Report Number: WUCSE-2002-21

2002-05-01

Flexible Scheduling in Middleware for Distributed rate-based real-time applications - Doctoral Dissertation, May 2002

Christopher D. Gill

Distributed rate-based real-time systems, such as process control and avionics mission computing systems, have traditionally been scheduled statically. Static scheduling provides assurance of schedulability prior to run-time overhead. However, static scheduling is brittle in the face of unanticipated overload, and treats invocation-to-invocation variations in resource requirements inflexibly. As a consequence, processing resources are often under-utilized in the average case, and the resulting systems are hard to adapt to meet new real-time processing requirements. Dynamic scheduling offers relief from the limitations of static scheduling. However, dynamic scheduling offers relief from the limitations of static scheduling. However, dynamic scheduling often has... **Read complete abstract on page 2.**

Flexible Scheduling in Middleware for Distributed rate-based real-time applications - Doctoral Dissertation, May 2002

Christopher D. Gill

Complete Abstract:

Distributed rate-based real-time systems, such as process control and avionics mission computing systems, have traditionally been scheduled statically. Static scheduling provides assurance of schedulability prior to run-time overhead. However, static scheduling is brittle in the face of unanticipated overload, and treats invocation-to-invocation variations in resource requirements inflexibly. As a consequence, processing resources are often under-utilized in the average case, and the resulting systems are hard to adapt to meet new real-time processing requirements. Dynamic scheduling offers relief from the limitations of static scheduling. However, dynamic scheduling offers relief from the limitations of static scheduling. However, dynamic scheduling often has a high run-time cost because certain decisions are enforced on-line. Furthermore, under conditions of overload tasks can be scheduled dynamically that may never be dispatched, or that upon dispatch would miss their deadlines. We review the implications of these factors on rate-based distributed systems, and posits the necessity to combine static and dynamic approaches to exploit the strengths and compensate for the weakness of either approach in isolation. We present a general hybrid approach to real-time scheduling and dispatching in middleware, that can employ both static and dynamic components. This approach provides (1) feasibility assurance for the most critical tasks, (2) the ability to extend this assurance incrementally to operations in successively lower criticality equivalence classes, (3) the ability to trade off bounds on feasible utilization and dispatching over-head in cases where, for example, execution jitter is a factor or rates are not harmonically related, and (4) overall flexibility to make more optimal use of scarce computing resources and to enforce a wider range of application-specified execution requirements. This approach also meets additional constraints of an increasingly important class of rate-based systems, those with requirements for robust management of real-time performance in the face of rapidly and widely changing operating conditions. To support these requirements, we present a middleware framework that implements the hybrid scheduling and dispatching approach described above, and also provides support for (1) adaptive re-scheduling of operations at run-time and (2) reflective alternation among several scheduling strategies to improve real-time performance in the face of changing operating conditions. Adaptive re-scheduling must be performed whenever operating conditions exceed the ability of the scheduling and dispatching infrastructure to meet the critical real-time requirements of the system under the currently specified rates and execution times of operations. Adaptive re-scheduling relies on the ability to change the rates of execution of at least some operations, and may occur under the control of a higher-level middleware resource manager. Different rates of execution may be specified under different operating conditions, and the number of such possible combinations may be arbitrarily large. Furthermore, adaptive rescheduling may in turn require notification of rate-sensitive application components. It is therefore desirable to handle variations in operating conditions entirely within the scheduling and dispatching infrastructure when possible. A rate-based distributed real-time application, or a higher-level resource manager, could thus fall back on adaptive re-scheduling only when it cannot achieve acceptable real-time performance through self-adaptation. Reflective alternation among scheduling heuristics offers a way to tune real-time performance internally, and we offer foundational support for this approach. In particular, run-time observable information such as that provided by our metrics-feedback framework makes it possible to detect that a given current scheduling heuristic is

underperforming the level of service another could provide. Furthermore we present empirical results for our framework in a realistic avionics mission computing environment. This forms the basis for guided adaption. This dissertation makes five contributions in support of flexible and adaptive scheduling and dispatching in middleware. First, we provide a middle scheduling framework that supports arbitrary and fine-grained composition of static/dynamic scheduling, to assure critical timeliness constraints while improving noncritical performance under a range of conditions. Second, we provide a flexible dispatching infrastructure framework composed of fine-grained primitives, and describe how appropriate configurations can be generated automatically based on the output of the scheduling framework. Third, we describe algorithms to reduce the overhead and duration of adaptive rescheduling, based on sorting for rate selection and priority assignment. Fourth, we provide timely and efficient performance information through an optimized metrics-feedback framework, to support higher-level reflection and adaptation decisions. Fifth, we present the results of empirical studies to quantify and evaluate the performance of alternative canonical scheduling heuristics, across a range of load and load jitter conditions. These studies were conducted within an avionics mission computing applications framework running on realistic middleware and embedded hardware. The results obtained from these studies (1) demonstrate the potential benefits of reflective alternation among distinct scheduling heuristics at run-time, and (2) suggest performance factors of interest for future work on adaptive control policies and mechanisms using this framework.

Department of Computer Science & Engineering

2002-21

FLEXIBLE SCHEDULING IN MIDDLEWARE FOR DISTRIBUTED RATE-BASED REAL-TIME APPLICATIONS - Doctoral Dissertation, May 2002

Authors: Christopher D. Gill

May 1, 2002

Abstract: Distributed rate-based real-time systems, such as process control and avionics mission computing systems, have traditionally been scheduled statically. Static scheduling provides assurance of schedulability prior to run-time and can be implemented with low run-time overhead. However, static scheduling is brittle in the face of unanticipated overload, and treats invocation-to-invocation variations in resource requirements inflexibly. As a consequence, processing resources are often under-utilized in the average case, and the resulting systems are hard to adapt to meet new real-time processing requirements.

Dynamic scheduling offers relief from the limitations of static scheduling. However, dynamic scheduling often has a higher run-time cost because certain decisions are enforced on-line. Furthermore, under conditions of overload tasks can be scheduled dynamically that may never be dispatched, or that upon dispatch would miss their deadlines. We review the implications of these factors on rate-based distributed systems, and posits the necessity to combine static and dynamic approaches to exploit the strengths and compensate for the weaknesses of either approach in isolation

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

FLEXIBLE SCHEDULING IN MIDDLEWARE FOR DISTRIBUTED RATE-BASED REAL-TIME APPLICATIONS

by

Christopher D. Gill

Prepared under the direction of Dr. Ron K. Cytron and Dr. Douglas C. Schmidt

---

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

May, 2002

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

FLEXIBLE SCHEDULING IN MIDDLEWARE
FOR DISTRIBUTED RATE-BASED REAL-TIME APPLICATIONS

by Christopher D. Gill

ADVISOR: Dr. Ron K. Cytron and Dr. Douglas C. Schmidt

May, 2002
Saint Louis, Missouri

Distributed rate-based real-time systems, such as process control and avionics mission computing systems, have traditionally been scheduled statically. Static scheduling provides assurance of schedulability prior to run-time and can be implemented with low run-time overhead. However, static scheduling is brittle in the face of unanticipated overload, and treats invocation-to-invocation variations in resource requirements inflexibly. As a consequence, processing resources are often under-utilized in the average case, and the resulting systems are hard to adapt to meet new real-time processing requirements.

Dynamic scheduling offers relief from the limitations of static scheduling. However, dynamic scheduling often has a higher run-time cost because certain decisions are

enforced on-line. Furthermore, under conditions of overload tasks can be scheduled dynamically that may never be dispatched, or that upon dispatch would miss their deadlines. We review the implications of these factors on rate-based distributed systems, and posits the necessity to combine static and dynamic approaches to exploit the strengths and compensate for the weaknesses of either approach in isolation.

We present a general hybrid approach to real-time scheduling and dispatching in middleware, that can employ both static and dynamic components. This approach provides (1) feasibility assurance for the most critical tasks, (2) the ability to extend this assurance incrementally to operations in successively lower criticality equivalence classes, (3) the ability to trade off bounds on feasible utilization and dispatching overhead in cases where, for example, execution jitter is a factor or rates are not harmonically related, and (4) overall flexibility to make more optimal use of scarce computing resources and to enforce a wider range of application-specified execution requirements.

This approach also meets additional constraints of an increasingly important class of rate-based systems, those with requirements for robust management of real-time performance in the face of rapidly and widely changing operating conditions. To support these requirements, we present a middleware framework that implements the hybrid scheduling and dispatching approach described above, and also provides support for (1) adaptive re-scheduling of operations at run-time and (2) reflective alternation among several scheduling strategies to improve real-time performance in the face of changing operating conditions.

Adaptive re-scheduling *must* be performed whenever operating conditions exceed the ability of the scheduling and dispatching infrastructure to meet the *critical* real-time requirements of the system under the currently specified rates and execution times of operations. Adaptive re-scheduling relies on the ability to change the rates of execution of at least some operations, and may occur under the control of a higher-level middleware resource manager. Different rates of execution may be specified under different operating conditions, and the number of such possible combinations may be

arbitrarily large. Furthermore, adaptive re-scheduling must occur within an acceptably predictable and narrow interval, while preserving schedulability assurances for critical operations, and optimizing other properties such as resource utilization to the extent possible. To address these constraints, we describe extensions to provide flexible and efficient strategies for rate selection and priority re-assignment.

Unfortunately, adaptive re-scheduling may in turn require notification of rate-sensitive application components. It is therefore desirable to handle variations in operating conditions entirely within the scheduling and dispatching infrastructure when possible. A rate-based distributed real-time application, or a higher-level resource manager, could thus fall back on adaptive re-scheduling only when it cannot achieve acceptable real-time performance through self-adaptation.

Reflective alternation among scheduling heuristics offers a way to tune real-time performance internally, and we offer foundational support for this approach. In particular, run-time observable information such as that provided by our metrics-feedback framework makes it possible to *detect* that a given current scheduling heuristic is under-performing the level of service another could provide. Furthermore we present empirical results for our framework in a realistic avionics mission computing environment. This forms the basis for *guided* adaptation.

This dissertation makes five contributions in support of flexible and adaptive scheduling and dispatching in middleware. First, we provide a middleware scheduling framework that supports arbitrary and fine-grained composition of static/dynamic scheduling heuristics, to assure critical timeliness constraints while improving non-critical performance under a range of conditions. Second, we provide a flexible dispatching infrastructure framework composed of fine-grained primitives, and describe how appropriate configurations can be generated automatically based on the output of the scheduling framework. Third, we describe algorithms to reduce the overhead and duration of adaptive rescheduling, based on sorting for rate selection and priority assignment. Fourth, we provide timely and efficient performance information through an

optimized metrics-feedback framework, to support higher-level reflection and adaptation decisions. Fifth, we present the results of empirical studies to quantify and evaluate the performance of alternative canonical scheduling heuristics, across a range of load and load jitter conditions. These studies were conducted within an avionics mission computing application framework running on realistic middleware and embedded hardware. The results obtained from these studies (1) demonstrate the potential benefits of reflective alternation among distinct scheduling heuristics at run-time, and (2) suggest performance factors of interest for future work on adaptive *control* policies and mechanisms using this framework.

To the honored memory of our recently departed loved ones: my grandparents Lawrence Gill, Dorothy Godfrey, and Paul Godfrey, and my aunt, Martha Schaeffer – the tapestries of our lives are immeasurably richer for having been interwoven with yours.

Contents

List of Tables

List of Figures

Acknowledgments

The past four years in the Center for Distributed Object Computing in the Department of Computer Science at Washington University has been an unprecedented period of growth for me, both professionally and personally. I owe an inestimable debt to a number of people, and gratefully wish to acknowledge their gifts of time, support, encouragement, ideas, attention, and patient tutelage. If I have omitted anyone, it is entirely unintentional, and solely a function of the number of people to whom I am indebted.

I wish first to thank my three mentors, who have each supported and helped shape the arc of my studies, teaching, and research. Dr. Douglas C. Schmidt made it all possible by giving me the opportunity to join the DOC group. He has been a tireless advisor and a vigorous advocate and collaborator throughout, even while founding a new branch of the DOC group at the University of California, Irvine, serving as DARPA ITO Deputy Director, and writing three books. Dr. David L. Levine introduced me to the challenging and fascinating world of real-time systems. Through example, instruction, and encouragement, he fostered my exploration of and interest in that world. Dr. Ron K. Cytron has been my advisor through the challenging completion and delivery phases of my dissertation, and chaired my final defense committee. Through his generous support, sage and persistent guidance, and good challenging questions, he has also played a major role in my metamorphosis from student and staff researcher to contributing member of the faculty.

I wish to thank the faculty and staff of the Department of Computer Science at Washington University for creating and nurturing a highly productive and supportive environment for both research and education. I have benefitted greatly through the rich opportunities for study, research collaboration, and teaching afforded me during my graduate program. I am additionally grateful for the opportunity to join this faculty, and I look forward to making a strong contribution in kind to the department, the school, and the larger community.

I am grateful for the time and effort spent reviewing, commenting on, and making suggestions for improvement to this dissertation by my final defense committee: Dr. Ron K. Cytron, Dr. Douglas C. Schmidt, Mr. David C. Sharp, Dr. Mark Franklin, Dr.

Roger Chamberlain, Dr. Jason Fritts, Dr. Bill Smart, and Mr. Fred Kuhns. I am also grateful to Dr. Douglas Niehaus who served as an external reader, offering numerous suggestions that helped greatly in refining this dissertation.

I am endebted to the past and present members of the DOC group, whose collaboration and friendship has made it a joy to work and study here. The attitudes of shared effort, knowledge and credit, and of hard work *and* play contribute to the code of excellence known as "The Way of DOC" (www.cs.wustl.edu/~schmidt/ACE_wrappers/etc/DOC-way.html) that cannot help but change the lives of all who pass through the DOC group's now many doors.

I truly appreciate the sustained support, collaboration, and encouragement I have enjoyed from The Boeing Company. The members of the Bold Stroke organization within Boeing Phantom Works in St. Louis, MO, have provided challenging problems, collegial inclusion in their broader discussions, abundant funding, and vigorous interaction within a series of programs that have resulted in several successful demos, many publications, a significant body of software, and the maturation of the ideas contained in this dissertation. Special thanks to Dr. David Corman, Mr. Steve Dorris, Mr. Bryan Doerr, Mr. Pat Goertzen, Ms. Jeanna Gossett, Mr. Greg Holtmeyer, Mr. Brian Mendel, Mr. Brian Pawlak, Mr. Nathan Scandella, Mr. David Sharp, Dr. Doug Stuart, Mr. Jim Urness, Mr. Tom Venturella, Mr. Don Winter, Mr. Russ Wolter, and Ms. Amy Wright, with whom I've had the privilege of working directly.

I owe a similar debt of gratitude to colleagues at several other companies, with whose collaboration this work has evolved. Dr. Rick Schantz and Dr. Joseph Loyall of BBN Technologies have generously shared insights, effort, and support as we have explored new areas of adaptive resource management in distributed systems. I have learned much in the process of building, integrating, measuring, and refining our combined research infrastructure, and hope that these collaborations will continue for many years to come. Dr. Ebrahim Moshiri, Mr. Malcolm Spence, and Mr. Kevin Stanley of Object Computing, Inc. have enthusiastically and energetically supported and encouraged my own growth and development, that of the DOC group, and that of the larger open-source middleware community. Dr. Rakesh Jha, Mr. Nigel Birch, and Mr. John Shackleton at Honeywell collaborated on the initial integration and subsequent evolution of the scheduler and RTARM in the ASTD and WSOA research programs.

I also wish to acknowledge support for several segments of this work by the Air Force Research Labs, Wright Patterson AFB, and by DARPA ITO. I am especially grateful to Mr. Kenneth Littlejohn, AFRL program manager, and Dr. Gary Koob, DARPA ITO program manager, for their interest in this work, and their guidance over a series of research projects.

Although I have been less visible in the dojo than I would have liked these past four years, I have also learned more deeply to appreciate and draw upon the training I have received from my Sensei, Mr. Mark Rubbert, his teacher, Koichi Kashiwaya Sensei, and the many other instructors and students of the St. Louis Ki Society, the Midland Ki Federation, and the Ki no Kenkyukai founded by Master Koichi Tohei. George Simcox Sensei, whose recent death is a reminder how precious the time we have together truly is, described Ki training as "attitudes leading to actions." It is those attitudes and the practice of living at peace within them that have allowed me over the past four years, in the words of Tohei Sensei, to "find spare time even when busy."

Finally, and most of all, I wish to express my unending gratitude to my family. My wife, Barb, and son, Paul, are the lights of my life and my greatest sources of happiness. My parents, David and Helen Gill, and my sister, Sarah Gill, have over the years provided boundless love and support, and that nurture has enabled what successes I've attained. I am doubly blessed by the loving family into which I married: my parents-by-marriage Bill and Lilian Eck and my siblings-by-marriage, Cathy and Kurt Bartley, Lori and Pat Maher, Bill and Margaret Eck, and Mary Jo and Steve Thaxton.

