Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing

Christopher D. Gill and Ron K. Cytron
{cdgill, cytron}@cs.wustl.edu
Department of Computer Science
Washington University, St.Louis

Douglas C. Schmidt
schmidt@uci.edu
Electrical & Computer Engineering
University of California, Irvine

Abstract

Increasingly complex requirements, coupled with with tighter economic and organizational constraints, are making it hard to build complex distributed real-time embedded (DRE) systems entirely from scratch. The proportion of DRE systems made up of commercial-off-the-shelf (COTS) hardware and software is therefore increasing significantly. There are relatively few systematic empirical studies, however, that illustrate how suitable COTS-based hardware and software have become for mission-critical DRE systems.

This paper provides the following contributions to the study of real-time quality of service (QoS) assurance and performance in COTS-based DRE systems: (1) it presents evidence that flexible configuration of COTS middleware mechanisms, and the operating system settings they use, allows DRE systems to meet critical QoS requirements over a wider range of load and jitter conditions than statically configured systems, (2) it shows that in addition to making critical QoS assurances, non-critical QoS performance can be improved through flexible support for alternative scheduling strategies, and (3) it presents an empirical study of three canonical scheduling strategies—specifically the conditions that predict success of a strategy for a production-quality DRE avionics mission computing system. Our results show that applying a flexible scheduling framework to COTS hardware, operating systems, and middleware improves real-time QoS assurance and performance for mission-critical DRE systems.

Keywords: Middleware and APIs, Quality of Service Issues, Distributed Real-time and Embedded Systems, Mission Critical Systems, Dynamic Scheduling Algorithms and Analysis.

I. Introduction

A. Emerging System Demands

Distributed, real-time, and embedded (DRE) systems are becoming increasingly widespread and important. Examples of DRE systems include *telecommunication networks*, *e.g.*, wireless phone services, *tele-medicine*, *e.g.*, remote surgery, *manufacturing process automation*, *e.g.*, hot rolling mills, and *defense systems*, *e.g.*, avionics mission computing systems. Although there are many types of DRE systems, they have one thing in common: *the right answer delivered too late becomes the wrong answer.* More specifically, DRE systems have the following types of requirements:

- As *distributed systems*, DRE systems require capabilities to manage connections and data transfer between separate computers.
- As *real-time systems*, DRE systems require predictable and efficient control over end-to-end system resources.
- As *embedded systems*, DRE systems have weight, cost, and power constraints that limit their computing and memory resources.

Designing DRE systems that implement their required capabilities, are dependable, and are parsimonious in their use of limited computing resources is hard; building them on time and within budget is even harder. A particularly essential task is supporting the quality of service (QoS) demands of mission-critical DRE systems that possess a mix of hard and soft real-time requirements, such as avionics mission computing systems [1], mission-critical distributed audio/video processing [2], [3], and real-time robotic systems [4].

This work was supported in part by Boeing, DARPA ITO, DARPA contract F33615-00-C-1697 (PCES) and AFRL contracts F3615-97-D-1155/DO (WSOA) and F33645-97-D-1155 (ASTD/ASFD).

B. Key Challenges: *Flexibility* and *QoS Assurance*

DRE systems have historically been custom developed in an *ad hoc* and inflexible manner. While many operational systems have been built this way, this development process failed to address the following challenges adequately:

Reducing total ownership costs: Custom software development and evolution is labor-intensive and error-prone for complex DRE systems, and can represent a substantial fraction of system lifecycle costs. Moreover, incommensurate lifetimes between long-lived DRE systems ($>=$ 20 years) and COTS platforms and tools (2–5 years) lead to pervasive software obsolescence that multiply total ownership costs by requiring periodic software redevelopment and COTS refresh.

Portable QoS management: Modern DRE systems must invest an ever-increasing proportion of functionality and QoS management in software. Rapidly emerging technologies and flexibility required for diverse operational contexts force deployment of multiple software versions on various platforms, while simultaneously preserving key QoS properties, such as real-time response and end-to-end priority preservation.

Dependence on rigid assumptions: Custom DRE systems are scheduled inflexibly so that if assumptions about the *total* resource load are violated, critical real-time constraints may be violated. Unfortunately this leads to provisioning of resources at levels that are both (1) excessive compared to what is needed to assure the minimum *critical* system requirements and (2) unrecoverable to improve average case performance.

Insufficient responsiveness to varying operating environments: Custom DRE systems make rigid assumptions about system load and load jitter that can in unexpectedly varying environments lead to (1) a violation of critical QoS requirements, and/or (2) reduced performance in meeting non-critical QoS requirements. While static scheduling might be replaced with dynamic scheduling in some systems, any *single-paradigm* approach will naturally suffer these same limitations.

some aspects of the total ownership cost challenges Outlined above are being addressed for business applications by COTS software, such as SOAP/.NET and J2EE. Until recently, however, little has been done to simultaneously meet all of these challenges for mission-critical DRE systems.

C. A Promising Approach: Real-time CORBA Middleware

Over the past several years, a promising solution to many of the challenges outlined above has emerged in the form of *distributed object computing (DOC) middleware*. DOC middleware is systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware [5].

Its primary role is to allow clients to invoke operations on target object implementations without concern for where the object resides, what language the object implementations are written in, the OS/hardware platform, or the types of communication protocols, networks, and buses used to interconnect distributed applications [6].

Real-time CORBA [7] is a DOC middleware standard that adds QoS control capabilities to the original CORBA specification by (1) improving system predictability and bounding priority inversions and (2) managing system resources end-to-end. At the heart of Real-time CORBA is an Object Request Broker (ORB) that provides run-time support to automate many DRE computing tasks, such as connection management, marshaling/demarshaling, demultiplexing, language and OS independence, resource scheduling and load balancing, error handling and fault-tolerance, and security.

First-generation ORBs did not provide features or optimizations to support DRE systems with stringent QoS requirements. To better meet these requirements, researchers at Washington University St. Louis and the University of California, Irvine have developed a second-generation ORB called TAO [8], which is an open-source implementation of Real-time CORBA that supports efficient, predictable, and flexible DRE computing. Prior work on TAO has explored many dimensions of high-performance and real-time ORB design and performance, including scalable event processing [9], request demultiplexing [10], I/O subsystem [11] and protocol [12] integration, connection architectures [13], asynchronous [14] and synchronous [15] concurrent request processing, adaptive load balancing [16], meta-programming mechanisms [17], and IDL stub/skeleton optimizations [18].

TAO isolates DRE systems from platform-specific QoS enforcement mechanisms by encapsulating a robust QoS framework for managing end-to-end resources within a standard set of CORBA interfaces. TAO also reduces DRE system dependence on rigid assumptions by enabling alternative policies and mechanisms to be plugged into its QoS framework. In fact, the Real-time CORBA 1.0 specification and its implementation in TAO address all the DRE system challenges outlined in Section I-B *except* for insufficient responsiveness to varying operational environments. The reason for this omission is because no *single* scheduling paradigm performs best in all environments, which motivates our research in this paper on the design and performance of flexible scheduling frameworks for DRE middleware and applications.

D. An Inclusive Solution: Multi-paradigm Scheduling

This paper extends our previous work on static [8] and dynamic [1] scheduling for Real-time CORBA by incorporating a *strategized scheduling framework* called *Kokyu*[1] as a service atop TAO. Kokyu enables the configuration and empirical evaluation of multiple scheduling paradigms, including:

- **Static** scheduling strategies, *e.g.*, rate monotonic scheduling (RMS) [19],
- **Dynamic** scheduling strategies, *e.g.*, earliest deadline first (EDF) [19] and minimum laxity first (MLF) [4], and
- **Hybrid static/dynamic** scheduling strategies, *e.g.*, maximum urgency first (MUF) [4] and RMS+MLF [20].

Kokyu is applicable to an important class of demanding real-world DRE systems, which includes avionics mission computing [21], [22], mission-critical distributed audio/video processing [2], [3], and real-time robotic systems [4]. To maintain scheduling assurances and simplify testing for these types of systems, we have enhanced our prior work [1], [8] to focus on DRE systems with the following characteristics:

- **Bounded execution time**, where the use of resources during each execution of a resource request stays within the limit of its specified duration.
- **Bounded rates**, where resource requests arrive within a specified period.
- **Known operations**, where all operations are visible to the scheduler prior to scheduling, or are reflected entirely within the execution times of other specified operations.
- **Critical and non-critical operations**, where deadlines of all critical operations must be assured, and non-critical deadlines should be met to the extent possible.

Real-time QoS requirements of DRE systems with these characteristics have been addressed historically by scheduling tasks within a *single paradigm*, such as:

- **Static scheduling**, that assigns priorities to *all* tasks statically and ensuring the task with the highest *fixed* priority always runs [19], [23], or
- **Dynamic scheduling**, that orders *all* tasks dynamically and ensuring the task with the highest *dynamic* priority is dispatched preferentially [19], [4].

