in complex DRE systems and (2) determine the appropriate role of RTC1.2 mechanisms with respect to other middleware mechanisms that could be used.

References

[1] Real-Time CORBA 1.0 Specification, *Aug. 2002*, www.omg.org/docs/formal/02-08-02.pdf

[2] Karr, Rodrigues, Krishnamurthy, Pyarali, and Schmidt, "Application of the QuO Quality-of-Service Framework to a Distributed Video Application," *DOA*, Rome, Italy, Sept 2001.

[3] Corman, Gossett, Noll, "Experiences in a Distributed, Real-Time Avionics Domain - Weapons System Open Architecture", ISORC, Washington DC, April 2002.

[4] Stewart and Khosla, "Real-Time Scheduling of Sensor-Based Control Systems," in *Real-Time Programming* , Pergamon Press, 1992.

[5] Review Draft of the 1.2 revision to the Real-Time CORBA Specification (OMG document realtime/03-08-01). Previously designated Real-Time CORBA 2.0, www.omg.org/docs/ptc/01-08-34.pdf

[6] Schmidt, Levine, Mungee. "The Design and Performance of the TAO Real-Time Object Request Broker", *Computer Communications* 21(4), April 1998.

[7] Gill, Levine, Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems 20(2)*, Kluwer, March 2001.

[8] Gill, Schmidt, and Cytron, "Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing", IEEE Proceedings 91(1), Jan 2003.

[9] Krishnamurthy, Gill, Schmidt, Pyarali, Mgeta, Zhang, and Torri, "The Design and Implementation of Real-Time CORBA 2.0: Dynamic Scheduling in TAO", RTAS 2004, Montreal, Canada, May 2004.

[10] Leach and Salz, "UUIDs and GUIDs Internet-Draft", www.opengroup.org/dce/info/draft-leach-uuids-guids-01.txt

[11] Schmidt, Stal, Rohnert, and Buschmann, "Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects", Wiley, NY, 2000.

[12] Liu and Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, January 1973.

[13] Chung, Liu, Lin, "Scheduling Periodic Jobs that Allow Imprecise Results," *IEEE Transactions on Computers*, vol. 39, Sept 1990.

[14] Frisbee, Niehaus, Subramonian, and Gill, "Group Scheduling in Systems Software", 12th Workshop on Parallel and Distributed Real-Time Systems (at IPDPS), April 2004, Santa Fe, NM

[15] Pyarali, Schmidt, Cytron, "Techniques for Enhancing Real-Time CORBA Quality of Service", IEEE Proc., 91(7), July 2003.

[16] Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development", Software Technology Conference, April 1998.

[17] Harrison, Levine, and Schmidt, "The Design and Performance of a Real-time CORBA Event Service," OOPSLA, Atlanta, GA, Oct, 1997.

[18] Huang, Jha, Heimerdinger, Muhammad, Lauzac, Kannikeswaran, Schwan, Zhao, and Bettati, "RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications", Workshop on Middleware for Distributed Real-Time Systems, RTSS, San Francisco, CA, 1997.

[19] Doerr, Venturella, Jha, Gill, and Schmidt, "Adaptive Scheduling for Real-time, Embedded Information Systems," DASC, St. Louis, MO, Oct. 1999.

[20] Gossett, Gill, Loyall, Schmidt, Corman, Schantz, and Atighetchi, "Integrated Adaptive QoS Management in Middleware: A Case Study", RTAS, Montreal, Canada, May 2004.

[21] Zinky, Bakken, and Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, 3(1), 1997.

[22] Corsaro, Schmidt, Gill, and Cytron, "Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Policies in Open Distributed Real-Time Systems", DOA, Sept. 2001, Rome, Italy.

[23] Loyall, Gossett, Gill, Schantz, Zinky, Pal, Shapiro, Rodrigues, Atighetchi, and Karr, "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," ICDCS, April 2001, Phoenix, AZ.