Christopher D. Gill

Washington University in Saint Louis
May 2002

Chapter 1

Introduction

1.1 Motivation

Supporting the quality of service (QoS) demands of next-generation real-time applications requires object-oriented (OO) middleware that is flexible, efficient, predictable, and convenient to program. Applications with both critical and non-critical real-time requirements, such as process control and avionics mission computing systems [67], impose severe constraints on the design and implementation of real-time OO middleware. For example, avionics mission computing applications typically manage sensors and displays, navigate the aircraft's course, and mediate interactions with other aircraft and ground-based stations. Middleware for such applications must support predictable real-time QoS requirements for both critical and non-critical tasks.

Although many of the properties of these applications, such as the allowed rates of execution, execution times, and criticality levels can be specified *a priori*, it is difficult in practice to achieve at once:

1. a complete static specification of all modes of operation,
2. flexibility to evolve and re-use the system across product lifecycles and families, and
3. efficiency in the use of resources at run-time.

Although historically many successful systems have been built and optimized using static scheduling, these approaches have proven labor intensive, very costly to maintain and extend, and difficult to adapt to modern business requirements for reduced production and certification cycle times.

We distinguish the *functional* behavior of the system (*e.g.*, the interfaces to and semantics of computation and data), from the *extra-functional* behavior (*e.g.*, timeliness and predictability). In this dissertation we focus on optimizing the extra-functional *performance* of the system by maintaining static timeliness assurances while increasing overall utilization of processing resources.

In particular, to address the number and diversity of:

1. operating environments and conditions,
2. the need to encapsulate certification requirements for key portions of these systems, and
3. the requirement for flexible integration of new functional and extra-functional requirements,

we have generalized the above static approach. We specify key properties of the system *a priori* where possible, but leave others for run-time specification and then focus our attention on the behaviors of:

1. run-time specification policies and mechanisms,
2. infrastructure configured to enforce the specifications, and
3. the application itself, running on the configured infrastructure.

The remainder of this chapter is structured as follows: Section 1.2 examines the design and implementation challenges for a representative real-time application; Section 1.3 describes the middleware context in which this work has been conducted; Section 1.4 outlines the principal contributions of this research; Finally, section 1.5 describes the overall structure of this dissertation.

1.2 Design and Implementation Challenges

Figure 1.1 illustrates the architecture of a representative real-time application – an OO avionics mission computing platform [41] – developed and deployed using OO middleware components and services based on Common Object Request Broker Architecture (CORBA) [90]. Sensors external to the mission computer generate key data about the status of the aircraft and its environment, such as navigation heading, geographic position, and nearby terrain features. These data are then replicated across multiple

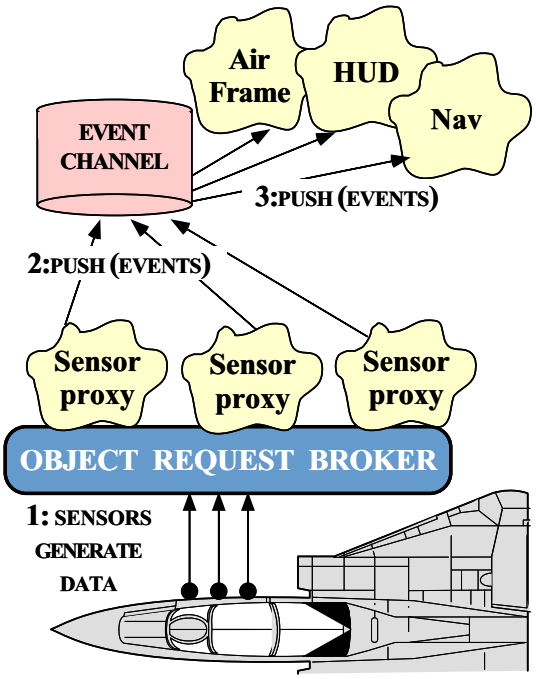

Figure 1.1: Example Avionics Mission Computing Application

mission computing processors using CORBA-compliant Object Request Broker (ORB) middleware. Notification of data availability and subsequent processing of the data are mediated by an event-push architecture in which

1. sensor proxies push events to an event channel, which
2. in turn pushes the events out to event consumers that have registered interest in those events.

Data processing is performed by application components implemented as event consumers. In response to received events, each event consumer may perform one or more of the following operations:

1. process data
2. produce or refine data
3. send events to other consumers

Notably, each of these operations is designed to complete the processing of each event within a well-bounded interval, so that worst-case assurances can be made about the time of completion of processing, both for each operation and in aggregate.

CORBA-compliant ORBs allow clients to invoke operations on a target object without concern for where the object resides, in what language the object's implementation is written, the OS/hardware platform, or the types of communication protocols,

networks, and buses used to interconnect distributed objects [132]. However, achieving these benefits for real-time applications requires the resolution of the following design and implementation challenges:

Scheduling assurance prior to run-time: In real-time applications the consequences of missing a critical deadline can be catastrophic. For example, failure to process an input from the pilot by a specified deadline can be disastrous in an avionics application, especially in mission critical situations. Therefore, it is essential to validate *prior to run-time* that all critical processing deadlines will be met.

Historically, validating stringent timing requirements has implied the use of static, off-line scheduling. For instance, the ARINC Avionics Application Software Standard Interface (APEX) for Integrated Modular Avionics (IMA) relies on two-level scheduling [7, 2]. One level consists of *partitions*, which are executed cyclically and scheduled statically, off-line. The second level consists of application *processes* within each partition, which are scheduled via a more flexible approach using priority-based preemption [7].

Severe resource limitations: Many real-time applications must minimize processing due to strict resource constraints, such as cost, weight, and power consumption restrictions. However, to provide *a priori* assurances for the critical operations, we must ensure worst-case processing requirements can be met.

Therefore, resource allocation and scheduling must always accommodate the worst case, even if non-worst case scenarios are common. For example, an application that relies on real-time image processing [104] must:

1. determine the worst-case processing time, and
2. ensure that sufficient resources are available for that case,

although the usual processing time could be much less than in the worst case.

Distributed processing: Clients running on one processor must be able to invoke operations on servants on other processors. Likewise, the allocation of operations to processors should be flexible. For instance, it should be transparent to the application design and implementation whether an operation resides on the same processor as the client that invokes it.

Testability: Real-time software is complex, critical, and long-lived. Therefore, maintenance is often problematic and expensive [87]. A large percentage of software maintenance involves testing. Current scheduling approaches are validated by extensive and tedious testing, and complete coverage may be difficult to ensure [128]. Therefore, analytical assurance is essential to help reduce validation and verification costs by focusing the requisite testing on the most strategic system components.

Adaptability across product families: Some real-time systems are custom-built for specific product families. Development and testing costs can be reduced if large, common components can be factored out. In addition, validation and certification of components can be shared across product families, amortizing development time and effort.

1.3 Applying CORBA to Predictable Real-Time Applications

To address the design and implementation challenges for the example application described in Section 1.2 and shown in Figure 1.1, standards-based commodity-off-the-shelf (COTS) middleware (*i.e.*, CORBA), was selected as the basis for a common open systems infrastructure across application product lines [24]. Early experience using CORBA on telecommunication [116] and medical imaging projects [104] illustrated that it is well-suited for conventional request/response applications with "best-effort" QoS requirements. Moreover, CORBA addresses issues of distributed processing and adaptation across product families by promoting the separation of interfaces from implementations and supporting component reuse [132].

However, conventional CORBA ORBs were not yet suited for demanding distributed rate-based real-time applications because they do not provide features or optimizations to schedule operations that require predictable real-time QoS [91]. To meet these requirements, the Center for Distributed Object Computing, under the direction of Dr. Douglas C. Schmidt, used and extended the ADAPTIVE Communication Environment (ACE) framework to develop a real-time CORBA ORB called The ACE ORB (TAO) [117]. TAO is an open source implementation of standard CORBA whose ORB and services support efficient and predictable real-time, distributed object computing.

The original work on TAO explored many dimensions of high-performance and real-time ORB design and performance, including event processing [41], request demultiplexing [105], I/O subsystem integration [59], concurrency and connection architectures [118], and IDL compiler stub/skeleton optimizations [35]. Taken together, these advances constitute a foundation for predictable distributed real-time behavior, upon which higher-level capabilities can be built.

Newer standards such as the Real-Time CORBA 1.0 (RTCORBA) specification [91], and the CORBA Messaging specification [89] upon which it depends, describe many of the necessary services, features, and interfaces for higher-level distributed real-time capabilities, such as end-to-end priority preservation and reliable asynchronous method invocation in CORBA middleware. Although these specifications play an important role in the evolution of standards-based COTS middleware, significant additional research [106, 4, 97] has been necessary to identify and address fundamental design and implementation issues not covered in sufficient detail by these standards.

Similarly, the Dynamic Scheduling Real-Time CORBA 2.0 (DSRTCORBA) Joint Final Submission [94] defines a framework for additional capabilities, such as *dynamic* end-to-end management of priorities, but leaves unspecified key areas such as (1) the appropriate heuristics for assigning priorities dynamically, (2) strategies to coordinate both static and dynamic priority management end-to-end, and (3) how the individual behavior of and interactions between mechanisms for static and dynamic priority management may impact end-to-end real-time behavior. The approach presented in this dissertation addresses questions left unanswered by the DSRTCORBA standard, and thus extends the state of the art in standards-based COTS middleware for distributed real-time systems, particularly those with requirements for adaptive and reflective management of QoS under changing operating conditions.

To maintain QoS assurances and to simplify testing for such demanding real-world real-time applications, we have extended prior work on TAO by providing a *strategized scheduling and dispatching framework* we have named *Kokyu*[1], for TAO. In particular, Kokyu focuses on applications with the following characteristics:

[1] *Kokyu* is a Japanese word meaning literally *breath*, but with implications of coordination and timing.

- *Stable epochs* – the behavior of the system over time can be described as a sequence of *epochs*, or periods during which system behavior is generally stable, with no significant variation in which operations are active, their rates, or costs of execution.
- *Variation between epochs* – the system behavior can change significantly from one epoch to another, and the number of significantly distinct behavioral states can be arbitrarily large.
- *Bounded executions* – operations are generally expected to stay within the limits of their specified execution times, and the total load on the system is expected to stay within feasible bounds. However, in our approach we also consider and address the effects of various levels of total system load and of randomized jitter in execution times, both

  1. within the advertised limits, and
  2. in excess those limits.

- *Known rates* – dispatch requests arrive and are executed within a specified period associated with an operation. For each operation, there is a set of one or more *possible* rates, though during each *epoch* of execution the rate of execution for each operation is fixed.
- *Known operations* – all operations are known to the scheduler before run-time, or are reflected entirely within the execution times of other specified operations. In addition to having different fixed rates (and possibly costs) of execution in different epochs, various subsets of the total set of operations may be enabled and disabled in each epoch.

Within these constraints, the Kokyu framework allows applications to (1) specify custom strategies for static and/or dynamic scheduling heuristics, (2) flexibly and automatically configure dispatching infrastructure to enforce the invariants of the chosen strategy, (3) upon a major shift in behavior, perform adaptive rescheduling of operations within closely bounded time-frames, and (4) adapt to minor variations in behavior by reflective alternation among multiple scheduling strategies. These capabilities allow applications simultaneously to (1) minimize risk to critical operations and (2) optimize performance of non-critical ones under a range of operating conditions. This framework

thus increases adaptability across application families and operating systems, while preserving the scheduling assurances and testability offered by previous work on statically scheduled CORBA operations. Furthermore, it offers a foundation for assurances of overall *performance*, as well as for critical deadline success.

1.4 Contributions

This section describes the research contributions of this dissertation in detail. Section 1.4.1 describes how the Kokyu scheduling framework extends and generalizes the previous-generation static TAO scheduling approach, to support arbitrary scheduling heuristics and improve CPU utilization while preserving criticality isolation. Section 1.4.2 describes how the reconfigurable Kokyu dispatching framework allows a customized fit between scheduling heuristics and application characteristics. Section 1.4.3 describes Kokyu scheduling framework extensions to support adaptive selection of operation rates at run-time. Section 1.4.4 describes a shared-memory-capable C++ framework for real-time performance data capture and feedback in constrained time and space. Finally, Section 1.4.5 describes how our framework, combined with our empirical measurements described in Chapter 8, supports optimized co-scheduling of applications and resource managers through run-time adaptive and reflective selection of scheduling heuristics. Table 1.1 summarizes the major research contributions of this dissertation.

1.4.1 Strategized Hybrid Scheduling Framework

The Kokyu framework began in an effort to extend and generalize the existing TAO static scheduling service [117]. The TAO static scheduling service allowed applications to specify key operation characteristics, such as period and worst-case execution time (WCET), and applied Rate Monotonic Scheduling (RMS) [70] to assign static priorities and Rate Monotonic Analysis (RMA) [70, 65, 56] to assess feasibility. The static scheduler produced a table of operations and priorities, which was then compiled and linked with a table-driven run-time scheduler. At run-time, a *dispatching module* looked up the assigned static priority for an operation to place each dispatch request for the operation into a first-in-first-out (FIFO) queue serviced by a thread at the appropriate operating system priority [41].

Table 1.1: Research Contributions

Technical Challenge	**Research Approach**	**Research Impact**
Increase CPU utilization over static approaches, but keep criticality isolation	Support arbitrary strategies that hybridize static/dynamic scheduling/dispatching	Increased utilization, critical operations still meet deadlines in overload
No one scheduling/dispatching strategy is optimal across all epochs of system behavior	Dispatching composed from primitive elements	Supports tailored fit of scheduling/dispatching
Closer bound on latency of adaptive re-scheduling	Integrated rate/priority selection mechanisms	Change from $O(n^2)$ to $O(n \lg n)$ or even $O(n)$ bound on adaptation time
Adaptation feedback in constrained time and space	Shared memory capable data cache with `inline` instrumentation methods	Low latency and small footprint for metrics feedback and recording
Optimized co-scheduling of resource managers and application components	Identify and empirically measure run-time observable characteristics to complement *a priori* analysis	Support for run-time reflective and adaptive policy selection

To support commonly used dynamic scheduling heuristics, such as Earliest Deadline First (EDF) [70] and Minimum Laxity First (MLF) [126, 71], as well as hybrid approaches such as RMS+EDF [70] and the Maximum Urgency First (MUF) [126] and RMS+MLF [15] strategies we examine in Chapter 8, we first generalize the priority and queue assignment approach, as Chapter 4 describes in detail. Second, we define a new configuration descriptor for dispatching queue, timer, and thread details based on the particular scheduling heuristic and operation characteristics specified.

1.4.2 Reconfigurable Dispatching Framework

To support flexible configuration of dispatching mechanisms in support of arbitrary scheduling policies, we have extended the design and implementation of the RMS dispatching module found in the original TAO Real-Time Event Channel (RTEC) [41]. First, we provide additional types of priority queues, to manage the ordering of dispatches according to policies such as EDF or MLF. Second, we show how flexible data structures can be used to support sharing and recycling of dispatching primitives such

as threads, timers, and queues. Third, we show how factories can be used for automatic configuration and reconfiguration of the dispatching infrastructure, using the configuration descriptors described in Section 1.4.1. Chapter 5 describes the design and implementation details of this dispatching framework in detail.

1.4.3 Optimized Rate Selection Technique

To improve the observed real-time performance of the Kokyu framework's interaction with higher-level resource managers, we extend the basic operation scheduling technique to support both rate selection and priority assignment in more closely bounded time. In particular, we reduce the observed $O(n^2)$ behavior to $O(n \lg n)$ and sometimes $O(n)$ in special cases by re-casting both rate selection and priority assignment as sorting problems. We support alternative rate selection strategies, and describe two representative algorithms, described in more detail in Chapter 6.

1.4.4 Performance Monitoring and Feedback

To support monitoring and control by higher-level resource managers, we provide a metrics infrastructure for instrumenting application and infrastructure components to gather, store, and propagate fine-grained real-time performance data within stringent time and space constraints. Chapter 7 documents (1) a shared-memory-capable data cache with `inline` instrumentation probes, (2) a common time frame manager across threads and rates of execution, (3) interceptors for measuring, and possibly cancelling operation dispatches at run-time, (4) monitoring interfaces for use both by higher-level resource managers and in the reflective alternation of heuristics described in Section 1.4.5, and (5) integration with external data logging and visualization services.