Static scheduling can minimize overhead stemming from, *e.g.*, dispatching and admission control mechanisms, while dynamic scheduling requires less *a priori* knowledge of operation characteristics, *e.g.*, rates of execution. However, using either of these scheduling paradigms *alone* imposes the following limitations:

1) It does not isolate critical and non-critical load,
2) It is brittle in the face of total load in excess of the feasible limit, even though critical load is below that limit, and
3) It is thus insufficiently responsive to variations in demands by the application or operating environment.

A hybrid static/dynamic scheduling paradigm used by the MUF [4] and RMS+MLF [20] strategies has been proposed to (1) *partition* critical and non-critical resource utilization using static mechanisms such as thread priorities, and then (2) dynamically schedule operations within one [20] or more [4] partitions. The hybrid static/dynamic scheduling paradigm can therefore assure feasible critical deadlines will be met, even when when total load is infeasible. When the total load is feasible, however, the additional overhead imposed by hybrid static/dynamic scheduling means that fewer non-critical deadlines can be met than in static scheduling.

To alleviate the drawbacks of single-paradigm scheduling—while still preserving its key benefits—our work with the Kokyu framework described in this paper allows DRE systems to specify *multi-paradigm* scheduling strategies that trade a small additional amount of overhead for increased flexibility in (1) assuring critical QoS requirements and (2) enhancing the availability of resources to improve non-critical performance. In particular, we present foundational work towards strategies that can enforce each preferred single-paradigm strategy along the entire range of resource utilization.

Figure 1 illustrates the benefits of the Kokyu multi-paradigm approach. The upper solid curved line shows a hypothetical ideal utilization of resources as system load increases. The solid square line illustrates static single-paradigm strategies, such as RMS, that can approach the ideal under certain conditions, but may miss critical assurances beyond a certain limit, which is illustrated by the utilization value dropping to zero. Similarly, purely dynamic approaches may offer feasibility improvements under special cases, *e.g.*, when rates are non-harmonic, yet the additional overhead they impose may result in missed critical assurances at an even lower level of load. Hybrid static-dynamic approaches, in contrast, offer feasibility along the length of the load axis (as long as the critical load is feasible),

[1] Kokyu is a Japanese word meaning literally "breath", but also implying timing and coordination.

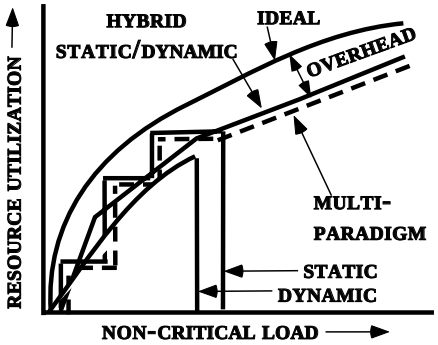

Fig. 1. Ideal, Static, Dynamic, and Hybrid Paradigms

and exhibit overhead that is intermediate between purely static and purely dynamic approaches.

The dashed curve in Figure 1 shows how multi-paradigm scheduling can approximate the best single-paradigm approach at each point along the horizontal load axis. Due to mode switches or other adaptation mechanisms, multi-paradigm approaches may incur more overhead than static and hybrid static/dynamic single-paradigm approaches. They are better suited than single-paradigm approaches, however, to approximate the ideal performance curve over its length.

This paper shows how the Kokyu framework supports alternative scheduling strategies implemented using COTS OS and middleware mechanisms. By doing so, Kokyu increases adaptability across product families, operating systems, and most importantly environmental conditions, while preserving the rigorous scheduling guarantees and testability offered by prior work on statically scheduled CORBA operations [8], [21], [22].

E. Paper Organization

The remainder of this paper is organized as follows: Section II describes the application, middleware, OS, and hardware configurations that comprise the open experimentation platform used for our empirical studies; Section III describes how our experiments quantitatively evaluate the suitability of COTS-based hardware and software for mission-critical DRE systems; Section IV presents the empirical results obtained on our open experimentation platform; Section V summarizes the observations and recommendations based on our results; Section VI compares our research on Kokyu with related work; and Section VII presents concluding remarks.

II. Open Experimentation Platform

The work in this paper focuses on a mission-critical system that is representative of an important class of DRE systems: *the operational flight program (OFP) in an avionics mission computing system*. An OFP manages sensors and operator displays, navigates the aircraft's course, and controls on-board equipment. The avionics system used for this paper consists of OFP components hosted on a domain-specific middleware infrastructure called *Bold Stroke*, which in turn is built using the distribution middleware capabilities and common middleware services provided by the TAO Real-time CORBA ORB [8].

Figure 2 illustrates the interactions between the Kokyu framework and OFP application and middleware components. Along with Figure 3 in Section II-A, this figure shows how the OFP application components were hosted on an open experimentation platform consisting of the following layers:

- An OS/hardware platform consisting of the VxWorks real-time operating system on embedded hardware, which is described in Section II-A.

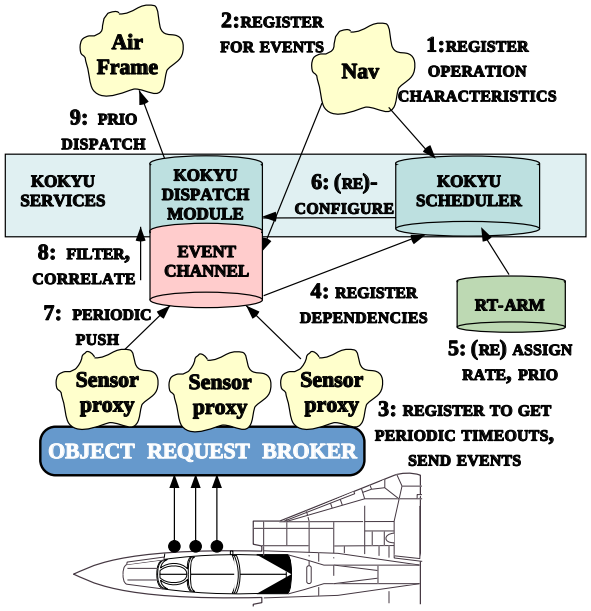

Fig. 2. Application and Middleware Layers

- The ACE ORB (TAO) [8], the TAO real-time *event channel* [9], and the Kokyu strategized scheduler [1] middleware, which is described in Section II-B.
- The Bold Stroke avionics domain-specific middleware [21], [22], which is described in Section II-C.
- The OFP application components used for the studies, which are described in Section II-D.

The remainder of this section describes these layers of the open experimentation platform.[2] Sidebar 1 defines key terminology used throughout the paper.

A. Overview of OS/Hardware Configurations

Figure 3 shows the COTS hardware and operating system used in the experiments described in Section III, consisting of a commercial VME-64 chassis with four commercial processor cards, a desktop computer running Windows NT 4.0, and a portable UNIX workstation. The desktop computer gathered metrics data and presented visualizations of processor utilization and deadline successes, failures, and cancellations. The UNIX workstation loaded the executable programs onto the boards in the VME chassis and provided a file server for the digital map display.

Two COTS processor cards, a Dy4-783 and a Dy4-177, performed the map display function. The Dy4-783 card had a memory-mapped display processor and the Dy4-177 card hosted an application component that ran the map display algorithms. The OFP system was distributed across the remaining two processor cards. The first system card was a 200 MHz, PowerPC 604, Motorola card, which ran the experimental system described in Section II-D on the VxWorks [24] 5.3.1 real-time operating system. The second system card was a 100 MHz, PowerPC 603, Dy4-177 card. This card contained a MIL-STD-1553 MUX bus interface card and the Ethernet interface for the VME chassis. All external communication, *e.g.*, over the 1553 bus to avionics remote terminals, or over the VME backplane to diagnostic and debug systems, went through this card. This card also controlled timing for frame sequencing and display updates, upon which operation rates on the Motorola card depended.

[2]This platform, and the studies conducted on it, were supported under the Adaptive Software Flight Demonstration (ASFD) program hosted by the Boeing Phantom Works Open Systems Architecture organization. This work was administered by the Embedded Systems Branch of the Information Directorate, Air Force Research Labs (AFRL), Wright-Patterson Air Force Base, Dayton, Ohio. Portions of the TAO ORB and the Bold Stroke open experimentation platform were developed under support from DARPA ITO.