1.4.5 Towards Adaptive Selection of Scheduling Heuristics

The most important contribution of this dissertation, which builds on the other four major contributions, is to demonstrate the benefits of alternation among several real-time scheduling heuristics at run-time, and suggest techniques for control of alternation using run-time reflection on the measured real-time behavior. Because

1. higher-level resource managers (such as, for example, the Real-Time Adaptive Resource Manager (RTARM) [44] described in Section 6.1) compete for the same

resources (*i.e.*, memory, CPU cycles, network bandwidth) as the applications they intend to manage, and

2. these same resource managers may themselves have both *critical* and *non-critical* operations, *e.g.*, staying in synch with the state of the system versus optionally performing adaptation,

we believe the problem of co-scheduling resource managers and applications in distributed rate-based real-time systems is both important and relevant to the work presented in this dissertation.

In particular, Chapter 8 presents empirical studies of three canonical heuristics, RMS, MUF, and RMS+MLF, each with and without operation cancellation, for a realistic avionics mission computing application running in a realistic middleware, operating system, and hardware setting. Chapter 9 describes open problems raised by this research, including the consideration in Section 9.4 of whether control laws for adaptation among heuristics, based on the empirical results presented in Chapter 8, can be

1. implemented effectively and efficiently using information available at run-time, and either
2. identified where possible for categories of applications, or
3. learned for a particular application.

1.5 Chapter Organization

The remainder of this dissertation is organized as follows. Chapter 2 surveys related work in the areas of real-time scheduling heuristics and frameworks, and real-time QoS management in operating systems and middleware. Chapter 3 provides an overview of the Kokyu framework, and describes the role it plays in supporting research extensions to an example avionics application. Chapter 4 describes the implementation details of the scheduling portion of the Kokyu framework, and describes architectural principles applied to improve the flexibility and power of the framework. Chapter 5 describes details of the flexible Kokyu dispatching infrastructure, including innovations for efficient support of adaptive reconfiguration at run-time. Chapter 6 presents results of early integration of the Kokyu scheduler with a RTARM [44] and describes the subsequent

refactoring and extension of the Kokyu scheduler to reduce the worst case time complexity bound for adaptive rescheduling. Chapter 7 describes the implementation of a performance measurement framework designed for low-latency and reduced-footprint collection and propagation of real-time performance data, intended for use in several ways:

1. for empirical analysis and visualization,
2. for run-time reflective evaluation of operating conditions,
3. for possible use in alternation among scheduling heuristics, and
4. for closed-loop run-time feedback to multiple higher-level QoS managers (such as, for example, the RTARM [44] and Quality Objects (QuO) [143]).

Chapter 8 presents empirical studies to evaluate the real-time performance of a realistic application with each of six distinct scheduling heuristics, and describes observed properties of the run-time behavior that appear useful to guide adaptation decisions using the Kokyu framework. Chapter 9 presents conclusions about the findings, potential impact, and indicated future directions of this work. Finally, Appendix A presents a synopsis and illustrations of key real-time scheduling terminology used in this dissertation.

Chapter 2

Survey of Related Work

Distributed real-time and embedded computing is an emerging field of study. Research efforts are focusing increasingly 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 like that in the Common Object Request Broker Architecture (CORBA) specification. Pioneering efforts to provide meta-capabilities such as configuration flexibility, reflection, and ultimately adaptation, while still meeting strict QoS assurances are beginning to extend this field. This work contributes to the emergence of that sub-field of distributed real-time and embedded systems research.

In this chapter we discuss representative work that is related to our approach. To motivate issues in the particular domain where we have conducted our empirical studies described in Chapter 8, Section 2.1 considers two alternative approaches to avionics platform QoS management. Section 2.2 examines research on adaptive systems in general, and draws connections to areas of future work described in Chapter 9. Section 2.3 describes CORBA-related QoS middleware research, including previous work in ADAPTIVE Communication Environment (ACE) and The ACE Object Request Broker (ORB) (TAO), upon which this research builds, and which this research extends. Section 2.4 considers related non-CORBA QoS middleware research, and relates our work to that research context. Section 2.5 describes related research on QoS in operating systems, including support for real-time scheduling and dispatching. Finally, Section 2.6 examines scheduling research, both to survey the classical scheduling literature upon which this work is built, and to note new approaches for managing variability in task loads, durations, and even adaptive scheduling decisions.

2.1 Avionics Platform Research

Two main branches of research are endeavoring to make QoS managed systems infrastructure a prevalent, and particularly a *reusable*, feature of avionics flight software systems. We first describe related work that seeks to standardize QoS management features in avionics domain-specific platforms. We then describe related work based on open systems and commodity-off-the-shelf (COTS) components.

Avionics Domain Platform Research: Standardized avionics platforms such as, for example, the ARINC Avionics Application Software Standard Interface (APEX) for Integrated Modular Avionics (IMA) [2] provide QoS assurances for systems in the avionics domain. IMA supports componentization so that different software components could be made by different vendors and still provide necessary assurances when brought together.

McElhone [78] examines the question of how to support operations with soft real-time constraints and possibly long running or variable length computations, in canonical avionics platforms such as IMA. He surveys current work in the area, and describes an overall vision for implementation through mapping those more complex kinds of operations to simpler infrastructure abstractions. We consider our work to be in keeping with this vision, though we focus on a more general open systems vision of middleware support (described next), of which avionics systems represent one application domain.

Open Systems Avionics Research: Winter, Sharp, Doerr, *et al.* [140, 120, 121, 24], 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 infrastructure, hosted on COTS standards-based hardware, operating systems, and middleware, 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.

2.2 Adaptive Systems Research

We examine three examples of research we characterize as belonging to the adaptive systems area. Within this area, we consider:

- adaptive QoS control middleware, within which application components are hosted
- defense-enabled systems, in which adaptive QoS management infrastructure is used to detect and survive intrusions
- an open controls platform, which is a coordination and adaptation middleware on which control system software components are hosted

Adaptive QoS Control: Li and Nahrstedt [69, 68] apply control theory to adaptive QoS management in distributed systems, and present a middleware framework for feedback control, using a task control model. Of particular interest is their examination of *adaptation agility*, which is a measure of the adaptation mechanisms to respond promptly to sudden and unexpected changes in the QoS environment. Where their approach is focused on applications with a streaming QoS model, we address the rate-based distributed real-time systems domain. However, several key ideas from that work, particularly the feedback architecture and configurability of adaptation agility (*e.g.*, to avoid hysteresis as we describe in Section 9.4) appear applicable to our work.

Defense-Enabled Systems: Webber, Cuckier, Pal, Loyall, *et al.*, describe how QoS aware middleware can be used to develop *defense-enabled* systems, with increased resistance to malicious attack even in the face of an untrustworthy environment [134, 101, 22, 100, 73]. To allow a system to participate in its own defense, they describe key policies and mechanisms (*e.g.*, for intrusion detection), using specialized infrastructure at the operating system and middleware levels. As we describe in Section 9.3.3, we believe the Kokyu framework could be extended readily to provide defense-enabling capabilities for real-time distributed systems, similarly using its own existing QoS management infrastructure.

Open Control Platform: Wills, Kannan, Rufus, *et al.*, at the Georgia Institute of Technology, describe algorithms and an integrated infrastructure for adaptation, transition and control, with application to unmanned aerial vehicles [138, 137, 113, 50, 112]. We believe our approach is highly complementary to their approach. In particular, by addressing the open problems of local and distributed transition management described in Sections 9.3.1 and 9.3.2, we will be able to provide a key solution to a part of the larger adaptation management problem they are addressing.

2.3 CORBA-related QoS Middleware Research

In this section, we survey related work in the area of CORBA-related QoS middleware. We identify relevant standards, and describe a body of previous research in TAO, upon which this work builds. We describe a number of related middleware research efforts, particularly those with a focus on scheduling or other forms of adaptive QoS management.

Standard Specifications: The approved Real-Time CORBA 1.0 [91] 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 approaches such as Dynamic Scheduling Real-Time CORBA 2.0 (DSRTCORBA) [92] and the non-CORBA Real-Time Specification for Java^{TM} (RTSJ) [12] standard, 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. In its current form, Kokyu is already accessible to DSRTCORBA under the C++ language binding.

To address the problem of heterogeneous scheduling policies on different end-systems raised by DSRTCORBA in particular, Corsaro defines *Juno* [19], a meta-level model for reconciling properties of diverse scheduling heuristics. The research described in this dissertation raises issues, including the adaptive transitions and multi-endsystem adaptation cuts described in Chapter 9, where we believe the Juno model is useful to transform properties and values of invocation dispatch requests.

TAO Real-Time ORB Research: Previous work on TAO has examined many dimensions of ORB middleware design, including static [117] operation scheduling, event processing [41], I/O subsystem [60] and pluggable protocol [96] integration, both synchronous [118] and asynchronous [4] ORB Core architectures, IDL compiler features [3] and optimizations [35], systematic benchmarking of multiple ORBs [33], patterns for ORB extensibility [119] and ORB performance [106]. This earlier work provides the foundation for our research on generalizing, hybridizing, and optimizing static [117] and dynamic [32] scheduling in the flexible middleware framework described in this research.

Finally, Pyarali [102] has provided crucial infrastructure for implementing the Real-Time CORBA 1.0 [91] specification in TAO, has identified and articulated patterns for distributed real-time systems software, and has demonstrated the ability to provide end-to-end priority isolation in an open systems context. We view these results as providing crucial lower-level middleware services upon which our flexible and adaptive approach may rely to maintain key QoS assurances end-to-end, particularly as we apply our approach to increasingly open systems.

Mitre Real-time CORBA: Krupp, *et al.*, [129] at the MITRE Corporation were among the first to elucidate the requirements of real-time CORBA systems. A system consisting of a COTS real-time OS, a COTS CORBA ORB, and a real-time OO database management system is under development [131]. Similar to TAO's original static scheduling service [117], their initial static scheduling approach used RMS, though a strategy for dynamic deadline monotonic scheduling support has been designed [17].

URI TDMI: Wolfe, *et al.*, developed a real-time CORBA system at the US Navy Research and Development Laboratories (NRaD) and the University of Rhode Island (URI) [141]. The system supports expression and enforcement of dynamic end-to-end timing constraints through timed distributed method invocations (`TDMI`s) [26]. A `TDMI` corresponds to TAO's `RT_Operation` abstraction [117]. Their `RT_Environment` structure contains QoS parameters similar to those in TAO's `RT_Info` abstraction. One difference between the TAO and URI approaches is that `TDMI`s express required timing constraints, *e.g.*, deadlines relative to the current time, whereas `RT_Operation`s publish their resource, *e.g.*, CPU time, requirements. The difference in approaches may reflect the different time scales, seconds versus milliseconds, respectively, and scheduling requirements, dynamic versus static, of the initial application targets. However, the approaches should be equivalent with respect to system schedulability and analysis.

In addition, NRaD/URI supply a new CORBA Global Priority Service, analogous to the Kokyu Scheduling Service, and augment the CORBA Concurrency and Event Services. The initial implementation uses *EDF within importance level* dynamic, on-line scheduling, supported by global priorities. A global priority is associated with each `TDMI`, and all processing associated with the TDMI inherits that priority. In contrast, TAO's initial Scheduling Service was static and off-line; it used importance as a "tie-breaker" following the analysis of other requirements such as data dependencies. Both

NRaD/URI and Kokyu readily support changing the scheduling policy by encapsulating it in their CORBA Global Priority and Scheduling Services, respectively.

BBN QuO: The *Quality Objects* (QuO) distributed object middleware is developed at BBN Technologies [143]. 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), (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, and (3) *code mobility* that enables QuO to migrate object functionality into local address spaces in order to tune performance and to further support highly optimized adaptive reconfiguration.

The QuO model employs several *QoS definition languages* (QDLs) that describe the QoS characteristics of various objects, such as expected usage patterns, structural details of objects, and resource availability. QuO's QDLs are based on the separation of concerns advocated by Aspect-Oriented Programming (AoP) [52]. The QuO middleware adds significant value to adaptive real-time ORBs such as TAO. We are currently collaborating with the BBN QuO team to integrate the TAO, Kokyu, and QuO middleware infrastructures within the WSOA program.

UCSB Realize: The Realize project at UCSB [49] supports soft real-time resource management of CORBA distributed systems. Realize aims to reduce the difficulty of developing real-time systems and to permit distributed real-time programs to be programmed, tested, and debugged as easily as single sequential programs. Realize integrates distributed real-time scheduling with fault-tolerance, fault-tolerance with totally-ordered multicasting, and totally-ordered multicasting with distributed real-time scheduling, within the context of OO programming and existing standard operating systems. The Realize resource management model can be hosted on top of TAO [49].

Kalogeraki, *et al.*, at UCSB have developed an approach based on object migration and replication, to improve performance of soft real-time distributed systems [48, 47, 46, 49]. This approach constitutes a higher level of adaptive control for soft real-time QoS management, and is complementary to our approach. In particular, a system developer might use the UCSB infrastructure to provide effective distribution of soft

real-time load across endsystems using the Kokyu framework to integrate scheduling and dispatching of both critical and non-critical load.

UIUC Epiq: The Epiq project [27] defines a real-time CORBA mechanism that provides QoS guarantees and run-time scheduling flexibility. Epiq explicitly extends TAO's original off-line scheduling model to provide on-line scheduling. In addition, Epiq allows clients to be added and removed dynamically via an admission test at run-time.

UCI TMO: The Time-triggered Message-triggered Objects (TMO) project [53] 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.

These additional features require the following capabilities [54]: (1) timely invocation and execution of time-triggered operations, (2) timely transmission of remote invocation requests from the client to the server, (3) timely handling of remote invocation requests on the server, and (4) timely execution of message-triggered operations. The TMO project supports these capabilities through two new CORBA services, called the TMO Execution Support (TMOES) and Cooperating Network Configuration Management (CNCM) services. The TMOES service allows applications to specify timing requirements, and enforces these requirements for TMO operations on each ORB endsystem. The CNCM service establishes communication channels that can deliver remote invocation requests within the necessary timing constraints.

TAO differs from TMO in that it provides a complete CORBA ORB, as well as CORBA ORB services and real-time extensions. Kokyu generalizes the timer-based invocation capabilities provided through TAO's Real-Time Event Service [41, 98]. Where the TMO model creates new ORB services to provide its time-based invocation capabilities [54], TAO provides a subset of these capabilities by extending the standard CORBA COS Event Service. We believe 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.

CORBA (m,k)-Firm Scheduling: Montez, *et al.*, present an approach based on hybridizing polymorphic invocation and (m,k)-firm scheduling assurances [82] in CORBA. Their approach offers an interesting balance of flexibility in missing some deadlines, with firm bounds on the number of such misses over a set of dispatches. This approach could prove beneficial for scheduling adaptive resource managers such as the Real-Time Adaptive Resource Manager [44] or QuO [143], and we plan to investigate this approach for implementation in our Kokyu framework.

2.4 Non-CORBA QoS Middleware Research

In addition to the CORBA-related QoS middleware research described in Section 2.3, we also survey QoS middleware research conducted outside CORBA. These alternatives serve to round out the current state of real-time QoS research in middleware, completing the context within which this dissertation makes its contributions.

Utah CRM: Regehr and Lepreau [111] propose the CPU Resource Manager, a middleware service for managing processor allocation using scheduling abstractions provided by COTS operating systems. They examine conversions between 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.

Real-Time Adaptive Resource Manager (RTARM): The RTARM was developed jointly by the Honeywell Technology Center, Texas A&M University, and the Georgia Institute of Technology adaptive resource manager [44]. It focuses on monitoring and adaptive management of run-time QoS. We have integrated our work with the RTARM in two collaborative research programs directed by Boeing. The first of these programs (ASTD) produced the empirical results that motivated the scheduling optimizations described in Chapter 6, which were in turn implemented in the second program (WSOA), and experiments to quantify the impact of those optimizations in a realistic system are currently underway.

DeSiDeRaTa: The DeSiDeRaTa project [136] was developed at the University of Texas, Arlington and Ohio University. DeSiDeRaTa focuses on a real-time path abstraction along which QoS properties are configured and enforced.

ARMADA: The ARMADA project [79, 1] defines a set of communication and middleware services that support fault-tolerant and end-to-end guarantees for real-time distributed applications. ARMADA provides real-time communication services based on the X-kernel and the Open Group's MK micro-kernel. This infrastructure provides a foundation for constructing higher-level real-time middleware services.

CMU Publisher / Subscriber: Rajkumar, *et al.*, [108] at CMU developed a real-time Publisher / Subscriber implementation model that is similar to TAO's Real-time Event Service [41], *e.g.*, it uses real-time threads to prevent priority inversion within its communication framework. The CMU model does not utilize any QoS specifications from publishers (event suppliers) or subscribers (event consumers), however. Therefore, scheduling is based on the assignment of request priorities, which is not addressed by the CMU model. In contrast, the Kokyu framework extends a previous-generation model of application-specified QoS descriptors for suppliers and consumers [117] to include characteristics such as operation criticality.

Real-Time Producer / Consumer: Jeffay, *et al.*, [45] introduced an earlier model of real-time computations that is also similar to the CMU Publisher / Subscriber work, and the TAO Event Channel, except that producers and consumers are defined in programming language constructs, and relationships between them are represented explicitly in a process graph. Jeffay's work extended the well-known producer / consumer model of inter-process communication to real-time systems.

2.5 Operating Systems Research

Operating systems research in QoS management is an essential counterpoint to this dissertation. Where middleware is concerned with end-to-end coordination of QoS properties, and in vertical integration of application requirements and underlying operating system resources, operating systems play a similar intermediary role, between the middleware (or application if there is no middleware), and the underlying hardware. We focus on three distinct approaches to real-time QoS extension of the Linux operating system, and one approach to scheduling in Scout, a path-based operating system.

KURT Linux: Srinivasan, Niehaus, *et al.*, have implemented real-time capabilities in the form of Linux kernel patches to provide the KU Real-Time (KURT) Linux operating system [122]. KURT offers fine-grained temporal resolution (UTIME), while providing a full-featured Linux environment. KURT also provides a planned scheduling facility, though arbitrary scheduling modules may be plugged into KURT.

In addition to KURT, the KU real-time researchers have developed a data streams kernel interface (DSKI) [88] that provides some similar capabilities to the metrics infrastructure described in Chapter 7. In particular, the DSKI offers flexible data collection points triggered by traversal by a thread in the running system. We believe the Kokyu data collection and streaming framework complements the DSKI, offering specific support for distributed data collection and propagation across multiple endsystems.

Resource Kernel: Oikawa and Rajkumar [95] have developed a portable resource kernel to provide timing guarantees for applications with time-multiplexed resources. This work has led to an implementation in Linux, called Linux/RK (which formed the basis for a commercial version called TimeSys Linux), with similar objectives to that of KURT Linux.

Scout OS: Scout [83, 84] is a communications-oriented path-based operating system. The dominant abstraction in Scout is a *path* [85], along which functional and QoS management capabilities are composed. The Best-Effort Real-Time (BERT) Scheduler [10] implemented for Scout establishes two classes of service, one with timeliness requirements and one with progress and fairness requirements. While our approach focuses on critical and non-critical classes of real-time service, we identify and enforce timeliness requirements for the non-critical class of service as well, to the extent possible.

UCI RED-Linux Scheduling Framework: Wang, *et al.* [133], at the University of California, Irvine, have proposed a general scheduling framework 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. Therefore, our dispatching mechanisms

can augment standards-based CORBA middleware and it can perform effectively on a wide range of commonly available real-time and general-purpose OS platforms.

In addition, the Kokyu framework differs from the RED-Linux scheduling framework in its emphasis on policy implementation. Whereas the RED-Linux implementation focuses on the details of how policies can be enforced by the OS kernel, Kokyu emphasizes encapsulation of these details to decouple each application from any particular scheduling policy. Wang, *et al.*, point out that the *allocator* portion of their framework be implemented as a middleware service, which suggests that the Kokyu and RED-Linux scheduling approaches are complementary.

2.6 Scheduling Research

We conclude by examining the past and current scheduling literature, with special emphasis on how the classical scheduling approaches relate to newer research, and in particular to this dissertation. We survey classical open-loop approaches, approaches geared toward adaptive *tolerance* of variations in operating conditions, and finally a crucial approach toward adaptive *control* of real-time QoS.

Classical Scheduling: Liu and Layland pioneered the Rate Monotonic Scheduling (RMS), Earliest Deadline First (EDF), and RMS+EDF [70] real-time scheduling algorithms . Additional scheduling algorithms of classical interest include Minimum Laxity First (MLF) [126, 71], which adds consideration of execution times to EDF. Chung, *et al.*, define RMS+MLF [15], which similarly adds consideration of of execution times to RMS+EDF. Stewart and Khosla define Maximum Urgency First (MUF) [126], which is in its simplest form defined as MLF+MLF. Notably, Stewart and Khosla show that MUF can emulate any of RMS, EDF, or MLF by appropriate configuration of a more general set of operation characteristics. Our approach *encapsulates* these mappings from operation characteristics to urgency as described in Section 4.3, and extends that mapping to configurations of primitive dispatching primitives as described in Section 4.4. In this way, we extend the emulation idea from MUF to provide flexible and *modular* scheduling and dispatching support for each of these classical scheduling algorithms.

OSU Share-based Scheduling: Tyan, *et al.* [130], at Ohio State University, have developed a general framework for share-based scheduling. They demonstrate their framework's ability to implement a number of well-known fair queueing algorithms, as well as

its ability to implement new kinds of share-based scheduling algorithms. Our research differs in that it uses priority based scheduling approaches, to address applications with hard real-time requirements. In our future research, we plan to investigate share-based scheduling and in particular its interaction with priority-based scheduling for various classes of real-time applications. Han and Tyan also provide a polynomial-time schedulability test [39] that appears useful for application and extension for (1) rate-based adaptation, and (2) migration among hybrid static/dynamic scheduling heuristics.

Flexible Computations: In addition to defining the RMS+MLF scheduling strategy, Chung, *et al.*, define a model for flexible *imprecise* computations [15] in which some initial segment of a computation may be critical and require strict assurances, while additional segments of the computation may be scheduled or not without harm to the application. The model that underlies our target platform described in Chapters 1 and 3, and again in Section 8.1, differs somewhat from the model of Chung, *et al.* [15]. In their model each operation may have a mandatory part followed by an optional part. A similar effect can be achieved in our approach by making an optional operation's task depend on a mandatory one's task. For example, a chain of *route leg* computations as described in Section 8.1 could be implemented in this way.

(m,k)-Firm Scheduling: Hamdaoui and Ramanathan define the (m,k)-firm scheduling algorithm [37], in which m out of any k consecutive dispatches of an operation are assured of completion prior to deadline. This work forms the basis of the CORBA approach suggested by Montez, *et al.* [82], which we describe in Section 2.3.

Statistical RMS: Atlas and Bestravos define the Statistical Rate Monotonic Scheduling (SRMS) algorithm [5] and describe an implementation of that algorithm in KURT Linux [6]. SRMS offers an alternative to (m,k)-firm scheduling where operations cannot be feasibly scheduled for all their deadlines. We have recently developed a set of prototype extensions to the middleware scheduling and dispatching infrastructure described in Chapters 4 and 5, to implement SRMS scheduling for decision aiding tasks [30].

Multi-Level Scheduling: Multi-level scheduling integrates different approaches at different *levels* of scheduling. One example is two-level hierarchical scheduling, which allows real-time applications to coexist with non-real-time applications in an open OS environment [23]. Another is standardized in the ARINC Avionics Application Software Standard Interface (APEX) for Integrated Modular Avionics (IMA) [2]. One level

consists of *partitions*, which are executed cyclically and scheduled statically and off-line. Within each partition, application *processes* are scheduled using potentially more flexible approaches.

Spring Kernel: The Spring kernel [81] was designed and developed at the University of Massachusetts, Amherst. Spring uses admission control to schedule newly arrived tasks. A new task will be admitted only if the set of tasks consisting of the newly arrived task, and all tasks that have already been admitted, can be scheduled. This incremental admission control strategy typifies many dynamic scheduling schemes.

Ramamritham, *et al.*, describe and evaluate algorithms for planned schedule generation, based on heuristic functions [110]. Each of these approaches incrementally constructs a schedule by selecting one task at a time to the schedule, based on the heuristic with which the algorithm is parameterized. Interestingly, they show that the heuristics perform well, finding a feasible schedule with high probability in many cases. This approach complements the Spring scheduling approach, as it supports on-line admission control and efficient incremental and dynamic generation of planned schedules.

Feedback Control Scheduling: One of the most important areas of related work is the pioneering research on feedback control real-time scheduling (FCS), conducted by Stankovic, Lu, *et al.*, at the University of Virginia. They apply control theory to real-time scheduling [76, 77, 123, 74, 75, 124] for soft real-time systems, to reduce the number of missed deadlines at run-time.

The primary difference between the FCS work and ours is that they focus on controlling a single performance metric, the deadline performance of soft real-time tasks. Our research is aimed primarily at distributed rate-based systems where at least two classes of operations are present, and deadlines for the highest class must be assured before soft real-time performance is optimized. For example, they specify a *miss ratio function* [74], which is comparable to our $srt\_fraction$ function described in Section 8.3.3, though they measure the fraction of srt deadlines missed, rather than made. However, we then predicate this raw measure of soft real-time performance with whether *any* critical deadlines have been missed in each sample, to obtain a performance metric that considers multiple criticality levels.

Another difference is that while Stankovic, Lu, *et al.*, focus on control adaptation *within* a scheduling strategy, our approach considers adaptation at a level just *above* the scheduling strategy. In particular, we focus on dispatching, scheduling, and performance

feedback mechanisms to support integrated adaptive QoS management at many levels of middleware [25, 72, 29]. We consider the feedback control scheduling research an important area of study, and we plan to investigate how the techniques from control theory demonstrated at the level of adaptation within a schedule could be applied to adaptation (1) between scheduling strategies, (2) across discrete rates within a scheduling strategy, and (3) end-to-end across distribution boundaries in distributed real-time systems.

Chapter 3

Overview of the Kokyu Framework

Next-generation mission-critical distributed real-time and embedded (DRE) systems, such as integrated avionics mission computing systems [67], teams of emergency rescue robots [142], and distributed real-time automobile management systems [40], must adapt swiftly to changing environmental conditions. Greater coordination allows elements at all levels to identify and respond effectively to transient opportunities and hazards. Achieving significant levels of coordination requires DRE systems with the ability to:

1. Accommodate unplanned tasks and evolving task characteristics in a distributed environment with rapidly changing information and resource availability conditions;
2. Trade performance of individual elements for system-level real-time performance objectives, and optimize real-time performance across heterogeneous criteria, such as reducing the rates of critical operations to allow more non-critical operations to be scheduled, lowering the priority of non-critical operations to ensure feasibility of critical operations, or using alternate scheduling heuristics to improve real-time performance;
3. Perform adaptive resource reallocations within firmly bounded time-scales.

Historically, many DRE systems of these kinds have been developed largely from scratch, using handcrafted optimizations on each endsystem and network node to achieve the coordination and performance goals outlined above. Unfortunately, expectations of increasing scale and decreasing development cycles make it hard to sustain

this development model in a cost-effective manner over long DRE system lifecycles. Solutions built instead using standards-based COTS middleware promises greater reuse of software architectures, patterns, frameworks, analysis techniques, and testing and certification results across entire families of systems.

Next-generation DRE systems also require explicit interfaces and mechanisms for key capabilities, such as fine-grain adaptive rescheduling, that are not available in today's COTS middleware solutions, such as Real-Time CORBA 1.0 [91]. Emerging COTS middleware approaches, such as Dynamic Scheduling Real-Time CORBA [92] and the RTSJ [12], add some elements for implementing these capabilities (*e.g.*, enhanced distributable threading models and real-time behavioral descriptors).

However, additional (and *unified*) higher-level approaches and services are still required to realize the full real-time performance benefits achievable with closer integration of scheduling mechanisms in middleware. Middleware is uniquely suited to address both (1) application-specific constraints such as whether or not operation rates are known in advance, and (2) optimized integration of common mechanisms to support flexible trade-offs within a common reusable infrastructure. Neither lower layers such as operating systems and network protocol stacks, nor higher layers such as domain-specific libraries or applications themselves, are appropriate contexts in which to combine these issues. Rather, middleware serves to mediate the higher and lower level concerns and can achieve improvements in both flexibility and performance through its appropriate interactions upward and downward in the overall system architecture.

To achieve both (1) reuse and flexibility across families of systems and (2) optimized real-time performance in DRE systems, this dissertation describes the following enhancements to current real-time middleware scheduling approaches:

- Hybridizing static and dynamic scheduling techniques to optimize run-time performance and relieve requirements for *a priori* knowledge of exact resource allocations and the order of transitions between allocations;
- Support for variable period tasks, to exploit degrees of freedom in performance of individual elements to achieve system-wide real-time properties;
- Flexible policies and integrated mechanisms for selecting periods and determining execution eligibility, to apply this approach effectively across arbitrary operation characteristics, while achieving rapid local adaptation to run-time variations in system requirements and resource availability.

The primary contributions of this research are implemented in *Kokyu*, which is an open-source middleware framework that supports adaptive distributed quality of service (QoS) management in real-time embedded middleware. The Kokyu project also serves as a foundation for ongoing efforts to identify and document design patterns [102] for integrated real-time QoS management in distributed real-time and embedded mission-critical systems.

3.1 Overview of the Target Platform

This section describes key features of the platform upon which our work is based. Figure 3.1 illustrates the architecture of the distributed OO avionics mission comput-

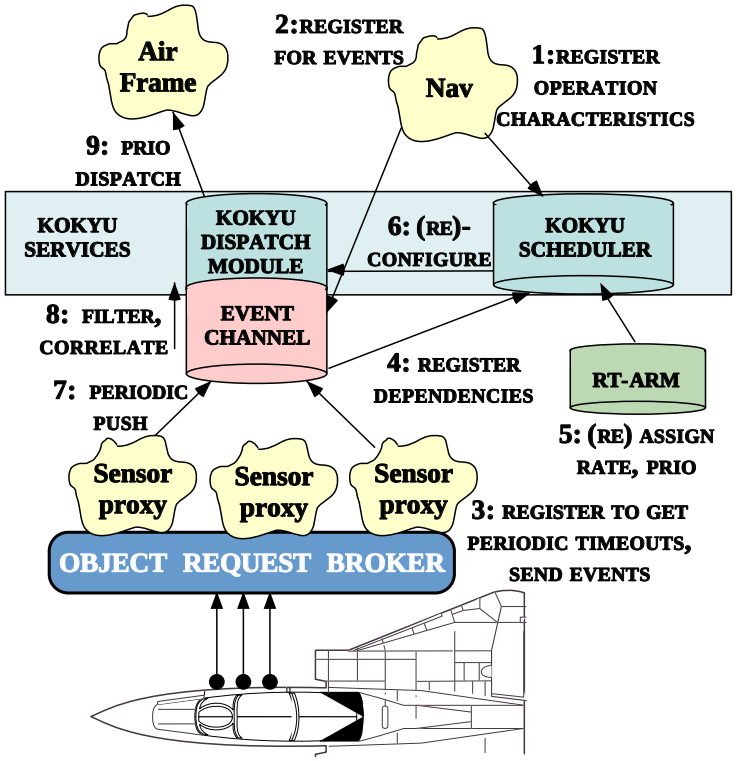

Figure 3.1: Kokyu in an Avionics Context

ing platform [41] used for the research we present in this dissertation. This platform was developed and deployed using OO middleware components and services based on

CORBA [90]. Key characteristics of the target platform that shape our middleware-based optimization approach are described below. These characteristics are shared by many other DRE systems, as well.

Operations and Tasks: Depending on the extent to which a specific application composes multiple operations within a single schedulable task, the number of schedulable tasks is on the order of 50-100. Some operations, such as computing the first leg of a navigation route, are *mandatory* and must finish before their deadlines. Other operations, such as computing subsequent legs of the route, are *optional*.

Variable Periods: Each task has a (possibly unary) harmonic set of discrete rates at which it can run, and the union of all these sets of rates is also harmonic. In our current research, rate reallocations are controlled by a Real-Time Adaptive Resource Manager (RTARM) [44]. RTARM is a middleware service developed by Honeywell that adapts the rates of tasks according to changing environmental conditions [25].

In our initial research [67, 25], we specified that a task would have the same execution time across all rates. Our current research [72], however, has revealed uses for variable execution times across available rates. Most notably, we use variable execution times to provide finer granularity decomposition for execution of optional operations.

Dependencies: The tasks may have precedence dependencies, resulting in a directed acyclic graph (DAG) over all operations that is established during or before application initialization. For example, an operation with a mandatory part and an optional part can be modeled in our approach with separate tasks, a mandatory one for the mandatory part and an optional one for the optional part, with a dependency of the optional operation's task on the mandatory one's task.

Tasks may be enabled or disabled at run-time by the application or a middleware resource manager, such as the RTARM. The application or a middleware resource manager may also enable and disable dependencies independently, subject to the constraint that an edge in the dependency DAG is treated as enabled at any given time *if and only if* it is enabled and connects two enabled tasks.