Sidebar 1: Terminology

For clarity, we define the following terms used in the discussion of the Bold Stroke open experimentation platform:

- **Operation**—A single short-lived computation run each time an event is pushed to its component.
- **Cancellation**—Interdiction of the event push to an operation so that it will not be invoked. We denote scheduling strategies using cancellation by a © annotation in Section IV.
- **Load chain**—A sequence of operations, where each operation itself (except the last one) pushes an event to invoke the next operation in the chain. Subsequent events have precedence dependencies on prior events in the chain, and cancelling an operation in the chain amounts to shedding the rest of the chain from that operation onward.
- **Route leg**—A segment of a navigation route computed in one operation invocation. Computing route legs was implemented as a load chain in our experiments, with each route segment successfully completed requesting the next segment, up to the length of the chain. In particular, a realistic system might declare the computation of the first one or two legs to be critical operations, that must be completed on time and cannot be cancelled, while subsequent route legs might be declared non-critical.
- **Replication service**—A middleware service provided by the Boeing Bold Stroke infrastructure for replicating data across mission-computing processors. Operation deadlines in the experimental system correspond to the points in time when their respective output values must be delivered and flushed to the replication service.
- **Remote terminals**—Connected sensors and actuators in the aircraft. In the open experimentation platform, emulation software for these was connected to the mission computer by a MIL-STD-1553 hardware bus, to simulate the inputs of actual sensors. The experimental system, middleware, and hardware were demonstrated in an AV-8B flight simulator at Boeing, which included an AV-8B cockpit and hardware remote terminals.

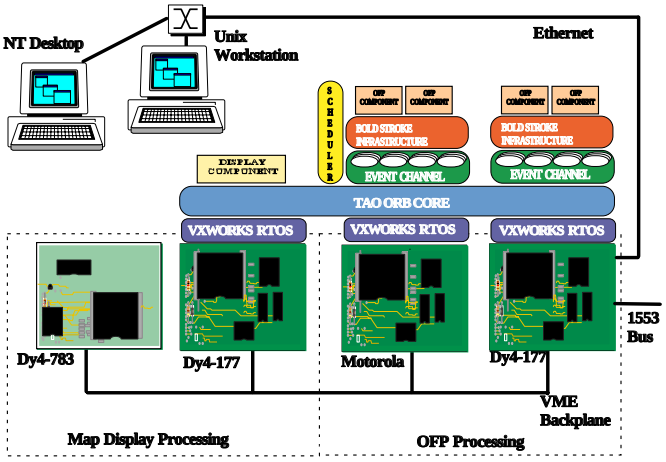

Fig. 3. Hardware and Software Configuration

B. Overview of DOC Middleware Configurations

The COTS distributed object computing middleware used for the ASFD demonstration were based on the TAO 1.2 implementation of Real-time CORBA [8], [7]. Real-time CORBA allows DRE developers to configure and control the following system resources:

- **Processor resources** via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service for real-time systems with fixed priorities
- **Communication resources** via protocol properties and explicit bindings to server objects using priority bands and private connections and
- **Memory resources** via buffering requests in queues and bounding the size of thread pools.

As shown in Figure 2, the TAO Real-time Event Channel [9] is a publish/subscribe service that mediates communication between components acting as proxies for (1) remote terminals that interact with the physical environment and (2) the operations that process the data. Sensor proxies flush relevant data to the replication service and then *push* events through the Real-time Event Channel to the processing operations.

Figure 2 also shows the Kokyu scheduling framework, which is a CORBA service that provides scheduling and dispatching services to TAO's Real-time Event Channel. Kokyu is responsible for (1) isolating critical processing from non-critical processing and (2) making the remaining CPU time available to non-critical processing. Kokyu provides these services via a scheduling strategy with which it is configured to (1) assign priorities to operations and (2) to specify the queueing discipline used at each priority level. By configuring the TAO Real-time Event Channel according to the specified set of priorities and queue disciplines, the middleware services described above enforce the mission computing system's real-time QoS assurances and performance.

C. Overview of the Bold Stroke Platform

The open experimentation platform for our work is based on the Bold Stroke domain-specific middleware [21], [22]. Bold Stroke uses COTS hardware and middleware to produce a standards-based component architecture for military avionics mission computing capabilities, such as navigation, data link management, and weapons control. A driving objective of Bold Stroke is to support reusable product-line applications, leading to a highly configurable application component model and supporting reusable middleware services, such as a replication service.

Bold Stroke has been developed and deployed using DOC middleware components and services based on the TAO Real-time ORB and Real-time Event Channel, and the Kokyu framework described in Section II-B. Figure 2 illustrates the middleware components in Bold Stroke. As shown in this figure, Bold Stroke uses TAO Real-time Event Channel atop the TAO ORB to communicate between components (1) on the same endsystem and (2) distributed across different endsystems. The Kokyu scheduler maintains information required for priority-preserving dispatching, which in the experimental framework described in Section III was performed in dispatching queues within the TAO Real-time Event Channel.

D. Overview of the OFP Application

The OFP application used as the basis of our multi-paradigm scheduling experiments provides avionics mission computing capabilities for an AV-8B (Harrier) aircraft. The baseline version evolved from

1) An AV-8B OFP written in assembly language, to
2) A single-board C/C++ OFP, and subsequently to
3) A distributed OFP using the Boeing AV-8 Open Systems Core Avionics Requirements airframe and the Boeing Bold Stroke domain-specific middleware described in Section II-C.

All major OFP components were implemented as periodically invoked operations, executed by event consumers. Operations were divided into two equivalence classes:

- **Hard real-time (HRT) for critical operations**—Critical operations in the HRT class are those whose failure to meet any given deadline has potentially significant consequences for the correctness of the application.

- **Soft real-time (SRT) for non-critical operations**—Deadline success for the non-critical SRT operations is desirable but not strictly mandatory.

There were five pre-defined rates of execution in the system: 40 Hz, 20 Hz, 10 Hz, 5 Hz, and 1 Hz. Each operation runs at one of these rates. For the ASFD open experimentation platform, new 20 Hz SRT functions were added to the OFP, including routes and steering components, as well as a digital map display.

III. Experimental Framework to Evaluate Multi-Paradigm Scheduling

Section II outlined the Bold Stroke architecture and the OFP application components for avionics mission computing. This section describes the design of experiments that empirically evaluate the suitability of COTS-based hardware and software for these types of mission-critical DRE systems. We focus on three canonical scheduling strategies—Rate Monotonic Scheduling (RMS) [19], Maximum Urgency First (MUF) [4], and RMS+Minimum Laxity First (MLF) [20]—to determine which performs better under representative environmental conditions with varying *load* and *load jitter*.

A. OFP Application Design and Implementation Challenges

Challenges addressed by Bold Stroke: The Bold Stroke architecture addresses the following key design and implementation challenges confronted by OFP applications:

a) Scheduling assurance of critical operations is required prior to run-time: In OFP applications, as in many other DRE systems, the consequences of missing a deadline at run-time can be catastrophic. For example, failure to process an input from the pilot by a specified deadline can be disastrous in an avionics application, *e.g.*, during navigation through a dense threat environment. It is therefore essential to assure *prior to run-time* that even in the worst-case scenario(s), all critical processing deadlines will be met. Bold Stroke has historically addressed this challenge through static scheduling and extensive testing and validation.

b) Severe resource limitations: Like many other DRE systems, OFP applications must perform *efficient* processing due to strict resource constraints, such as cost, weight, and power consumption restrictions. In particular, it is desirable to provision only the resources needed to meet worst-case critical processing requirements. Bold Stroke has historically addressed this challenge by clustering operations within an OFP application into a set of coarse-grain mutually exclusive *modes*, and provisioning resources for the worst-case mode.

c) Adaptability across product families: Some DRE real-time systems are custom-built for specific product families. Development and testing costs can be reduced if critical and non-critical resource requirements can be shown to be isolated. In addition, validation and certification of components can be shared across product families, which amortizes development time and effort. Bold Stroke addresses this challenge by using CORBA to separate interfaces from implementations and support component reuse [8].

Challenges addressed by Kokyu: We apply the Kokyu scheduling framework to the Bold Stroke architecture to address the above challenges in a broader range of contexts, as described in Section IV. Furthermore, Kokyu addresses the following design and implementation challenges confronted by OFP applications, but not addressed historically by the Bold Stroke platform itself:

d) Robust performance under widely varying environmental conditions: As noted in Section I, next-generation DRE systems must repond flexibly to variations in load and load jitter imposed by the external environment. For example, next-generation avionics mission computing applications implement features, such as on-demand imagery download [2] and decision aiding systems [25], whose resource demands (1) vary total load at longer time-scales across a series of stable epochs of operation, according to inputs from the environment and/or human users and (2) produce different degrees of load jitter in invocation-to-invocation demands across shorter time-scales within each epoch according to relevant factors, such as progress of a navigation computation in a rapidly evolving threat environment.

e) Safe addition of non-critical processing: To more fully occupy under-utilized resources in non-worst-case scenarios, it is desirable to perform additional non-critical processing. While missing a non-critical operation's deadline does not compromise system correctness, reduced or even zero value accrues to the application for that operation's use of the resources. It is crucial, however, to assure that non-critical processing does not interfere with critical processing and cause critical deadlines to be missed.

These design and implementation challenges addressed by Bold Stroke and Kokyu are also fundamental to many other DRE systems with similar requirements and constraints. Our previous work [1] described the design and implementation challenges we addressed to apply Kokyu to Real-time CORBA and thus integrate Kokyu within the Bold Stroke architecture. This paper extends our earlier work by presenting empirical studies that show how Kokyu can then meet the above open challenges not historically addressed by Bold Stroke. The results in this paper can be generalized to a broader class of DRE systems that perform both critical and non-critical processing and that operate in dynamically varying environments.