3.2 The Kokyu Framework

Kokyu is a portable middleware scheduling framework designed to provide flexible scheduling and dispatching services within the context of higher-level middleware, such

as The ACE ORB [13] (TAO). As shown in white in Figure 3.2, Kokyu currently pro-

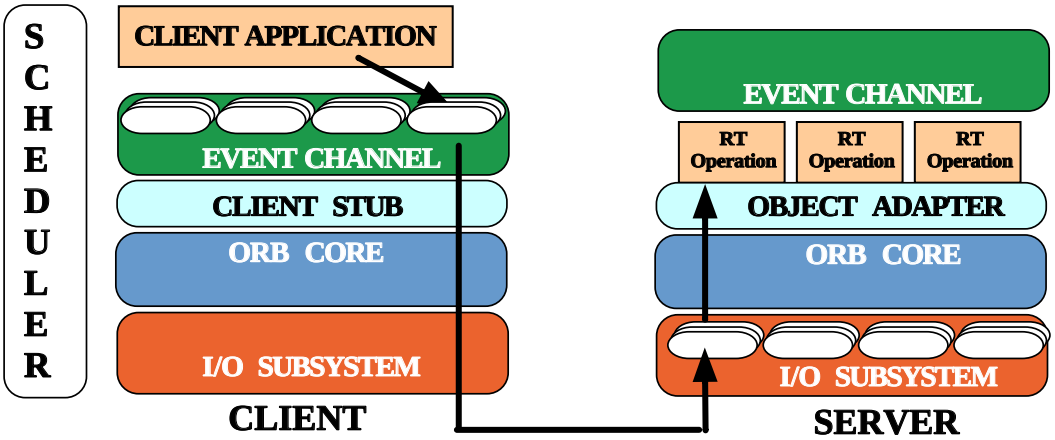

Figure 3.2: Kokyu Services used by TAO

vides real-time scheduling and dispatching services for TAO's real-time CORBA Event Service [41], which mediates supplier-consumer relationships between application operations. Figure 3.2 also illustrates further potential applications of Kokyu services to TAO, including early (*i.e.*, low-layer) scheduling control of request upcalls on server-side ORB endsystems.

Kokyu consists primarily of two cooperating infrastructure segments, illustrated in Figure 3.3:

1. A pluggable scheduling infrastructure with efficient support for adaptive execution of diverse static, dynamic, and hybrid static/dynamic scheduling heuristics; and
2. A flexible dispatching infrastructure that allows composition of primitive operating system and middleware mechanisms to enforce arbitrary scheduling heuristics.

The *scheduler* is responsible for *specifying* how operation dispatch requests are ordered, by assigning priority levels and rates to tasks, and producing a configuration specification for the dispatching mechanism. The *dispatcher* is responsible for *enforcing* the ordering of operation dispatches using different threads, requests queues, and timers

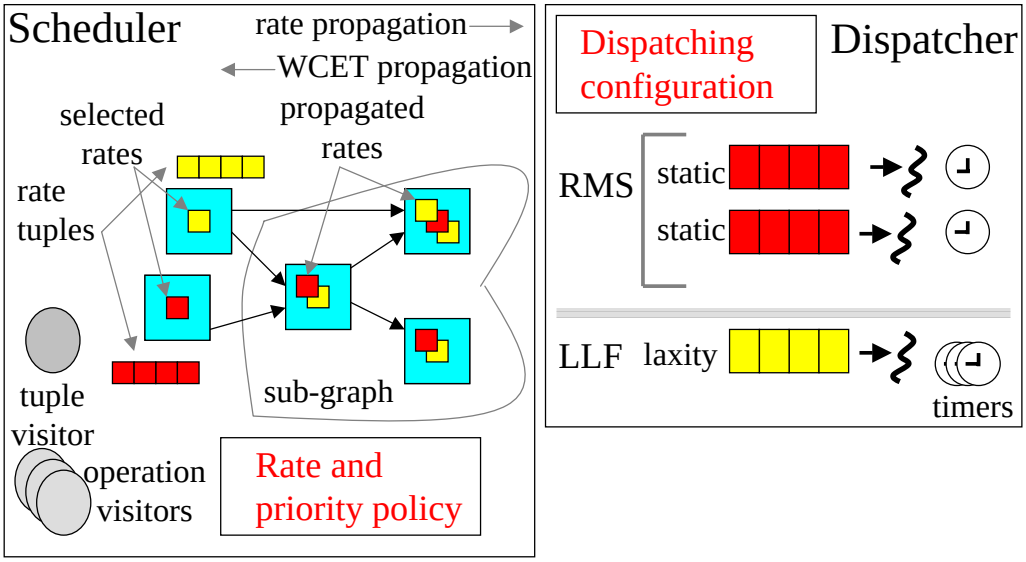

Figure 3.3: Kokyu Scheduling and Dispatching Infrastructure

configured according to the scheduler's specification. The combined framework provides an implicit projection of scheduling heuristics into appropriate dispatching infrastructure configurations, so that the scheduling and dispatching infrastructure segments can be optimized both separately and in combination.

3.3 Strategized Scheduling

The Kokyu scheduling framework is designed to support a variety of scheduling heuristics discussed earlier, including RMS, EDF, MLF, and MUF. In addition, this framework provides a common environment to compare systematically both existing and new scheduling strategies. This flexibility is achieved in the Kokyu framework via the *Strategy* pattern [28], which allows parts of the sequence of steps in an algorithm to be replaced, thus providing interchangeable variations within a consistent algorithmic framework. The Kokyu scheduling framework uses the Strategy pattern to encapsulate a family of scheduling algorithms within a fixed CORBA IDL interface, thereby enabling different strategies to be configured *independently* from applications that use them.

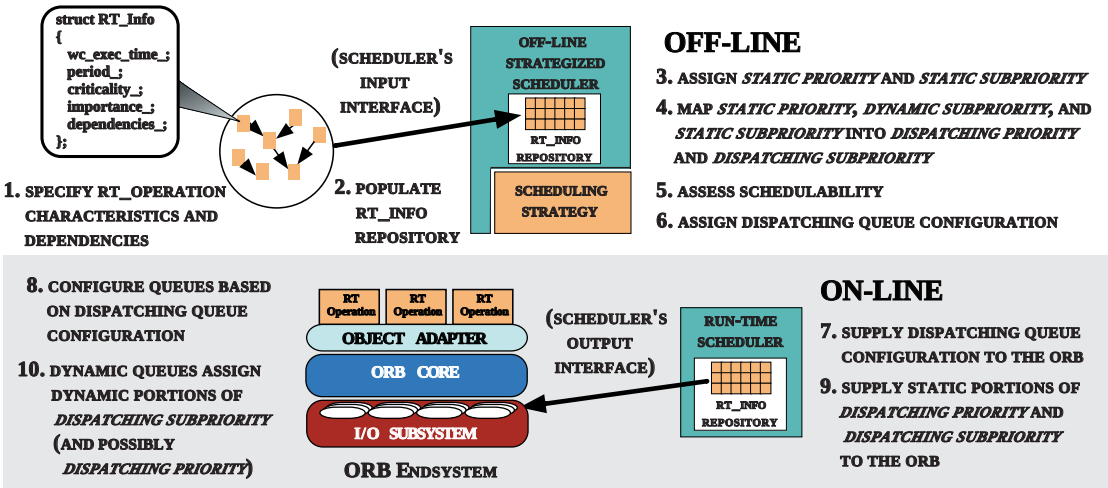

Figure 3.4: Processing Steps in the Kokyu Service Architecture

The service architecture and behavior of the Kokyu scheduling framework is illustrated in Figure 3.4. This architecture has evolved from earlier work on a CORBA scheduling service [117] that supported purely static RMS for avionics mission computing applications [41, 67, 61]. Based on this work, as well as our experience prototyping dynamic scheduling strategies, we have identified the following set of common steps shown in Figure 3.4 that are necessary to configure and process requests for a broad range of scheduling strategies:

Step 1: A CORBA application specifies QoS information and passes it to the Kokyu reconfigurable scheduler, which is implemented as a CORBA object (*i.e.*, it implements an IDL interface). The use of CORBA IDL allows applications to specify sets of values (*i.e.*, `RT_Info`s) that concisely capture the characteristics of each of its schedulable operations (*i.e.*, `RT_Operation`s), along with any data dependencies between these operations.

Step 2: At configuration time, which can occur either off-line or on-line, the application passes this QoS information into the Kokyu reconfigurable scheduler. The reconfigurable scheduler stores the QoS information in its repository of `RT_Info` descriptors. The reconfigurable scheduler then constructs operation dependency graphs based on `RT_Info`s registered with it by the application. The scheduler identifies threads of execution by examining the terminal nodes of these dependency graphs.

The scheduler can then infer information induced by the dependency graph, such as the effective periods of execution of dependent operations. Nodes that have incoming edges but no outgoing edges in the dependency graph are called *consumers*. Consumers are dispatched after the nodes on which they depend. Nodes that have outgoing edges but no incoming edges are called *suppliers*. Suppliers correspond to distinct threads of execution in the system. Nodes with incoming *and* outgoing edges can fulfill both roles.

Step 3: Next, the Kokyu scheduling framework assigns static priorities and subpriorities to operations. These values are assigned according to the specific strategy used to configure the Kokyu scheduling framework. For example, when the TAO Kokyu scheduling framework is configured with the MUF strategy, static priority is assigned according to operation criticality. Likewise, static subpriority is assigned according to operation importance and dependencies. By assigning and caching static information at configuration time, the Kokyu scheduling framework can minimize overhead and non-determinism at run-time.

Step 4: Based on the specific strategy used to configure it, the Kokyu scheduling framework divides the dispatching priority and dispatching subpriority components into statically and dynamically assigned portions. The static priority and static subpriority values are used to assign the static portions of the dispatching priority and dispatching subpriority of the operations. These dispatching priorities and subpriorities reside in TAO's `RT_Info` repository. Performing this step at configuration time helps minimize run-time overhead and non-determinism.

Step 5: In this step, the Kokyu scheduling framework assesses schedulability. A set of operations is considered *schedulable* if all critical operations will meet their deadlines. Schedulability is assessed according to whether all operations within and above the minimum critical static priority level will be able to meet their deadlines, based on the worst case simultaneous arrival of all operations, *i.e.*, the *critical instant* [70]. Operations are augmented with a dynamic subpriority based on the critical instant, and their resulting dispatching priority and dispatching subpriority are used to assess worst case feasibility of the critical operations. This *static* analysis can provide a worst-case schedulability assessment for static, dynamic, and hybrid strategies alike.

Step 6: Based on the assigned dispatching priorities, and in accordance with the specific strategy used to configure the Kokyu scheduling framework, the number and types

of dispatching queues needed to dispatch the generated schedule are assigned. For example, when the Kokyu scheduling framework is configured with the MLF strategy, there is a single queue, which uses laxity-based ordering. As in Step 3, this static information is cached in the `RT_Info` repository until it is needed at run-time to configure the Kokyu dispatching infrastructure.

Step 7: When TAO's ORB endsystems and applications are initialized at run-time, the configuration information in the `RT_Info` repository is used by the Kokyu scheduling framework's run-time scheduler component, which is collocated within an ORB endsystem. The ORB uses this reconfigurable scheduler to retrieve (1) the thread priority at which each queue dispatches operations and (2) the type of dispatching prioritization used by each queue. By encapsulating the thread priority and dispatching type information behind its output interface, the Kokyu framework decouples the *policies* for dispatching behavior from the *mechanisms* used to enforce those policies.

Step 8: In this step, a factory configures a *dispatching module* (*i.e.*, in the I/O subsystem, ORB Core, and/or Event Service), as described in Chapter 5. The dispatching module factory uses the configuration information provided by the Kokyu scheduling framework to create the correct number and types of queues and associate them with threads at the correct priorities that service the queues.

Step 9: When an operation request arrives from a client at run-time, the appropriate dispatching module must identify the dispatching queue to which the request belongs and initialize the request's dispatching subpriority. The reconfigurable scheduler component of the Kokyu scheduling framework (1) retrieves the static portions of the dispatching priority and dispatching subpriority from the `RT_Info` repository and (2) supplies them to the dispatching module. By caching static information that was computed at configuration time, TAO's strategized Kokyu scheduling framework minimizes run-time overhead and non-determinism for each operation invocation.

Step 10: If the dispatching queue where the operation request is placed was configured as a *dynamic queue* in step 8, the dynamic portions of the request's dispatching subpriority (and possibly its dispatching priority) are assigned. The queue first does this when it enqueues the request and then updates these dynamic portions only as necessary when other operations are enqueued or dequeued. By efficiently managing updates to dynamic information, TAO's dynamic queues minimize the amount of overhead they introduce.

Steps 3-6 represent the strategized portion of the scheduling framework, which varies with each distinct scheduling strategy. Steps 1-2 and 7-10 represent the fixed portion of the framework, which remains the same for all scheduling strategies. In the original scheduling framework upon which this research builds, steps 1-6 typically occurred off-line during a schedule configuration process, while steps 7-10 typically occurred on-line.

With the extensions and optimizations to the scheduling and dispatching infrastructure described in Chapters 4, 5, and 6, steps 1-6 can be performed on-line as well. Our earliest work on Kokyu's scheduling infrastructure [32]

1. introduced strategized support for hybrid static and dynamic scheduling heuristics,

2. decoupled scheduling heuristics from application characteristics and dispatching mechanisms,

3. provided middleware mechanisms for dynamic scheduling, and

4. did preliminary evaluation of infrastructure alternatives in the context of well-known scheduling heuristics.

As illustrated on the left side of Figure 3.3, Kokyu's scheduling infrastructure has since evolved into a light-weight common interface and a set of richer pluggable strategies that encapsulate details of both scheduling data structures and heuristics. Each scheduling strategy contains algorithms and data structures used to (1) select rates of operations and (2) assign operations to the dispatching priority lanes described below.

For example, if an application only had information about the periodicity of tasks and did not know in advance what periods it would need to handle, it could plug in a strategy that used comparison sorting to order tasks for priority assignment according to rate monotonic scheduling [70] (RMS). However, an application that knew all possible values for both periodicity and criticality could use a form of radix sorting to order operations for priority assignment according to RMS+LLF [15]. Supporting strategies with different data structures for different degrees of information about the operations to be scheduled allows use-case-specific optimizations to the timeliness of adaptive re-scheduling. Section 3.5.2 considers these issues in detail.

3.4 Flexible Dispatching Framework

The right side of Figure 3.3 shows the essential features of Kokyu's flexible task dispatching infrastructure. Key features of the dispatching infrastructure that are essential to performing our optimizations are as follows:

Timers: Each top-level operation in the dependency graph has an associated rate of invocation, which is implemented by associating each top-level operation with a timer at that rate.

Dispatching queues: Each task is assigned by our strategized Kokyu scheduling framework [32] to a specific dispatching queue, each of which has an associated queue number, a queueing discipline, and a unique operating-system-specific priority for its single associated dispatching thread.

Dispatching threads: Operating-system thread priorities decrease as the queue number increases, so that the 0th queue is served by the highest priority thread. Each dispatching thread removes the task from the head of its queue and runs its entry point function to completion before retrieving the next task to dispatch. As described in Section 3.5.1, adapters can be applied to operations to intercept and possibly short-circuit the entry-point upcall. In general, however, the outermost operation entry point must complete on each dispatch.

Queueing disciplines: Dispatching thread priorities determine which queue is active at any given time: the highest priority queue with a task to dispatch is always active, preempting tasks in lower priority queues. In addition, each queue may have a distinct discipline for determining which of its enqueued tasks has the highest eligibility, and must ensure the highest is at the head of the queue at the point when one is to be dequeued. We consider three disciplines:

- *Static* – Tasks are ordered by a static subpriority value – results in FIFO ordering if all static subpriorities are made the same; static queues at different priority levels can be used to implement an RMS scheduling strategy.
- *Deadline* – Tasks are ordered by time to deadline; a single deadline queue can be used to implement the earliest deadline first [70] (EDF) scheduling strategy.

- *Laxity* – Tasks are ordered by slack time, or *laxity* – the time to deadline minus the execution time; a single laxity queue can be used to implement the minimum laxity first [126] (MLF) scheduling strategy; laxity queues at different priority levels can be used to implement the maximum urgency first [126] (MUF) scheduling strategy.

Any discipline for which a maximal eligibility may be selected can be employed to manage a given dispatching queue in this approach. Scheduling strategies can be constructed from one or more queues of each discipline alone, or combinations of queues with different disciplines can be used, as in [15].

3.5 Middleware Scheduling and Dispatching Optimizations

Careful optimization of middleware is needed to meet the goals of mission-critical DRE systems described in Chapter 1. In this section we present several key optimizations that we have applied to realistic avionics mission computing applications in the target platform environment described in Section 3.1.

We adopt the definition of *system modes* used by Cross and Schmidt [20] as follows. A *mode* is a Boolean function on the states of a system's constituent configuration items. For example in the context of an avionics mission computing application, "the aircraft is landing" is a mode, and "aft sensors are at their highest rates" is a mode. The value of a mode can change abruptly. For example, the failure of a component can affect modes. In DRE systems the time allotted to respond to mode changes may be very short. In fact, this requirement is one of the key technical differences between mission-critical DRE applications and mainstream commercial business applications.