B. Experimental Design

We have applied the open experimental platform described in Section II to determine the degree to which the challenges described in Section III-A can be met (1) using Commercial off-the-shelf (COTS) hardware, operating systems, and middleware (*i.e.*, using Dy4 and Motorola cards, the VxWorks OS, and the TAO, TAO Real-time Event Channel, and Kokyu middleware) and (2) across a range of environmental conditions. The remainder of this section describes the hypotheses tested, the variables that were controlled, and the variables that were measured in our studies.

1) Hypotheses: The hypotheses explored in these studies are shown in Table I. This table also notes which challenges described in Section III-A are addressed by each hypothesis. To test these hypotheses, and to study the potential benefits and consequences of (1) supporting alternative scheduling strategies and (2) working toward the ability to perform beneficial adaptation among them at run-time, we ran identical trials using each of the following canonical scheduling strategies:

Hypothesis	Challenges
Multi-paradigm scheduling is needed to both (1) maintain QoS assurances for DRE systems while (2) increasing performance beyond levels achievable by single-paradigm approaches.	A, B, and D
Infrastructure factors, such as dynamic queue overhead, may influence both the ability to enforce critical processing *assurances*, and the ability to improve non-critical processing *performance*.	C and E

TABLE I
Hypotheses Studied and Challenges Addressed

- RMS [19], which is a purely static strategy that assigns priorities in rate order and manages requests at each priority level in first-in-first-out (FIFO) order.
- MUF [4], which is a hybrid static/dynamic strategy that assigns static priorities by operation criticality, and schedules within each static priority by minimum laxity.

- RMS+MLF [20], which first schedules critical operations according to rate and then non-critical operations at lower priority according to laxity.

We selected these strategies since they are most applicable to OFP application requirements to support both hard real-time (HRT) and soft real-time (SRT) operations under a range of load and load jitter conditions.

2) Controlled Variables: To examine effects of varying load and load jitter in the production-quality avionics mission computing environment described in Section III-A, many next-generation DRE systems must satisfy resource demands that

- Vary overall at longer time-scales across a series of stable epochs of operation and
- Produce different degrees of jitter in invocation-to-invocation demands across shorter time-scales within each epoch.

To model variation in both load and load jitter imposed by these types of demands, we added operations to a sequence of twelve epochs of operation, each representing a distinct *operating region* [2] numbered 0–11, as shown in Figure 4.

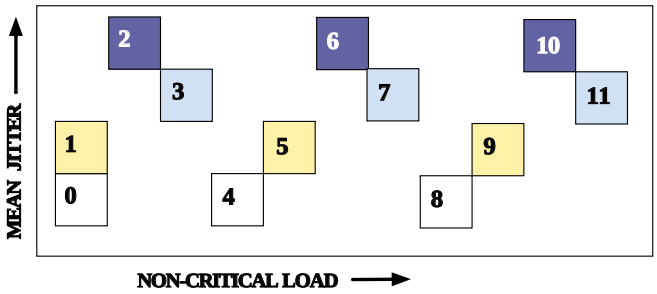

Fig. 4. Operating Regions

In addition to the fixed OFP operations, which were present and active in each operating region, we introduced chains of additional 20 Hz SRT route leg updates (see Sidebar 1) to each operating region. We varied the length of the request chain to move from lowest to highest *fundamental* non-critical load. We did this incrementally from region 1 to region 11, while keeping the fundamental critical load constant across operating regions. We kept the non-critical load the same in region 0 and region 1 to ensure that we compared the effects of two different levels of jitter with no change in fundamental load in at least one case.

To examine the effects of (1) varying levels of load jitter across similar fundamental loads and (2) similar levels of jitter across varying non-critical loads, we added an additional HRT event consumer to the second card at each of the following rates: 10 Hz, 5 Hz, and 1 Hz HRT. The additional operations acted in these experiments as surrogates for the workload variation that would normally be associated with a distributed production OFP. The CPU utilization by these additional HRT event consumers was randomized across a given range in each operating region, with the range of variation cycling every four regions through the following:

1) 0 msec (lowest mean and lowest variance)
2) 0–5 msec (medium-low mean, medium variance)
3) 5–10 msec (highest mean, medium variance)
4) 0–10 msec (medium-high mean, highest variance)

Execution time variability within each range was implemented as a pseudo-random sequence initialized with the same seed for each strategy. The system moved to the next operating region every 150 seconds in each trial. The same profile of load and load jitter was therefore applied for each strategy, allowing direct comparisons of trials for different strategies. Table II shows how the HRT execution variability and additional SRT loads were combined in each region:

Region	Variable HRT Execution	SRT Load Chain Length
0	0 msec	1 route leg
1	0 to 5 msec	1 route leg
2	5 to 10 msec	2 route legs
3	0 to 10 msec	3 route legs
4	0 msec	4 route legs
5	0 to 5 msec	5 route legs
6	5 to 10 msec	6 route legs
7	0 to 10 msec	7 route legs
8	0 msec	8 route legs
9	0 to 5 msec	9 route legs
10	5 to 10 msec	10 route legs
11	0 to 10 msec	11 route legs

TABLE II
LOADS FOR EACH OPERATING REGION

- **Regions 0, 4 and 8** have fixed HRT event consumer loads, with no additional variability.
- **Regions 1, 5, and 9** have variability of between 0 msec and 5 msec for each of the 10 Hz, 5 Hz, and 1 Hz rates, for a total variability of between 0 and 80 msec of each 1 Hz frame, *i.e.*, between 0 and 8 percent variability.
- **Regions 2, 6, and 10** have variability of between 5 msec and 10 msec for each of the 10 Hz, 5 Hz, and 1 Hz rates, for a total variability of between 80 and 160 msec of each 1 Hz frame, *i.e.*, between 8 and 16 percent variability.
- **Regions 3, 7, and 11** have variability of between 0 msec and 10 msec for each of the 10 Hz, 5 Hz, and 1 Hz rates, for a total variability of between 0 and 160 msec of each 1 Hz frame, *i.e.*, between 0 and 16 percent variability.

Total variability was thus lowest in regions 0, 4, and 8, higher in regions 1, 5, and 9, higher still in regions 3, 7, and 11, and highest in regions 2, 6, and 10. The *range* of variability was lowest in regions 0, 4, and 8, was comparable in odd-numbered regions, and was highest in regions 2, 6, and 10.

Each of the scheduling strategies examined in these trials was studied both with and without SRT operation cancellation enabled. If cancellation was enabled, an operation's *upcall monitor adapter* would simply omit an upcall to the operation if its advertised worst-case execution time exceeded the time remaining before its deadline at the point of upcall.

The route leg update operation was registered as both an event consumer and event supplier for TAO's Real-time Event Channel. When an event consumer routine is called, it updates one route leg. If there are remaining steps in its computation chain (according to the chain length for the current region, as described in Table II), it pushes a SRT event to be consumed if needed. If a SRT event to the route leg update consumer is cancelled, therefore, additional SRT events are not pushed to the Real-time Event Channel even if the mode indicates that there should be additional updates.

The end point of a route leg is a necessary input to the next route leg (*i.e.*, its starting point). If a route leg missed its deadline, its end point would be produced after the data are flushed to the replication service. Any subsequent route legs computed in that chain would then likely be erroneous. Shedding the route leg load chain at the first missed deadline removes operations that would otherwise consume CPU time without adding utility. The cancellation policy outlined above therefore enables an increase in efficiency in operation dispatching, without a loss of utility for the larger class of chained operations, of which route leg updates are one example.

3) Measured Variables: To measure the effects of varying load and load jitter described in Section III-B.2, we instrumented the application and middleware using lightweight, high-resolution time stamps to profile system behavior. We collected three types of in-

formation:

1) Latency of dispatching enqueue and dequeue actions
2) Missed, made, and cancelled operation deadlines
3) Latency of the operation executions themselves

A key challenge in collecting and using this information is to do so without violating either the space- or time-requirements of the OFP application. In particular, data collection and extraction must be done so that (1) relevant data are collected and not lost, (2) data extraction is sufficient to avoid data collection overflowing available data storage space(s), and (3) neither collection nor extraction of data interferes with the real-time constraints of the system itself. To achieve this, we first optimized the data probes and cache for both efficiency and flexibility. Second, we leveraged the existing phasing of application operations to provide regular windows of reduced contention for the CPU, in which to extract collected data. Figure 5 shows the resulting framing of operations in the executing OFP. This framing is designed to improve real-time behavior as follows: (1) frame periods are harmonic and (2) initiation of requests is staggered to reduce contention, *i.e.*, avoiding the canonical critical instant for as many operations as possible.

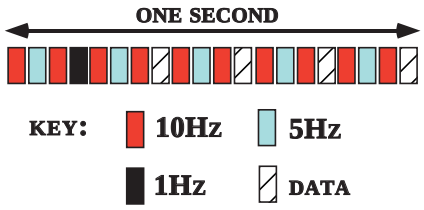

Fig. 5. Framing of Operation Requests and Metrics Data Extraction Points

IV. Empirical Results

We now present our results from running the trials described in Section III-B, using the open experimental platform described in Section II. Specifically, we systematically examine the hypotheses described in Table I and note how a particular OFP challenge described in Section III-A is or is not met in each case. We thus empirically evaluate the suitability of COTS-based hardware and software—in particular our use of TAO, the TAO Real-time Event Channel, and the Kokyu framework, for mission-critical DRE systems.

A. Extending QoS Assurances

Hypothesis – Multi-paradigm scheduling is needed to both (1) maintain QoS assurances for DRE systems while (2) increasing performance beyond levels achievable by single-paradigm approaches: We apply multi-paradigm scheduling to meet challenges A, B, and D described in Section III-A. In particular, in cases where critical requirements are feasible—but total processing requirements are not—we expect that multi-paradigm scheduling will maintain critical assurances where single-paradigm (*i.e.*, static, dynamic, or even hybrid) approaches cannot. Second, we expect multi-paradigm scheduling to provide more effective use of scarce resources than single paradigm approaches, by considering *scheduling* modes as well as application modes. Finally, we expect that multi-paradigm scheduling will *both* meet critical assurances and improve non-critical performance robustly under widely varying environmental conditions.

Overview of the test: To evaluate this hypothesis, we examined the dispatching load and how each strategy performed in meeting critical deadlines as the load increased. In particular, we examined the total number of operation deadlines missed, made, and cancelled for each of the six strategies examined, *i.e.*, RMS, MUF, and RMS+MLF each with and without cancellation of SRT operations.

Summary of test results: Figure 6 shows effective load on the system with each scheduling strategy, *i.e.*, the total number of requests enqueued, in each of the operating regions. Scheduling strategies using operation cancellation are indicated by a © annotation. MUF and RMS+MLF (both with cancellation) enqueued fewer

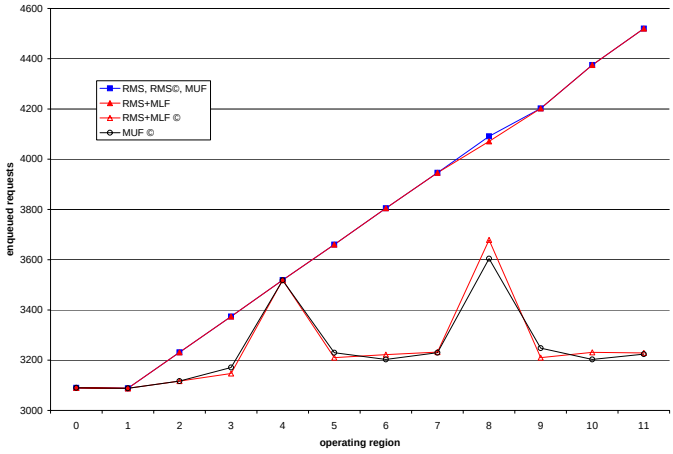

Fig. 6. Total Requests Enqueued

dispatch requests overall due to the effects of cancellation on the chains of operations described in Section III-B.2, *i.e.*, when one operation of a chain is cancelled, subsequent requests for that operation are not made. The other strategies, RMS, MUF, and RMS+MLF (all without cancellation), and RMS with cancellation, enqueued a total number of dispatch requests that rose linearly from around 3,100 in regions 0 and 1 to above 4,500 in region 11.

Figure 7 shows the total number of HRT and SRT operation deadlines made, missed, and cancelled for the MUF strategy *with* cancellation. Figure 8 shows the same results for MUF *without* cancellation. The total operation loads in RMS+MLF were similar to those in MUF, both with and without cancellation respectively. Cancellation in RMS+MLF was similarly successful in reducing the number of operation deadlines missed though again with a lower number of operation deadlines made. As with MUF, RMS+MLF met more deadlines under lower levels of jitter, *i.e.*, in operating regions 0, 4, 8, than under higher levels of jitter, *i.e.*, in operating regions 1–3, 5–7, and 9–11, respectively.

Figure 9 shows the total number of HRT and SRT operation deadlines made, missed, and cancelled for the RMS strategy *without* cancellation. Performance results for RMS *with* cancellation were nearly identical to those in Figure 9, except that RMS with cancellation first missed HRT deadlines in operating region 6, rather than 7. RMS with cancellation failed to cancel even a single non-critical operation dispatch request: both RMS with cancellation and RMS without cancellation showed a total operation load similar to that of MUF without cancellation and RMS+MLF without cancellation. Both RMS with cancellation and RMS without cancellation show a significant number of HRT deadlines missed in the later, more heavily loaded operating regions, and RMS with cancellation both (1) missed more HRT deadlines overall and (2) first missed deadlines in an earlier operating region with lower total load, than RMS without cancellation.

Analysis of test results: In each of the operation behavior graphs above, it is instructive to compare the slope of the top curve, which indicates the increase in the total number of dispatch requests in subsequent operating regions. In Figure 8 the slope of the total requests curve is similar to that shown in Figure 6, though the curve is slightly lower as some dispatch requests are for internal dependency correlations in the event channel, and not for application

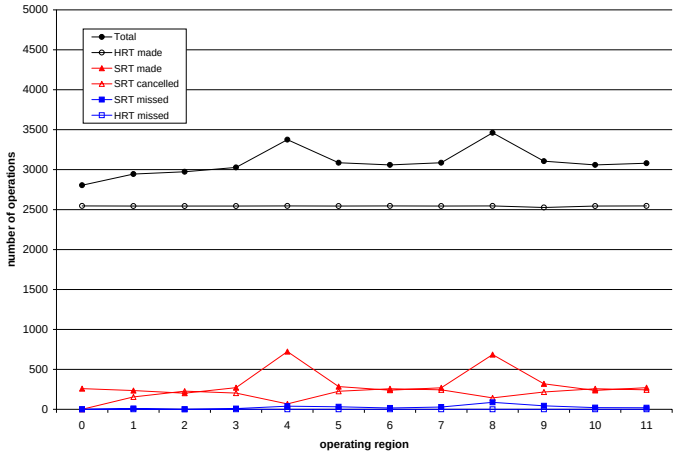

Fig. 7. MUF Operation Behavior With Cancellation

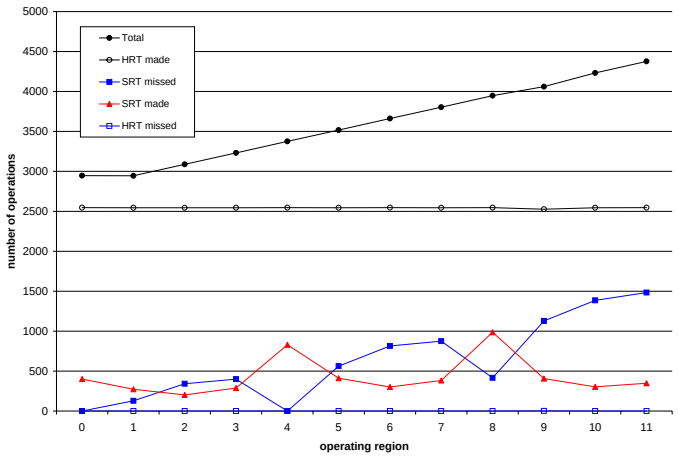

Fig. 8. MUF Operation Behavior Without Cancellation

operations. Without cancellation, the total operation load in MUF was thus proportional to the number of enqueued requests.

In Figure 7, the slope of the total requests curve was much less than in Figure 8, indicating a lower and more slowly increasing total operation load. The total operation load in MUF with cancellation was well bounded, which we attribute to the effects of cancellation on route leg update chains. Cancellation in MUF successfully reduced the number of operation deadlines missed, though it also resulted in a lower number of operation deadlines made. Both with and without cancellation, MUF met more deadlines under lower levels of jitter, *i.e.*, in operating regions 0, 4, 8, than under higher levels of jitter, *i.e.*, in operating regions 1–3, 5–7, and 9–11, respectively.

Interestingly, adding cancellation had no apparent benefit at all with RMS in this application. In fact, it showed a greater number of HRT deadlines missed and a lower number of HRT deadlines made, in regions 6 though 11. We attribute this effect to the priority assignment in RMS, under which 20 Hz SRT requests for operations in the route leg chains were dispatched at the highest priority.

a) Summary: The results above support the hypothesis that multi-paradigm scheduling is needed to extend QoS assurances and performance for DRE systems beyond those achievable by single-paradigm approaches. RMS was only able to meet critical deadlines in operating regions 0 through 6. With two exceptions discussed in Section IV-B, MUF and RMS+MLF were able to meet critical deadlines in all operating regions. However, RMS made more non-critical deadlines in operating regions 0 through 6. We therefore believe multi-paradigm scheduling is both beneficial and empirically supported for use in mission-critical DRE systems.