For this dissertation, we define a *mode partition* as an equivalence partition over the set of possible states of the system. Our middleware scheduling optimizations focus on two high-level mode partitions–*steady-state* and *adaptive*–of the target avionics mission computing platform. As illustrated in Figure 3.5, the steady-state mode partition contains all steady behavioral states, with a particular rate and priority assigned to each operation while in that state. Finally, we define an *operating region* based on the notion of a *QoS region* in QuO [143], as a set of states in the steady-state partition that can be reached from one another without crossing into the adaptive partition.

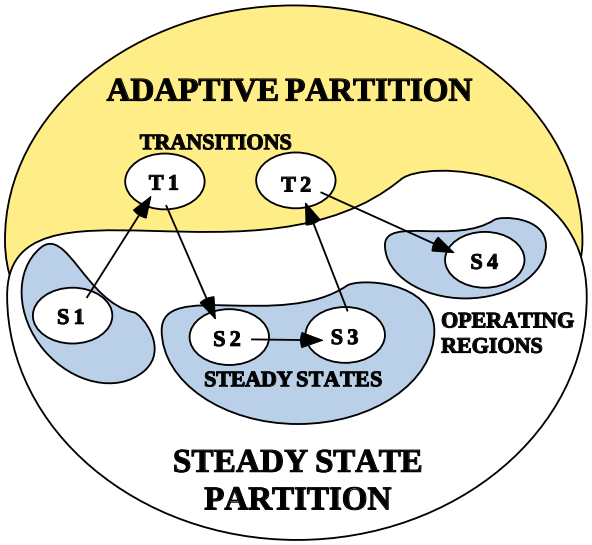

Figure 3.5: System Mode Partitions

The adaptive mode partition consists of the sequence of transitions between steady behavioral states, in which a new round of rate selection and priority assignment must be performed. Section 3.5.1 describes optimizations to the steady-state mode partition, and Section 3.5.2 describes optimizations to the adaptive mode partition. In our current research, the RTARM described in Sections 2.4 and 3.1 is invoked from the steady-state mode partition, but may transition the system into the adaptive mode partition during its execution.

3.5.1 Steady-State Optimizations

Existing research [15, 9] on adaptive scheduling of mandatory and optional operations has largely focused on properties that can be specified *a priori*, such as the computational complexity of the scheduling algorithm, the error function for optional tasks during overload, and the value to the application of completing various stages of task execution. While these approaches are valuable for establishing the essential theory of building adaptive DRE systems, we believe an empirical approach is also useful to guide design decisions and reveal opportunities for application-specific and domain-specific optimizations in middleware.

For example, hybridization of the rate monotonic scheduling (RMS), earliest deadline first (EDF), and minimum laxity first (MLF) scheduling techniques has been

proposed to isolate mandatory tasks from optional tasks, and optimize the execution behavior of those tasks [15]. The approach in this earlier work is to assign all mandatory tasks to higher priority levels using RMS, and assign all optional tasks to one or more lower priority levels using EDF, MLF, or other scheduling techniques. Other proposals suggest a similar priority partitioning of mandatory and optional tasks, but choose other combinations of scheduling techniques, such as scheduling each priority partition using MLF [126].

Clearly, a variety of scheduling approaches and hybrid combinations thereof are possible–and often desirable–for scheduling various types of DRE applications. However, choosing the approach that is best suited to a particular application or application domain requires attention not only to the characteristics and requirements of the application, but also of the platforms and middleware on which it is hosted. Here, we focus primarily on the empirically measured low-level characteristics of the dispatching infrastructure on which the scheduling policies will be enforced in our flexible scheduling framework. As shown in Section 5.4.3, the overhead associated with task dispatching differs for each of the three queueing disciplines described in Section 3.4. Static queueing incurs the lowest enqueue and dequeue overheads, followed by the deadline discipline, and the laxity discipline has the highest.

These results suggest scheduling optimizations for our target application platform, based on reducing the dispatching overhead of an intermediate criticality class and the level of confidence in the advertised execution times of operations. Since the RTARM described in Sections 2.4 and 3.1 must manage adaptive transitions whenever a change in application state requires a reallocation of rates, it must operate at a higher priority than the optional operations. However, if its operations cannot be feasibly scheduled with the mandatory operations, at least some of them must be assigned to an intermediate priority partition between the optional and mandatory operations. To meet the three system objectives described at the beginning to this chapter, we describe four types of performance optimizations for this scenario, illustrated in Figure 3.6:

A. Dynamic scheduling: If we cannot feasibly schedule all of the RTARM operations with the mandatory operations, or the combination produces a barely feasible schedule and we lack confidence in the precision of the advertised execution times, we might trade some measure of overhead for stricter partitioning between the mandatory and RTARM operations, and schedule the RTARM operations in an intermediate priority queue using a deadline- or laxity-based discipline. This optimization allows the target system some

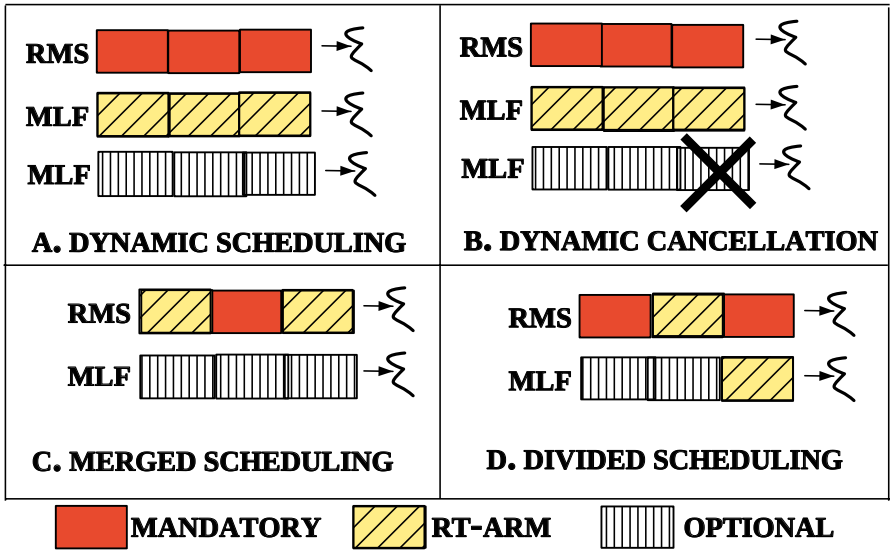

Figure 3.6: Steady-State Optimizations

flexibility to meet our goal to accommodate unplanned tasks and unexpected variations in operation characteristics (*i.e.*, some jitter in the execution times), especially of the RTARM or optional operations.

B. Dynamic cancellation: If we cannot feasibly schedule all of the operations within a priority partition, we must consider whether to allow futile dispatches of operations, even though we know they will miss their deadlines. Reducing the number of futile dispatches and wasted CPU time may improve the performance of other operations and increase either the number of made deadlines, the amount of work completed before deadlines, or both. This optimization can help meet our goal to trade performance of individual elements for overall performance objectives, *e.g.*, maximizing the availability of the CPU for operations that *can* meet their deadlines.

Cancellation adds overhead, however, so it should not be applied to mandatory partitions that are known to be feasible, especially when the benefits of optimizations, such as static dispatching, are desired. Moreover, a balance between optimism and pessimism must be achieved for cancellation to be effective. As described in Chapter 8, our initial measurements of this technique using a rather pessimistic cancellation strategy actually *reduced* the number of optional operations that made their deadlines. The figure shows more operations making their deadlines without cancellation, compared to the fewer operations making their deadlines with pessimistic cancellation. With a more accurate cancellation threshold, however, we believe the technique will give the target

system more exact control over individual operation dispatches, thereby allowing more deadlines to be met overall.

C. Merged scheduling: If we can feasibly and confidently schedule the RTARM operations and mandatory operations together using RMS, then merging the RTARM operations upward into the mandatory partition serves to reduce (1) the number of threads needed to dispatch operations and (2) the expected queueing overhead for RTARM operations. This optimization can help with our goal of improving real-time performance (*i.e.*, reducing overhead) across heterogeneous criteria (*i.e.*, criticality and rate).

D. Divided scheduling: If we can partition the RTARM operations themselves into mandatory and optional segments (*e.g.*, to consider different ranges of available rates) and the RTARM mandatory segment is feasible with the other mandatory operations, then we can merge it upward into the RMS partition, reducing overhead for at least the mandatory part of RTARM. Specifically, if we can feasibly schedule the part of the RTARM responsible for assessing the current status of the system, we avoid having to schedule additional operations later to re-establish consistency between the RTARM and the current system state. Note that we might still keep the optional RTARM tasks at a higher priority level than the other optional tasks, to prevent interference and increase quality of adaptive transition solutions.

By ensuring that the critical status assessment portion of the RTARM is feasibly scheduled, and thus avoiding consistency recovery costs, this optimization can help meet our goal of performing adaptive resource reallocations within firmly bounded timescales. This optimization can also help meet our goal to improve real-time performance across heterogeneous criteria, *i.e.*, criticality and rate or laxity, by maximizing the number of operations assigned to more efficient dispatching queues.

3.5.2 Adaptive Optimizations

This research offers a middleware-based solution in which lower-level QoS management services are leveraged where possible, or are provided in middleware when necessary. This solution also complements and provides services to higher-level COTS and custom middleware QoS management techniques from the broader research community [143, 44, 117].

End-to-End Admission Control: Mission-critical distributed real-time and embedded system requirements pose new challenges for resource allocation. For adaptive rate reconfiguration, remote dependencies require an end-to-end admission control protocol to ensure that (1) appropriate adaptation is performed on each endsystem to maintain the end-to-end timing constraints and (2) sufficient resources are feasibly reserved on each endsystem. Thus, it is essential to identify and develop policies and mechanisms for end-to-end real-time admission control that address these challenges.

For example, consider a sensor processing application with sampling operations and processing operations on different endsystems [25]. Depending on the environment and application state, sensor sampling operations may run at any one of a set of rates. Whenever the sampling portion of the application is ready to adapt by changing the rate at which it samples the sensor input, the following two activities must occur:

Reserving local resources: Any increase in the rate of execution of an operation must be validated for scheduling feasibility on its local endsystem. For example, if an operation doubles its rate of execution, it will use twice as much CPU time.

Adaptation handshaking: Remote dependencies may also complicate adaptation in distributed systems. For example, sampling at a higher rate than can be processed may be of no benefit. If so, the admission control protocol may need to negotiate the same rate for both sampling and processing operations, and ensure the operations can be scheduled feasibly at that rate on their respective endsystems.

Integrated Middleware Framework Historically, embedded real-time applications have often coupled QoS management and application logic, and provided timing assurances by relying on static architectures, such as cyclic executives. Unfortunately, these solutions can be brittle when requirements change, particularly when changes occur at run-time. Moreover, such coupling can expose application developers to accidental complexities, and thus increase the risk of insidious errors.

Encapsulating QoS management mechanisms within the Kokyu framework and allowing flexible configuration of policies shields application developers from error-prone QoS management details, and provides flexibility in meeting diverse end-to-end QoS requirements. Canonical QoS management mechanisms encapsulated by our framework include (1) QoS service configuration, (2) admission control, (3) QoS exception propagation, (4) QoS exception handling, (5) pacing, (6) shaping, and (7) classification.

Integrating mechanisms within the Kokyu framework offers improved adaptive real-time performance. For example, consider two mechanisms: (1) *classifying* operations for dispatch priority assignment, and (2) rate selection for adaptive *admission control*. If the possible rates of all operations are known at admission control time, priority assignment can be performed at the same time as rate selection, as illustrated in Figure 3.7:

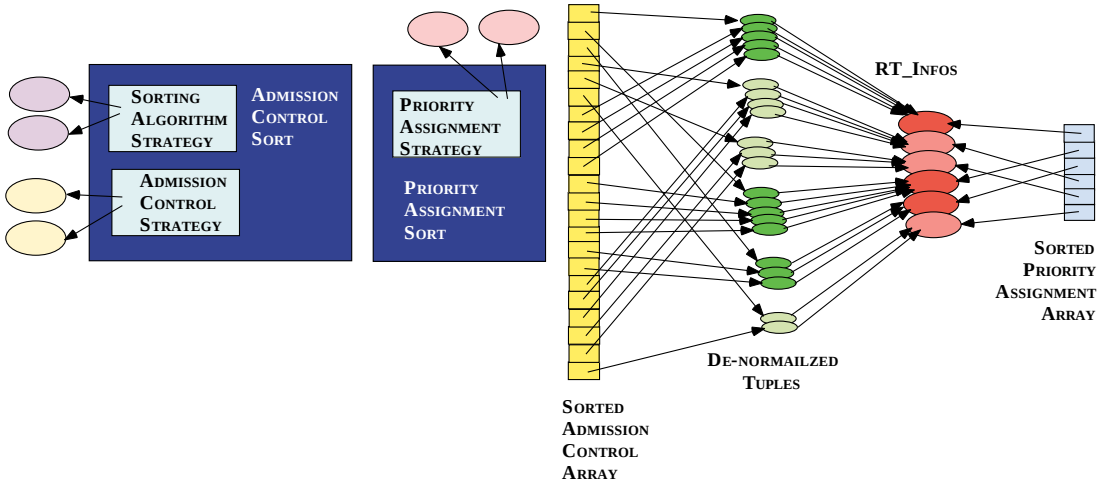

Figure 3.7: Integrated Framework Architecture

We integrate classification and rate selection in Kokyu as follows:

Operation Characteristics: We store information about operation characteristics (*e.g.*, period, criticality, and worst-case, average, and best-case execution times), in a `RT_Info` descriptor. The application, or a higher level management layer, stores the values for the characteristics of each operation in its `RT_Info` descriptor.

Denormalized Tuples: In adaptive systems, multiple possible values may be specified for some operation characteristics. The Kokyu framework must select a value for each such characteristic. Priority assignment may need to be performed as well because priority assignment may depend on value selection (*e.g.*, RMS [70] depends on selected rates).

To reduce the computational complexity of admission control, it is useful to perform both selection of values for operation characteristics and priority assignment in the same pass. This can be done by (1) de-normalizing the `RT_Infos` and the sets of

possible values into a sequence of tuples, (2) sorting the tuples according to both the priority assignment and admission control policies, and (3) performing value selection and priority assignment on the sorted sequence.

Composing Strategies: By using a *stable* sorting technique, or by composing admission control and priority assignment comparisons, the constraints of both policies can be met, assuming they are not contradictory. Moreover, it may be possible to apply increasingly efficient sorting algorithms depending on the information known about the operations at admission control time. For example, if all the rates are known in advance, it may be possible to apply an $O(n)$ algorithm, *e.g.*, radix sort. Otherwise, an $O(nlogn)$ comparison sort, *e.g.*, heap sort, is needed.

3.6 Improving RTARM and Scheduler Interaction

In our prior adaptive scheduling research [25], a previous-generation RTARM [44] interacted with a previous-generation instance of our scheduler via its *sensitivity interface*. This interface allowed the RTARM to

1. Propose a specific assignment of rates to operations
2. Obtain a boolean feasibility assessment for that assignment and
3. Obtain a number representing the sensitivity of that feasibility result to increases or decreases in the rates assigned to the operations.

The RTARM performed these steps whenever a transition between steady states was needed. Two scheduling thresholds were defined for each scheduling strategy: one for mandatory operations that corresponded to the achievable utilization bound under that scheduling strategy, and one for mandatory and optional operations together that corresponded to a higher (slightly overscheduled) total scheduling bound. A particular assignment of rates to the set of operations was considered feasible if all mandatory operations could be assured to be schedulable feasibly, as in [15].

To perform its assignment algorithm, the RTARM iteratively extended a set of rate-to-operation bindings, adding new bindings and updating existing ones based on responses from the scheduler to feasibility and sensitivity queries. The individual performances of the RTARM and the scheduler sensitivity implementation were reasonable, as shown in Figures 6.2 and 6.3 in Section 6.2. Both

1. the number of calls to the sensitivity interface, and

2. the amount of time spent assessing feasibility and sensitivity within each operation,

were roughly proportional to the number of operations in the schedule.