B. Impact of Infrastructure Factors on Scheduling Feasibility

Hypothesis – Infrastructure factors, such as dynamic queue or cancellation overhead, may influence both the ability to enforce critical processing assurances, and the ability to improve non-critical processing performance: Multi-paradigm scheduling can extend the range of environmental conditions over which assurances can be made and performance improved (as described in Section IV-A). However, we must also examine the effects of infrastructure factors on multi-paradigm scheduling, to meet challenges C and E described in Section III-A. In particular, DRE system developers must during validation and certification *consider* special cases where critical assurances are violated, to ensure isolation of critical and non-critical resource requirements. Furthermore, careful study is needed to *identify* those special cases and ensure non-critical processing is added safely. We therefore must examine queueing and cancellation overhead empirically to further address the challenge of daptability across product families, while also addressing the challenge of safely adding non-critical processing, as described in Section III-A.

Overview of the test: To evaluate this hypothesis we first examined the queueing latency induced by the infrastructure itself. We then compared the ability of strategies incurring differing levels of overhead to meet critical deadlines. As before, we examine the total number of operation deadlines missed, made, and cancelled for each of the scheduling strategies.

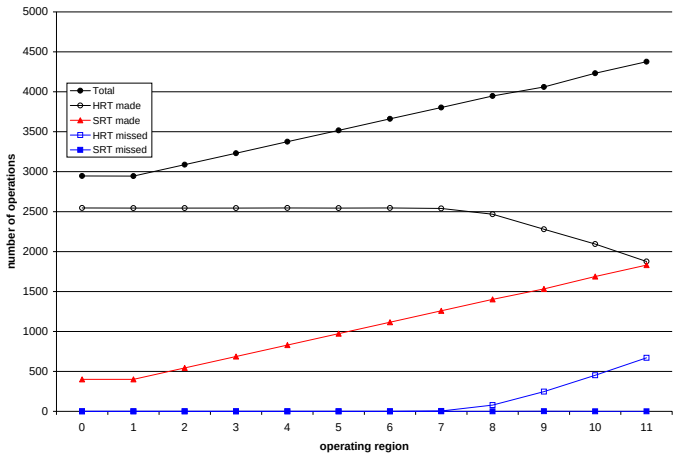

Fig. 9. RMS Operation Behavior Without Cancellation

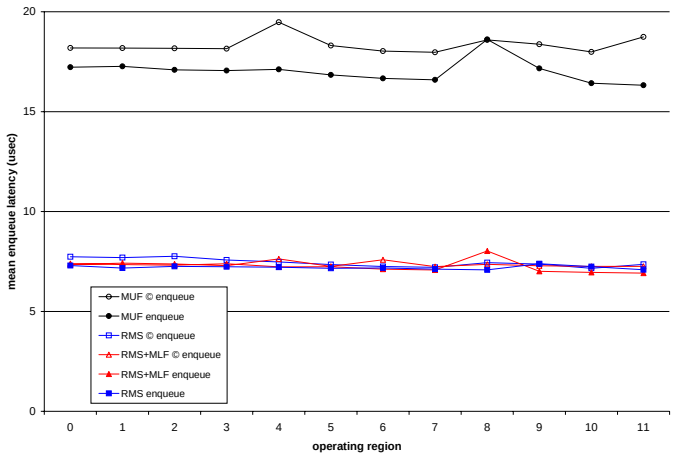

Fig. 10. Mean Enqueue Latency Per Operation

Summary of test results: Figures 10 and 11 show the mean enqueue and dequeue latencies for each strategy in each operating region, respectively. These figures illustrate that enqueue calls showed higher latency than dequeue calls. MUF with and without cancellation had the highest mean enqueue and dequeue latencies, with lower latencies for RMS and RMS+MLF both with and without cancellation.

In light of the differences in overhead between MUF and RMS+MLF, it is instructive to examine closely the HRT deadlines missed in strategies other than RMS beyond the total feasibility limit. In addition to the missed HRT deadlines for RMS with and without cancellation described in Section IV-A, one HRT deadline was missed in region 9 in each of the MUF without cancellation and RMS+MLF with cancellation strategies. Interestingly, this is the only case of a missed HRT deadline outside RMS; it occurred in the same region at the same sampling point for both strategies.

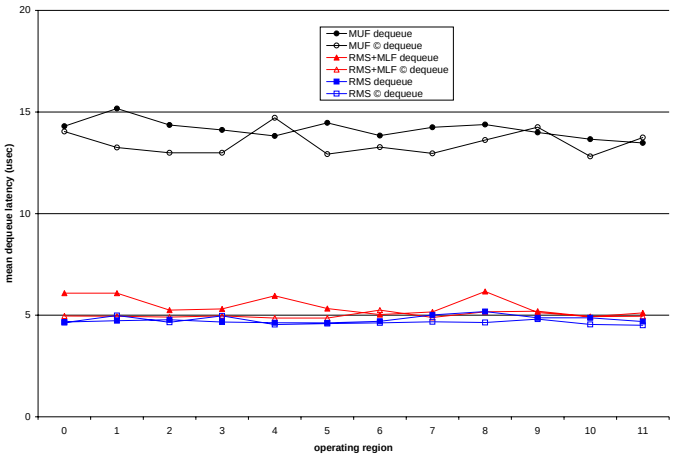

Fig. 11. Mean Dequeue Latency Per Operation

Analysis of test results: The most important feature of the enqueue and dequeue latency plots above is that the mean enqueue and dequeue latencies did not rise significantly with increasing load or variations in jitter. Including preemption and jitter delays, the combined average queueing latency in each strategy (1) took around 12 μsec per dispatch request for RMS and RMS+MLF, (2) took around 32 μsec per dispatch request for MUF, and (3) for each strategy remained comparable across operating regions.

We observed one missed HRT deadline in region 9 in each of the two strategies: MUF without cancellation and RMS+MLF with cancellation. We now examine the possible causes of this phenomenon. As Section III-B.2 describes, the same pseudo-random sequence was used for the load jitter function, and the same basic load function was used across strategies. It is therefore notable that the same operation missed one deadline in the same data sample of the same region in two different strategies. The HRT operation that missed its deadline in both cases was the 10 Hz HRT additional operation used to induce randomized jitter to various operating regions, as described in Section III-B.2.

The range of jitter in this operation for region 9, shown in Table II, is 0 to 5 msec, or 0 to 5 percent of a 100 msec 10 Hz frame. There was no significant difference in latency for that one operation among the strategies in that region, either in the minimum, maximum, or mean, or at the sample point at which the deadline was missed. However, MUF without cancellation and RMS+MLF with cancellation had slightly higher accrued HRT latency overall at sample 140, where the deadline was missed. Moreover, even if preemption by the 40 Hz reactor thread occurred, the deadline had already been missed and the cause must be attributed to other factors. It therefore appears likely the missed deadline resulted from an overall vulnerability of the RMS+MLF strategy with cancellation and the MUF strategy without cancellation at that point, rather than from a single anomaly. In particular, if delays from preemption by spurious VxWorks network task interrupts contributed to this effect, it appears unlikely that a single long preemption interval was involved.

Summary: These results support the hypothesis that infrastructure factors may influence both the ability to enforce critical processing assurances, and the ability to improve non-critical processing performance. In particular, the missed deadlines in MUF without cancellation and RMS+MLF with cancellation correlate with additional overhead of mechanisms for (1) dynamic queue management and (2) operation cancellation, respectively. We therefore believe that while multi-paradigm scheduling is empirically supported for use in mission-critical DRE systems, additional experiments and careful and thorough testing are needed to more fully assess the impacts of these kinds of mechanisms on mission-critical DRE systems.

V. Observations and Recommendations

Sections III and IV focused on the empirical study of canonical scheduling strategies for avionics mission computer OFPs. Mission computing software, like many other next-generation DRE software, is increasingly required to execute in more flexible ways and in increasingly varying environments. Characterizing the actual performance of the Kokyu middleware infrastructure in a realistic setting under a variety of load and load jitter conditions is therefore of fundamental importance. Moreover, new increasingly non-deterministic types of processing, such as video and imaging [2], are being targeted for transition to these DRE systems. The Kokyu framework's ability to manage variations in execution load and load jitter through alternative scheduling strategies increases the applicability of these techniques to DRE systems with next-generation software requirements and architectures.

Our work also opens a larger possibility: performing truly adaptive scheduling using alternative strategies at run-time, to accommodate variations in the systems operating environment and current mission objectives. There are several ongoing areas of research to complete, as Section VII describes, before this type of run-time adaptation will be applicable to avionics mission computing OFPs. Based on the results in this paper, however, these problems appear tractable, and planned future work will lead to a more complete solution.

Below, we present key observations and recommendations based on our empirical results from Section IV. These observations and recommendations apply both to the particular avionics mission computing application studied and to a larger family of mission-critical DRE systems.

A. Extend Assurances via Hybrid Scheduling

Observation – Hybrid static/dynamic scheduling strategies met critical deadlines in operating regions where static strategies could not: The hybrid static/dynamic scheduling strategies MUF and RMS+MLF (both without cancellation) were effective in managing dynamic SRT load, and isolating HRT and SRT resource utilization, across a wider range of total load. Moreover, they did so under different levels and ranges of randomized jitter in the execution times of certain HRT and SRT operations at different rates. These results support the hypothesis that multi-paradigm scheduling is needed and beneficial to extend QoS assurances for DRE systems beyond those achievable by single-paradigm approaches.

Recommendation – Applying hybrid scheduling can be effective for mission-critical DRE applications that experience overload: Criticality-aware hybrid static/dynamic scheduling in middleware should be considered for systems that (1) have both critical and non-critical operations, (2) have critical load that is always feasible, and (3) may incur total load in excess of the feasible bound.

B. Pay Attention to Infrastructure Overhead

Observation – Overhead from cancellation and dynamic scheduling is reasonable, but impacts performance and may impact feasibility: Dynamic queue management is used to a lesser extent by the RMS+MLF variants, and to a greater extent by the MUF variants. The overhead of increased dynamic queue management was noticeable, but was within a reasonable scalar ($\sim$1.5) of the more static queue management overhead. Moreover, this overhead was in large part justified by increases in effectiveness or efficiency or both. Queueing loads appeared to remain relatively stable for each scheduling strategy, as may be expected for such a harmonic periodic application. Developers of rate-based real-time distributed applications should therefore consider dynamic scheduling in middleware to be a reasonable and useful technique.

While in all but one sample MUF and RMS+MLF were able to enforce critical assurances, the same sample late in operating region 9 showed a single missed deadline for MUF without cancellation and RMS+MLF with cancellation. These two strategies had intermediate overhead among the strategies that made all other critical deadlines in region 9. These results support the hypothesis that infrastructure factors, such as dynamic queue overhead, may influence both the ability to enforce critical processing assurances, and the ability to improve non-critical processing performance.

Recommendation – Perform careful empirical evaluation of sources of overhead associated with chosen scheduling strategies, and in particular their impacts on performance and feasibility: The above observations suggest a vulnerability of scheduling strategies that impose overheads such as cancellation or dynamic queue management to missing critical deadlines. This is apparently due to some form of interference between non-critical and critical processing. Additional experiments are needed, however, to isolate the particular mechanisms and effects involved. Moreover, careful empirical testing of specific DRE systems is always recommended.

C. Apply Multiple Scheduling Paradigms

Observation – The dominant scheduling strategy differed across operating regions: In Figure 12 we recolor each of the operating regions originally portrayed in Figure 4 to show the scheduling strategy that performed best in each region. The static RMS

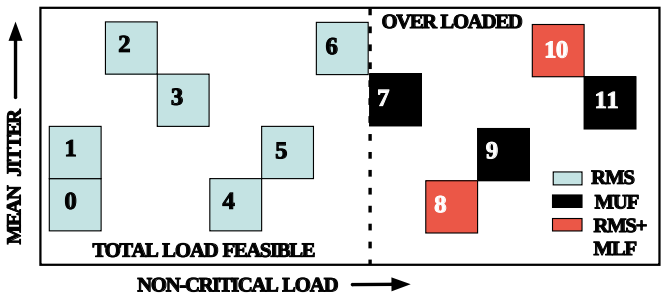

Fig. 12. Most Effective Strategy by Operating Region

strategy without cancellation performed best among the strategies studied when the total load was below the feasible limit. Above that limit the hybrid static/dynamic RMS+MLF or MUF strategies performed best. These results support the hypothesis that the efficiency and effectiveness of any given scheduling strategy are functions of environmental factors, in addition to the effects of the infrastructure overheads discussed in Section IV-B.

Recommendation – Use different scheduling strategies under different load conditions: For the avionics mission computing application studied, we recommend using the following scheduling strategies in the following cases:

- RMS if the system is not subject to overloads,
- RMS+MLF or MUF if the system is subject to overloads but some degradation of non-critical performance is acceptable when the system is not overloaded, or
- Using mode switching at run-time between RMS when the system is not overloaded, and RMS+MLF or MUF when it is.

VI. Related Work

Distributed real-time and embedded (DRE) computing is an emerging field of study. An increasing number of research efforts are focusing on end-to-end quality of service (QoS) properties, such as timeliness, by integrating QoS management policies and mechanisms, *e.g.*, real-time scheduling into standards-based middleware, such as Real-time CORBA. Pioneering efforts are beginning to extend this field by providing meta-capabilities, such as configuration flexibility, reflection, and ultimately adaptation, while still meeting strict QoS assurances. This section describes representative work that is related to our Kokyu framework.

Avionics platform research: The following two branches of research are endeavoring to make QoS-managed system infrastructure a prevalent and reusable feature of avionics computing systems:

• *Avionics domain platform research:* Standardized avionics platforms, such as the ARINC Avionics Application Software Standard Interface (APEX) for Integrated Modular Avionics (IMA) [26], provide QoS assurances for systems in the avionics domain. McElhone [27] examines the question of how to support operations with soft real-time constraints and possibly long running or variable length computations, in canonical avionics-specific platforms, such as IMA.

• *Open systems avionics research:* Sharp, Doerr, *et al.* [21], [22] address the challenge of retaining key QoS assurances in avionics systems, while achieving improvements in modularity, reuse, cycle times, and cost across families of flight software applications. The Bold Stroke avionics domain-specific middleware described in Section II-C has emerged and evolved through that work. Our research on flexible and adaptive real-time scheduling and dispatching was conducted within the context of the Bold Stroke infrastructure, and has contributed to its evolution.

CORBA-related QoS middleware research: There is a growing body of work related to CORBA-based QoS middleware. We focus below on related CORBA middleware research efforts that address scheduling or other forms of adaptive QoS management.

• *Standard specifications:* The OMG Real-Time CORBA 1.0 [28] specification includes interfaces for an optional scheduling service that can be implemented readily using Kokyu's flexible scheduling and dispatching capabilities. We plan to release an implementation of this service built using the Kokyu framework. Emerging COTS middleware standards, such as Dynamic Scheduling Real-Time the Common Object Request Broker Architecture (CORBA) 2.0 (DSRTCORBA) [29], as well as the non-CORBA Real-Time Specification for Java^{TM} (RTSJ) [30], generalize the possible range of scheduler implementations, rather than specifying a particular scheduling approach. Kokyu offers a natural basis for reuse of policies and mechanisms in implementing schedulers and associated dispatching infrastructures for either of these standards.

• *BBN QuO:* The *Quality Objects* (QuO) distributed object middleware is developed at BBN Technologies [31]. QuO is based on CORBA and provides the following support for agile applications running in wide-area networks: (1) *run-time performance tuning and configuration* through the specification of *QoS regions*, behavior alternatives, and reconfiguration strategies that allows the QuO run-time to adaptively trigger reconfiguration as system conditions change (represented by transitions between operating regions) and (2) *feedback* across software and distribution boundaries 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 Kokyu into the QuO framework, as described in [2].

• *UCSB Realize:* The Realize project at UCSB has developed an approach based on object migration and replication, to improve performance of soft real-time distributed systems [32], [33]. This approach constitutes a higher level of adaptive control for soft real-time QoS management, and is complementary to Kokyu. In particular, a system developer might apply Realize to provide soft real-time load balancing across endsystems, using the Kokyu framework to integrate scheduling and dispatching of both critical and non-critical load.

• *UCI TMO:* The Time-triggered Message-triggered Objects (TMO) project [34] at the University of California, Irvine, supports the integrated design of distributed OO systems and real-time simulators of their operating environments. The TMO model provides structured timing semantics for distributed real-time object-oriented applications by extending conventional invocation semantics for object methods, *i.e.*, CORBA operations, to include (1) invocation of time-triggered operations based on system times and (2) invocation and time bounded execution of conventional message-triggered operations. TMO, Kokyu, and TAO are complementary technologies because (1) TMO and Kokyu extend and generalize TAO's existing time-based invocation capabilities and (2) TAO provides a configurable and dependable connection infrastructure needed by the TMO CNCM service.

Non-CORBA QoS research: In addition to CORBA-related QoS middleware research, our work on Kokyu is also related to the following QoS research conducted outside CORBA:

• *Utah CRM:* Regehr and Lepreau [35] propose the CPU Resource Manager (CRM), a middleware service for managing processor allocation using scheduling abstractions provided by COTS operating systems. They examine different kinds of QoS reservations and propose a unifying low-level middleware abstraction layer to shield developers from accidental complexities produced by variations in scheduling abstractions at the operating system level. Our approach focuses on *encapsulation* of scheduling and dispatching policies, and providing flexible infrastructure to allow arbitrary composition of heuristics. Rather than enclosing a known set of common abstractions, our aim is to provide flexible support for diverse and possibly unanticipated combinations of scheduling requirements, mechanisms, and policies in middleware.