The combined behavior of the RTARM and scheduler was not as good as we might hope, however, since the product of the number of calls and the time per call produces an overall performance curve that is quadratic in the number of operations. Therefore, we apply the following refinements to optimize the combined behavior, illustrated in Figure 3.8, and described in greater detail in Chapter 6:

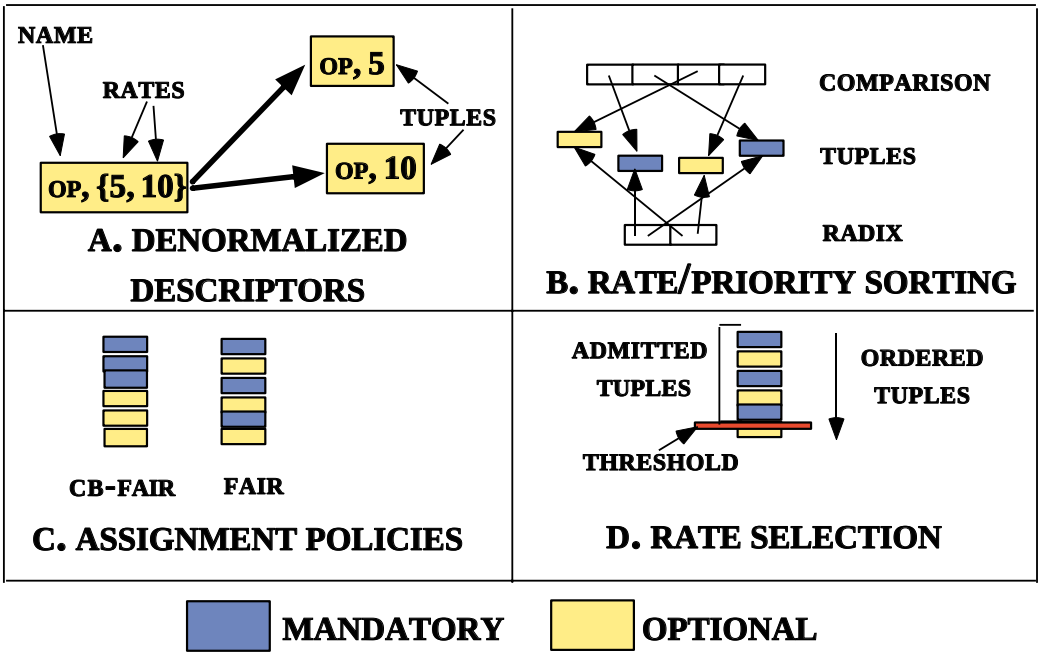

Figure 3.8: Adaptive Optimizations

A. De-normalized operation descriptors: We de-normalize the available rate set and fixed characteristics for each operation into a sequence of flat tuples of characteristics (containing *e.g.*, the operation handle, a particular rate, the execution time at that rate).

B. Rate and priority sorting: We recast rate and priority assignment as a sorting problem over operation characteristics, with at worst an $O(nlog(n))$ bound on worst-case performance, and an $O(n)$ bound on worst-case performance in certain special instances of the more general problem.

C. Assignment policies: We encapsulate specific sort ordering strategies as policies for rate assignment, much as we have done previously for scheduling policies [32].

D. Rate Selection: Once the tuples are sorted, we perform a single $O(n)$ traversal of the tuples to select the rate of each operation and determine expected utilization values based on the rates selected and the advertised execution times.

3.7 Metrics Framework

Finally, Kokyu provides an integrated metrics framework, which Chapter 7 describes in detail. We have optimized this framework for use in shared memory, so that platforms such as the VME-connected embedded boards described in Section 8.1.3 can collect and exchange metrics data for performance evaluation and adaptive feedback, within bounded space and time. The metrics infrastructure provides the following:

- A shared-memory-capable metrics cache, that uses explicit type casting and based smart-pointers to support run-time sizing and placement within shared memory segments mapped at arbitrary process offsets, as Section 7.1 describes.
- Inline instrumentation and monitoring classes, including a shared-memory-capable timeprobe class and an upcall adapter that can be used for adaptive feedback, cancellation, or both, as described in Section 7.1.
- Integration with Higher Level Resource Managers such as QuO and RTARM, also described in Section 7.1.
- A time frame manager, described in Section 7.2, that keeps accurate track of the current frame start and end, and also a frame counter at each rate to support adaptive transition management of the kind described in Section 9.3.2.
- Integration with Remote Logging and Visualization Services, as described in Section 7.3.

Chapter 4

Kokyu Scheduling Framework Implementation

Many hard real-time systems, such as those for avionics mission computing and manufacturing process controllers, have traditionally been scheduled statically using Rate Monotonic Scheduling (RMS) [56]. Static scheduling provides schedulability assurance prior to run-time and can be implemented with low run-time overhead [117]. However, static scheduling has the following disadvantages:

Inefficient handling of non-periodic processing: Static scheduling treats aperiodic processing as if it was periodic (*i.e.*, occurring at its maximum possible rate). Resources are allocated to aperiodic operations either directly or through a sporadic server[1] to reduce latency. In typical operation, however, aperiodic processing may not occur at its maximum possible rate. One example is interrupts, which potentially may occur very frequently, but often do not.

Unfortunately, with purely static scheduling, resources must be allocated pessimistically and scheduled under the assumption that interrupts occur at the maximum rate. When they do not, utilization is effectively reduced because unused resources cannot be reallocated.

Utilization phasing penalty for non-harmonic periods: In statically scheduled systems, achievable utilization can be reduced if the periods of all operations are *not* related harmonically. Operations are related harmonically if their periods are integral multiples

[1] A sporadic server [66] reserves a portion of the schedule to allocate to aperiodic events when they arrive.

of one another. When periods are not harmonic, the phasing of the operations produces unscheduled gaps of time. This reduces the maximum schedulable percentage of the CPU (*i.e.*, the *schedulable bound*), to below unity. The *utilization phasing penalty* is the difference between the value of the schedulable bound and 100%.

Liu and Layland established a least upper utilization bound of $n(2^{1/n} - 1)$ [70] for a set of operations to be statically schedulable, where n is the number of distinct non-harmonic operation periods in the system. Recent research has shown that this bound may be overly pessimistic in some cases [39]. However, the fact remains that with static priority assignment *some* unschedulable gaps may be created by non-harmonic periods.

Inflexible handling of invocation-to-invocation variation in resource requirements: Because priorities cannot be changed easily[2] at run-time, allocations must be based on worst-case assumptions. Thus, if an operation usually requires 5 msec of CPU time, but under certain conditions requires 8 msec, static scheduling analysis must assume that 8 msec will be required for every invocation. Again, utilization is effectively penalized because the resource will be idle for 3 msec in the usual case.

Impact of situational factors on resource requirements: Recent advances in static priority analysis [80, 38] have shown that the schedulable bound for statically prioritized operations can be improved dramatically in some cases. These techniques rely, however, on advance knowledge of (1) arrival patterns of operation dispatch requests and (2) sequences of operation execution times. In many distributed real-time applications, such as those for avionics mission computing and image processing, variation in load on the system is largely due to *situational* factors. Thus, such detailed information may not be available accurately prior to run-time.

In general, static scheduling limits the ability of real-time systems to adapt to changing conditions and changing configurations. In addition, static scheduling provides resource access guarantees at the cost of lower resource utilization. To overcome the limitations of static scheduling, therefore, we have investigated the use of dynamic strategies to schedule CORBA operations for applications with real-time QoS requirements.

[2]Priorities can be changed via *mode changes* [117], but that is too coarse to capture invocation-to-invocation variations in the resource requirements of complex applications.

4.1 Overcoming Static Scheduling Limitations with Dynamic Scheduling

Other scheduling algorithms provide some relief from the limitations of RMS. For instance, Earliest Deadline First (EDF) scheduling assigns higher priorities to operations with closer deadlines. EDF is commonly used for dynamic scheduling because it permits run-time modification of rates and priorities. In contrast, static techniques like RMS require fixed rates and priorities.

Dynamic scheduling offers a way to address the drawbacks of static scheduling described above. In particular, dynamic scheduling strategies offer optimal utilization capabilities [70] and handle invocation-to-invocation variations in execution times efficiently. If these drawbacks can be alleviated without incurring excessive overhead or non-determinism, dynamic scheduling can be beneficial for real-time applications with deterministic QoS requirements.

Demanding real-time applications, such as avionics mission computing, cannot tolerate unnecessary overhead and non-determinism at run-time. Therefore, we restrict our attention in this dissertation to scheduling approaches that do not perform frequent schedulability analysis at run-time. In particular, we do not consider strategies that require run-time admission control for dynamic scheduling of each operation. Rather, we only consider scheduling strategies where it is possible to select the set of operations critical to the application *statically*. Among such strategies, we are most interested in those whose *dynamic* run-time behavior allows greater resource utilization. Unfortunately, many dynamic scheduling strategies do not offer the *a priori* guarantees of static scheduling. For instance, purely dynamically scheduled systems (*i.e.*, those without any form of admission control for dynamically generated operations) can behave non-deterministically under heavy loads. Thus, operations that are critical to an application may miss their deadlines because they were (1) delayed by non-critical operations or (2) delayed by an excessive number of critical operations.

Hybrid static and dynamic approaches may be used to combine the benefits of both. The remainder of this section reviews several strategies for dynamic and hybrid static/dynamic scheduling, using the terminology defined in Appendix A. These scheduling strategies include purely dynamic techniques, such as EDF and Minimum Laxity First (MLF), as well as the hybrid Maximum Urgency First (MUF) strategy.

4.1.1 Purely Dynamic Scheduling Strategies

This section briefly reviews two of the well known purely dynamic scheduling strategies, EDF [70, 56], and MLF [126]. These strategies are illustrated in Figure 4.1 and discussed below. Figure 4.1 also shows the hybrid static/dynamic MUF [126] schedul-

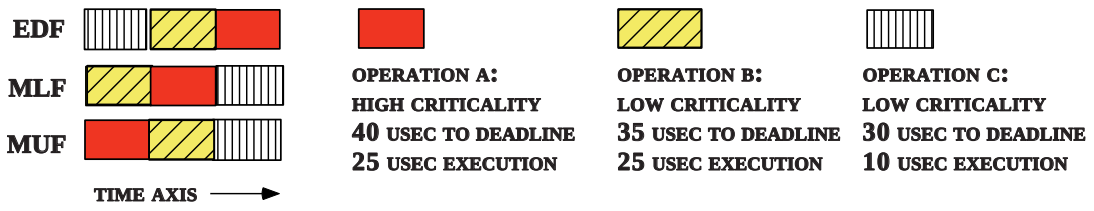

Figure 4.1: Dynamic Scheduling Strategies

ing strategy discussed in Section 4.1.2.

Earliest Deadline First (EDF): EDF [70, 56] is a dynamic scheduling strategy that orders dispatches[3] of operations based on time-to-deadline, as shown in Figure 4.1. Operation executions with closer deadlines are dispatched before those with more distant deadlines. The EDF scheduling strategy is invoked whenever a dispatch of an operation is requested. Depending on the mapping of priority components into thread priorities, the new dispatch may or may not preempt the currently executing operation, as discussed in Section 5.2.

A key limitation of EDF is that an operation with the earliest deadline is dispatched, whether or not there is sufficient time remaining to complete its execution prior to the deadline. Therefore, the fact that an operation cannot meet its deadline will not be detected until *after* the deadline has passed. Moreover, that operation will continue to consume CPU time that could otherwise be allocated to other operations that could still meet their deadlines.

Minimum Laxity First (MLF): MLF [126] refines the EDF strategy by taking into account operation execution time. It dispatches an operation whose *laxity* is least, as shown in Figure 4.1. Laxity is defined as the time-to-deadline minus the remaining execution time.

[3] A *dispatch* is a particular execution of an *operation*.

Using MLF, it is possible to detect that an operation will not meet its deadline *prior* to the deadline itself. If this occurs, a scheduler can reevaluate the operation before allocating the CPU, as discussed for the MUF scheduling strategy in Section 4.1.2.

Evaluation of EDF and MLF:

• **Advantages:** From a scheduling perspective, the main advantage of EDF and MLF is that they overcome the utilization limitations of RMS when rates are non-harmonic. In particular, the utilization phasing penalty described at the beginning of this chapter cannot occur with EDF and MLF, because they assign priorities based on run-time characteristics. In addition, EDF and MLF handle harmonic and non-harmonic periods comparably. Moreover, they respond flexibly to invocation-to-invocation variations in resource requirements, allowing CPU time unused by one operation to be reallocated to other operations. Thus, they can produce schedules that are optimal in terms of CPU utilization [70]. Finally, both EDF and MLF can dispatch operations within a single static priority level [70, 126], which is useful for non-preemptive single-threaded environments.

• **Disadvantages:** From a performance perspective, a disadvantage of purely dynamic scheduling approaches like MLF and EDF is that their scheduling strategies require higher overhead to evaluate at run-time. In addition, these purely dynamic scheduling strategies offer no control over *which* operations will miss their deadlines if the schedulable bound is exceeded. As operations are added to the schedule to achieve higher utilization, the margin of safety for *all* operations decreases. As the system becomes overloaded, therefore, the risk of missing a deadline may increase for every operation.

4.1.2 Maximum Urgency First (MUF)

The MUF [126] scheduling strategy supports the deterministic rigor of the static RMS scheduling approach *and* the flexibility of dynamic scheduling approaches like EDF and MLF. MUF is the default scheduler for the Chimera real-time operating system [127]. We support a variant of MUF in the Kokyu scheduling framework, which is presented in more detail in Section 4.2. MUF can assign both static *and* dynamic priority components. In contrast, RMS assigns all priority components statically based on fixed rates

and EDF/MLF assign all priority components dynamically based on deadlines/laxities. The hybrid priority assignment in MUF overcomes the drawbacks of the individual scheduling strategies by combining techniques from each, as shown in Table 4.1 and described below:

Table 4.1: MUF Priority Components

Component	**Precedence**	**Assignment**	**Type**	**Basis**
Criticality	Highest	Static	Binary	*Application Defined*
Dynamic Subpriority	Intermediate	Dynamic	Float	1/laxity
Static Subpriority	Lowest	Static	*Application-Defined*	

Criticality: In MUF, operations with higher *criticality* are assigned to higher static priority levels. Assigning static priorities according to criticality prevents operations critical to the application from being preempted by non-critical operations. In general, MUF allows any number of criticality values. For the empirically studied avionics mission computing application described in Chapter 8, however, we restrict our attention to the case with two values: *critical* or *non-critical*. This restriction appears to be reasonable for many real-time applications with deterministic QoS requirements, although

1. the Kokyu scheduling and dispatching framework readily supports multiple criticality levels, and

2. crafting policies and mechanisms with more than two criticality levels may be motivated by particular use cases.

By separating resource demands of critical and non-critical operations, MUF provides greater control over which operations miss deadlines during overload conditions.

To assign criticalities, Stewart and Khosla [126] suggest initially ordering operations by rate. Ordering operations by rate reduces the risk that non-critical operations will miss their deadlines. Next, all tasks whose executions fall entirely within 100% CPU utilization are designated as *critical* and all other tasks *non-critical*. [126] also describes a variant of MUF criticality assignment that relaxes ordering by rate to allow any ordering of operations. By relaxing this requirement, applications can select which

operations are critical and which are not. Ordering operations by application-defined criticality reflects a subtle and fundamental shift in the notion of priority assignment. In particular, RMS, EDF, and MLF exhibit rigid mappings from *empirical* operation characteristics to a *single* priority value.

Moreover, EDF and MLF offer little or no control over which operations will miss their deadlines under overload conditions. In contrast, MUF affords applications the ability to distinguish operations arbitrarily, giving them *explicit* control over *which* operations will miss their deadlines under conditions of overload. Therefore, it can protect a critical *subset* of the entire set of operations. This fundamental shift in the notion of priority assignment leads to the generalization of scheduling techniques discussed in Section 4.2.

Dynamic Subpriority: Dynamic subpriority is used to differentiate two operations that have the same criticality. In MUF, *dynamic subpriority* has lower precedence than criticality. Therefore, dynamic subpriority is the primary basis for scheduling operations within a single static priority level at run-time.

An operation's dynamic subpriority is evaluated whenever it is enqueued in or dequeued from a dynamically ordered dispatching queue. At the instant of evaluation, dynamic subpriority in MUF is a function of the laxity of an operation.

An example of such a simple dynamic subpriority function is the inverse of the operation's laxity.[4] Operations with the smallest positive laxities have the highest dynamic subpriorities, followed by operations with higher positive laxities, followed by operations with the most negative laxities, followed by operations with negative laxities closer to zero. Assigning dynamic subpriority in this way provides a consistent ordering of operations as they move through the *pending* and *late* dispatching queues, as described below.

By assigning dynamic subpriorities according to laxity, MUF offers higher utilization of the CPU than the statically scheduled strategies. MUF also allows deadline failures to be detected *before* they actually occur, except when an operation that would otherwise meet its deadline is preempted by a higher criticality operation. Moreover, MUF can apply various types of error handling policies when deadlines are missed [126].

[4]To avoid division-by-zero errors, any operation whose laxity is in the range $\pm\epsilon$ can be assigned (negative) dynamic subpriority $-1/\epsilon$ where ϵ is the smallest positive floating point number that is distinguishable from zero. Thus, when the laxity of an operation reaches ϵ, it is considered to have missed its deadline.