• *UCI RED-Linux Scheduling Framework:* Wang, *et al.*, at the University of California, Irvine, have proposed a general scheduling framework [36] to unify three distinct kinds of scheduling approaches: *priority-based*, *time-based*, and *share-based*. They decompose scheduling behavior into policy (*allocator*) and mechanism (*dispatching*) components, which are similar to the Kokyu scheduling service framework. They have implemented the dispatching portion of this framework in their real-time extensions to the Linux kernel, called RED-Linux. While the RED-Linux approach to scheduling relies on special-purpose extensions to the OS kernel, our Kokyu framework relies only on commonly available OS features, such as preemptive thread priorities. Our dispatching mechanisms can therefore augment standards-based CORBA middleware and can perform effectively on a wide range of commonly available real-time and general-purpose OS platforms.

VII. Concluding Remarks

To quantify the tradeoffs between static and dynamic scheduling algorithms, we developed a strategized scheduling service framework called Kokyu and integrated this framework with TAO [8], which is our high-performance, real-time ORB, and the TAO Real-time Event Channel, which is a QoS-enabled publish/subscribe service.[3] Our experimental results demonstrate that no single scheduling paradigm is ideal in all cases, and therefore multi-paradigm scheduling is both suitable and beneficial to mission-critical DRE applications. In particular, multi-paradigm scheduling can provide both *assurances* and increased *performance* to DRE applications with both critical and non-critical operations.

This paper describes how we used the TAO ORB, TAO's Real-time Event Channel, and Kokyu to empirically measure the overhead, effectiveness, and efficiency of different scheduling strategies in a production-quality DRE application: an operational flight program for avionics mission computing built atop the Boeing Bold Stroke domain-specific middleware. Our empirical measurements provide a foundation upon which we are developing practical guidelines to configure and use multi-paradigm scheduling strategies for Real-time CORBA applications. We conclude by summarizing our lessons learned in this work and outlining our planned areas of future work.

Summary of lessons learned: The following are key lessons learned from our application of COTS hardware and software technologies to avionics missions computing:

• *Multi-paradigm scheduling is necessary and beneficial.:* While standards, such as the Real-time CORBA 1.0 and 2.0 specifications, address key issues for mission-critical DRE systems, they leave essential areas unspecified, notably: (1) which scheduling strategies are suitable to a particular DRE system and (2) which will outperform the others under each set of environmental conditions within which the system runs. Our empirical results demonstrate the limitations of any *single-paradigm* approach, and show that RMS is preferable when total load is feasible, whereas strategies that can isolate critical and non-critical processing are preferable in overload situations. Our results also indicate that hybrid static/dynamic scheduling strategies can be used in Real-time CORBA applications to (1) offer higher resource utilization than purely static scheduling strategies with acceptable run-time cost, (2) preserve scheduling assurances for critical operations even for an overloaded schedule, and (3) provide applications the flexibility to adapt to varying application requirements and platform features.

• *Careful instrumentation and analysis to measure infrastructure overhead and its impact is necessary:* While hybrid static/dynamic scheduling mechanisms added some overhead, our results show that (1) the overhead is within reasonable bounds for DRE applications, and (2) offered suitable performance across different levels of load and load jitter. The case of a missed critical deadline reported in Section IV-B urges caution, however, as well as careful empirical evaluation when applying these techniques to mission-critical DRE systems. Our results show that while operation cancellation did not improve *effectiveness* of scheduling strategies, it did improve *efficiency* when moderate or high levels of jitter were present.

Future work: We are currently exploring the following areas in our future research on multi-paradigm scheduling of Real-time CORBA operations:

[3]TAO, TAO's Real-time Event Channel, and Kokyu are available as open-source software from `www.cs.wustl.edu/~schmidt/TAO.html`.

1) **Performance models**—We are investigating models for the results seen in this work, particularly whether the better performance of MUF under moderate jitter is due to (1) incidental slack-stealing effects allowed by the greater overhead of dynamic scheduling or (2) a particular capability of the scheduling mechanism itself.
2) **Distributed scheduling behavior**—Further empirical measurements are needed to determine the impact of factors such as network latency on the end-to-end performance of dynamically scheduled distributed systems.
3) **Application requirements**—A detailed examination of the impact of application specific requirements, such as policies for handling missed deadlines, will help guide the development of additional strategies for dynamically scheduled systems.
4) **Adaptive control**—We are exploring whether adaptive control laws for alternation between scheduling strategies can be identified and demonstrated to be effective for broad classes of DRE systems.

ACKNOWLEDGMENTS

We gratefully acknowledge the support and direction of the AFRL program manager for ASFD, Kenneth Littlejohn, and of Boeing Bold Stroke Principal Investigators Bryan Doerr and David Sharp. In addition, we would like to thank Greg Holtmeyer for his contributions to this research, Douglas Niehaus for his suggestions on improving this paper, and Fred Kuhns for his observation that the better performance by MUF under moderate jitter conditions could be due to a form of slack stealing by non-critical operations.

REFERENCES

[1] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, vol. 20, March 2001.

[2] J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, and D. Karr, "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pp. 625–634, IEEE, April 2001.

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

[4] D. B. Stewart and P. K. Khosla, "Real-Time Scheduling of Sensor-Based Control Systems," in *Real-Time Programming* (W. Halang and K. Ramamritham, eds.), Tarrytown, NY: Pergamon Press, 1992.

[5] R. E. Schantz and D. C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering* (J. Marciniak and G. Telecki, eds.), New York: Wiley & Sons, 2001.

[6] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Reading, Massachusetts: Addison-Wesley, 1999.

[7] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6 ed., Dec. 2001.

[8] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[9] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), pp. 184–199, ACM, October 1997.

[10] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *Transactions on Computing*, vol. 47, no. 4, 1998.

[11] F. Kuhns, D. C. Schmidt, C. O'Ryan, and D. Levine, "Supporting High-performance I/O in QoS-enabled ORB Middleware," *Cluster Computing: the Journal on Networks, Software, and Applications*, vol. 3, no. 3, 2000.

[12] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[13] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, vol. 21, no. 2, 2001.

[14] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[15] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. L. Levine, "Evaluating Policies and Mechanisms to Support Distributed Real-Time Applications with CORBA," *Concurrency and Computing: Practice and Experience*, vol. 13, no. 2, pp. 507–541, 2001.

[16] O. Othman, C. O'Ryan, and D. C. Schmidt, "An Efficient Adaptive Load Balancing Service for CORBA," *IEEE Distributed Systems Online*, vol. 2, Mar. 2001.

[17] N. Wang, D. C. Schmidt, O. Othman, and K. Parameswaran, "Evaluating Meta-Programming Mechanisms for ORB Middleware," *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies*, vol. 39, Oct. 2001.

[18] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

[19] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46–61, January 1973.

[20] J.-Y. Chung, J. W.-S. Liu, and K.-J. Lin, "Scheduling Periodic Jobs that Allow Imprecise Results," *IEEE Transactions on Computers*, vol. 39, pp. 1156–1174, September 1990.

[21] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

[22] B. S. Doerr and D. C. Sharp, "Freeing Product Line Architectures from Execution Dependencies," in *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[23] C. D. Locke, "Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives," *The Journal of Real-Time Systems*, vol. 4, pp. 37–53, 1992.

[24] Wind River Systems, "VxWorks 5.3." www.wrs.com/products/html/vxworks.html.

[25] C. D. Gill, J. W. Hoffert, D. C. Sharp, and P. H. Goertzen, "An Evolution of QoS Context Propagation in Event-Mediated Avionics Software Architectures," in *Proceedings of the 20th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Oct. 2001.

[26] ARINC Incorporated, Annapolis, Maryland, USA, *Document No. 653: Avionics Application Software Standard Inteface (Draft 15)*, Jan. 1997.

[27] C. McElhone, "Soft Computations within Integrated Avionics Systems," in *Proceedings of the IEEE National Aerospace and Electronics Conference (NAECON 2000)*, October 2000.

[28] Object Management Group, *Real-time CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

[29] Object Management Group, *Dynamic Scheduling Real-Time CORBA 2.0 Joint Final Submission*, OMG Document orbos/2001-06-09 ed., Apr. 2001.

[30] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[31] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

[32] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser, "Dynamic Migration Algorithms for Distributed Object Systems," in *21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, (Phoenix AZ), IEEE, Apr. 2001.

[33] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser, "Dynamic Scheduling of Distributed Method Invocations," in *21st IEEE Real-Time Systems Symposium*, (Orlando, FL), IEEE, November 2000.

[34] K. H. K. Kim, "Object Structures for Real-Time Systems and Simulators," *IEEE Computer*, pp. 62–70, Aug. 1997.

[35] J. Regehr and J. Lepreau, "The Case for Using Middleware to Manage Diverse Soft Real-Time Schedulers," in *Proceedings of the International Workshop on Multimedia Middleware (M3W '01)*, (Ottowa, Canada), October 2001.

[36] Y.-C. Wang and K.-J. Lin, "Implementing A General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel," in *IEEE Real-Time Systems Symposium*, pp. 246–255, IEEE, December 1999.