For example, if an operation has negative laxity prior to being dispatched, it can be diverted from the dispatching queue. This allows operations that can still meet their deadlines to be dispatched instead. This can be achieved by ordering operations by dynamic subpriority, and refreshing this order whenever an operation enqueue or dequeue request is made. This effectively partitions the queue into two sections, with pending dispatches ahead of late dispatches.

• **Pending dispatches:** Operations that can meet their deadlines are dispatched preferentially until there are no more operations that can meet their deadlines. At this point, operations that missed or will miss their deadlines are dispatched. This strategy has the beneficial effect that operations able to meet their deadlines will not be delayed by operations unable to meet their deadlines.

• **Late dispatches:** Operations that missed their deadlines are dispatched in the order of their now negative dynamic subpriorities. This order is the same as that produced by their original positive dynamic subpriorities. Operations that reached negative laxity first are dispatched ahead of operations that reached negative laxity later.

When this strategy is used in the *late* dispatch queue, it preserves the order that operations had in the *pending* dispatch queue. In addition, it reduces overhead in the dispatching mechanism. MUF's ability to detect failures early and to specify policies for error handling provides applications with greater control over the *consequences* of scheduling failures, even when they occur for non-critical operations.

Static Subpriority: In MUF, *static subpriority* is a static, application-specific, optional value. It is used to order the dispatches of operations that have the same criticality and the same dynamic subpriority. Thus, static subpriority has lower precedence than either criticality or dynamic subpriority. Assigning a unique static subpriority allows a total dispatch ordering of operations at run-time. For a given arrival pattern of operation requests, the total ordering ensures that the dispatch order will always be identical. This assurance improves system predictability, reliability, and testability.

The variant of MUF provided by the Kokyu scheduling framework enforces a total dispatch ordering by providing an `importance` field in the TAO `RT_Info` CORBA operation QoS descriptor [117], which is described in Appendix A. Kokyu uses `importance`, as well as a topological ordering of operations, to assign a unique static subpriority for each operation within a given criticality level. Incidentally, the original definition of MUF in [126] uses the terms *dynamic priority* and *user priority*,

whereas we use the terms *dynamic subpriority* and *static subpriority* for Kokyu, respectively. We selected different terminology to indicate the subordination to static priority. These terms are interchangeable when referring to the MUF strategy, however.

4.2 Design Goals of the Kokyu Scheduling Framework

To alleviate the limitations with existing scheduling strategies described at the beginning of this chapter, our real-time scheduling research focuses on developing a CORBA-based framework that enables applications to

1. maximize total utilization,
2. preserve scheduling guarantees for critical operations, and
3. adapt flexibly to different application and platform characteristics.

These goals are illustrated in Figure 4.2 and summarized below:

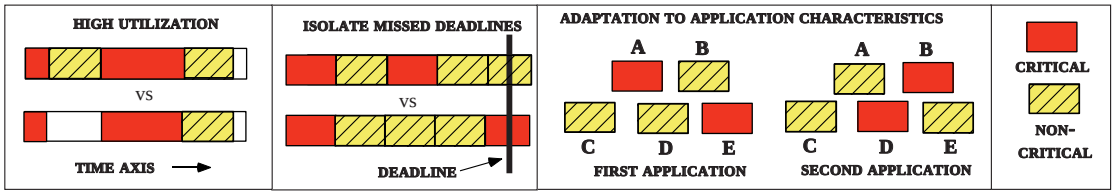

Figure 4.2: Design Goals of the Kokyu Scheduling Framework

Goal 1 – Higher utilization: The leftmost pair of timelines in Figure 4.2 demonstrates our first research goal: *higher utilization*. This timeline shows a case where a critical operation execution did not, in fact, use its worst-case execution time. With dynamic scheduling, an additional non-critical operation could be dispatched, thereby achieving higher resource utilization and more importantly increasing the overall useful output of the system.

Goal 2 – Preserving scheduling guarantees: The next pair of timelines in Figure 4.2 demonstrates our second research goal: *preserving scheduling guarantees for critical operations*. In the lower timeline, priority is based only on traditional scheduling parameters, such as rate and laxity. In the upper timeline, criticality is also included. Both

timelines depict schedule overrun. When criticality is considered, only non-critical operations miss their deadlines.

Goal 3 – Adaptive scheduling: The sets of operation blocks on the right in Figure 4.2 demonstrate our third research goal: *providing applications with the flexibility to adapt to varying application requirements and platform features*. In this example, two applications (or, two distinct modes of a single application) use the same five operations. However, the first considers operations A and E critical, whereas the second considers operations B and D critical. By allowing applications (or modes) to select which operations are critical, it is possible to provide scheduling behavior that is appropriate to each application's individual requirements.

These three goals motivate the design of the Kokyu scheduling framework. For the real-time systems [41, 117, 60, 118, 61] to which TAO has been applied, it has been possible to identify a core set of operations whose execution before deadlines is *critical* to the integrity of the system. Therefore, the Kokyu scheduling framework is designed to ensure that critical CORBA operations will meet their deadlines, even when the total utilization exceeds the schedulable bound.

If it is possible to ensure missed deadlines will be isolated to non-critical operations, then adding non-critical operations to the schedule to increase total CPU utilization will not increase the risk of missing critical deadlines. The risk will only increase for those operations whose execution prior to deadline is *not* critical to the integrity of the system. In this way, the risk to the whole system is minimized when it is loaded for higher utilization.

4.2.1 Kokyu Scheduling Input Interface

Real-time applications must specify their QoS information to their selected scheduling strategy. The scheduling strategy then uses this application-specific QoS information to ensure that the QoS received by the application conforms to this information. The key design issues for QoS specification, and how TAO's strategized scheduling framework addresses them, are as follows:

Decoupling QoS specification and strategy details: Although the application must specify its QoS information to the instantiated scheduling strategy, it is essential that it not tightly couple the application to any specific scheduling strategy. This flexibly

allows the scheduling strategy to be varied independently of the application-specific QoS information. Thus, changing the scheduling strategy need not require changes to the application.

TAO's scheduling strategy framework is designed to minimize unnecessary constraints on the values that application developers specify to the input interface described in Section 4.2.1. For instance, one (non-recommended) way to implement the RMS, EDF, and MLF strategies in TAO's scheduling service framework would be to implement them as variants of the MUF strategy. This can be done by manipulating the values of the operation characteristics [126]. However, this approach would tightly couple applications to the MUF scheduling strategy and the strategy being emulated.

There is a significant drawback to tightly coupling the behavior of a scheduling service to the characteristics of application operations. In particular, if the value of one operation characteristic used by an application changes, developers must remember to manually modify other operation characteristics specified to the scheduling service in order to preserve the same mapping. In general, TAO's scheduling service framework shields application developers from such unnecessary details.

Defining a fixed input interface: TAO's scheduling service framework decouples QoS specification from any specific scheduling strategy by providing a *fixed* Common Object Request Broker Architecture (CORBA) Interface Definition Language (IDL) *input interface*. All scheduling strategy details are hidden behind this interface. To achieve this encapsulation, TAO's scheduling service framework allows applications to specify the entire set of possible operation characteristics using this fixed input interface.

As illustrated in steps 1 and 2 of Figure 3.4, applications use TAO's scheduling service input interface to convey QoS information. The scheduling strategy then uses this information to prioritize operations. TAO's scheduling service input interface consists of theCORBA IDL interface operations shown in Figure 4.3 and described below:

create(): This operation takes a string with the operation name as an input parameter. It creates a new `RT_Info` descriptor for that operation name and returns a handle for that descriptor to the caller. If an `RT_Info` descriptor for that operation name already exists, `create` raises the DUPLICATE_NAME exception.

add_dependency(): This operation takes two `RT_Info` descriptor handles as input parameters. It places a dependency on the second handle's operation in the first handle's `RT_Info` descriptor. This dependency informs the scheduler that a flow of control

```
interface  Scheduler
{
    // . . .

    // Create a new RT_Info descriptor for entry_point
    handle_t create ( in string entry_point )
        raises ( DUPLICATE_NAME );


    // Add dependency to handle's RT_Info descriptor
    void add_dependency ( in handle_t handle,
                          in handle_t dependency  )
        raises ( UNKNOWN_TASK );


    // Set values of operation characteristics
    // in handle's RT_Info descriptor
    void set ( in handle_t handle,
               in Criticality criticality,
               in Time worstcase_exec_time,
               in Period_period,
               in Importance importance  )
        raises ( UNKNOWN_TASK );

    // . . .
}
```

Figure 4.3: Kokyu Scheduling Framework IDL Input Interface

passes from the second operation to the first. If either of the handles refers to an invalid `RT_Info` descriptor, `add_dependency` raises the UNKNOWN_TASK exception.

set(): This operation takes an `RT_Info` descriptor handle and values for several operation characteristics as input parameters. The `set` operation assigns the the passed input values to the corresponding operation characteristics in the `RT_Info` descriptor. If the passed handle refers to an invalid `RT_Info` descriptor, `set` raises the UNKNOWN_TASK exception.

4.2.2 Kokyu Scheduling Output Interface

An ORB must obtain QoS enforcement information generated by the scheduling strategy. This information is then used by an ORB to enforce the QoS specified by the scheduling strategy. The key design issues for QoS enforcement, and how the Kokyu scheduling framework addresses them, are as follows:

Decoupling QoS enforcement and strategy details: While an ORB must obtain QoS enforcement information generated by the instantiated scheduling strategy, it must not

tightly couple the ORB to any specific scheduling strategy. This allows the scheduling strategy to be varied independently from the ORB, so that changing the scheduling strategy does not require any changes to the ORB.

Defining a fixed output interface: the Kokyu scheduling framework decouples QoS enforcement from any specific scheduling strategy by providing a *fixed* CORBA IDL *output interface*, behind which the scheduling strategy details are hidden. As illustrated in steps 7 through 10 of Figure 3.4, the ORB uses the Kokyu scheduling output interface to obtain QoS enforcement information and configure its dispatching modules accordingly. The output interface for Kokyu's scheduling framework consists of the CORBA IDL interface operations shown in Figure 4.4 and described below:

```
interface  Scheduler
{
    // . . .

    // Get configuration information for the queue that will dispatch all
    // RT_Operations that are assigned dispatching priority d_priority
    void dispatch_configuration ( in Dispatching_Priority d_priority,
                                  out OS_Priority os_priority,
                                  out Dispatching_Type d_type )
        raises ( UNKNOWN_DISPATCH_PRIORITY,
                 NOT_SCHEDULED );


    // Get static dispatching subpriority and dispatching
    //  priority assigned to the handle's RT_Operation
    void priority ( in handle_t handle,
                    out Dispatching_Subpriority d_subpriority,
                    out Dispatching_Priority d_priority)
        raises ( UNKNOWN_TASK,
                 NOT_SCHEDULED );

    // . . .
}
```

Figure 4.4: Kokyu Scheduling Framework IDL Output Interface

dispatch_configuration(): This operation provides configuration information for the queues in the dispatching modules used by the ORB endsystem (step 7 of Figure 3.4). It takes a dispatching priority value as an input parameter. It returns the OS thread priority and dispatching type corresponding to that dispatching priority level. At run-time, the Kokyu scheduler retrieves these values from its efficient `RT_Info` repository, where they were stored in step 6 of Figure 3.4. The `dispatch_configuration` operation

will raise the UNKNOWN_DISPATCH_PRIORITY exception if it is passed a dispatching priority parameter that is not in the schedule. Likewise, if a schedule has not been generated, the `dispatch_configuration` operation raises the NOT_SCHEDULED exception.

priority(): This operation provides dispatching priority and dispatching subpriority information for an operation request (step 9 of Figure 3.4). It takes an `RT_Info` descriptor handle as an input parameter and returns the assigned dispatching subpriority and dispatching priority as output parameters. The Kokyu reconfigurable scheduler retrieves the dispatching priority and dispatching subpriority values stored in the `RT_Info` repository in step 4 of Figure 3.4). If the passed handle does not refer to a valid `RT_Info` descriptor, `priority` raises the UNKNOWN_TASK exception. If a schedule has not been generated, `priority` raises the NOT_SCHEDULED exception.

4.3 Input Mappings Implemented in Kokyu

Each scheduling strategy must provide a platform-independent assignment of priority values to each operation. This allows an ORB to leverage commonality among scheduling strategies, while preserving appropriate variations among them. The key design issues for platform-independent priority assignment, and how the Kokyu scheduling framework resolves them, are as follows:

Decoupling priority from OS-specific mechanisms: It is important that the priorities and subpriorities assigned by the scheduling strategies remain independent of specific priority enforcement capabilities of the underlying OS platform, at least at some level. This allows the same scheduling strategy to be implemented, with only minor modifications, on platforms with diverse priority enforcement capabilities.

Defining a platform-independent input mapping: Kokyu decouples priority from the specific priority enforcement capabilities of the underlying OS platform by providing a separate, platform-independent, level of priority assignment. Kokyu's scheduling framework leverages the commonality among these mappings to make its implementation more uniform. The variations between these mappings provide hooks for adaptation to the requirements of specific applications. Furthermore, the Kokyu scheduling framework simplifies development and experimentation with *new* scheduling strategies within TAO's standards-compliant real-time CORBA middleware framework.

Input mappings for MUF, MLF, EDF, and RMS have been implemented in the Kokyu scheduling framework, as described below. In each mapping, static subpriority is assigned first using importance and second using a topological ordering based on dependencies. The canonical definitions of MLF, EDF, and RMS do not include a minimal static ordering. Adding it to Kokyu's strategy implementations for these strategies has no adverse effect, however. This is because MLF, EDF, and RMS require that *all* operations are guaranteed to meet their deadlines for the schedule to be feasible, under *any* ordering of operations with otherwise identical priorities. Moreover, static ordering has the benefit of ensuring determinism for each possible assignment of urgency values.

Defining a platform-independent mapping for MUF: Kokyu provides a platform-independent mapping from operation characteristics onto urgency for MUF as shown in Figure 4.5(A). Static priority is assigned according to criticality in this mapping. There

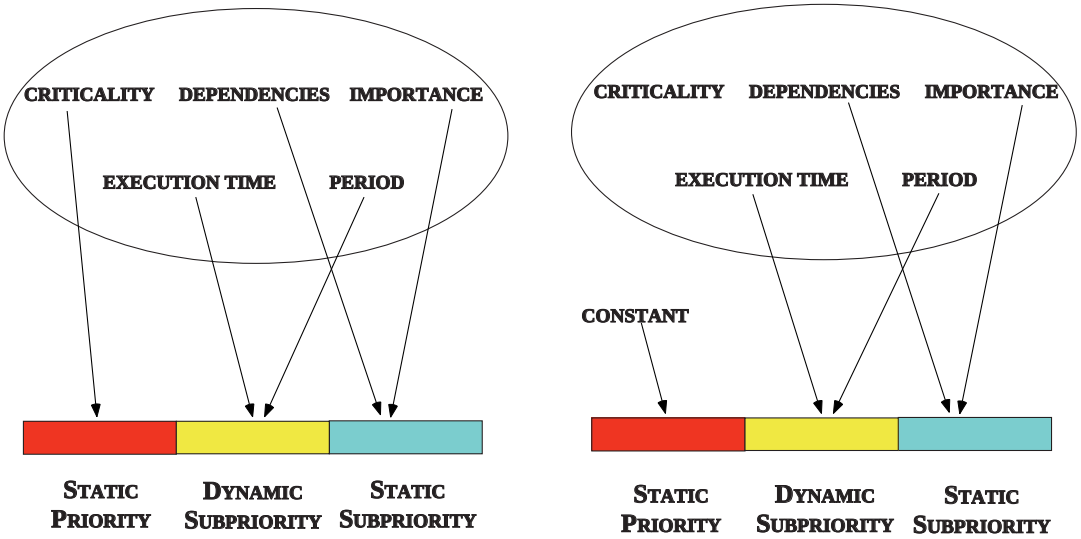

Figure 4.5: Input Mappings: (A) MUF (B) MLF

are only two static priorities since we use only two criticality levels in Kokyu's MUF implementation. The critical set in this version of MUF is the set of operations that were assigned the *high* criticality value.

When MUF is implemented with only two criticality levels, the minimum critical priority is the static priority corresponding to the high criticality value. In the more general version of MUF [126], where multiple criticality levels are possible, the critical set may span multiple criticality levels.

Dynamic subpriority is assigned in the MUF input mapping according to *laxity*. Laxity is a function of the operation's period, execution time, arrival time, and the time of evaluation.

Defining a platform-independent mapping for MLF: Kokyu provides a platform-independent mapping from operation characteristics onto urgency for MLF as shown in Figure 4.5(B). The mapping for MLF assigns a constant (zero) value to the static priority of each operation. This results in a single static priority. The minimum critical priority is this lone static priority. The MLF strategy assigns the dynamic subpriority of each operation according to its laxity.

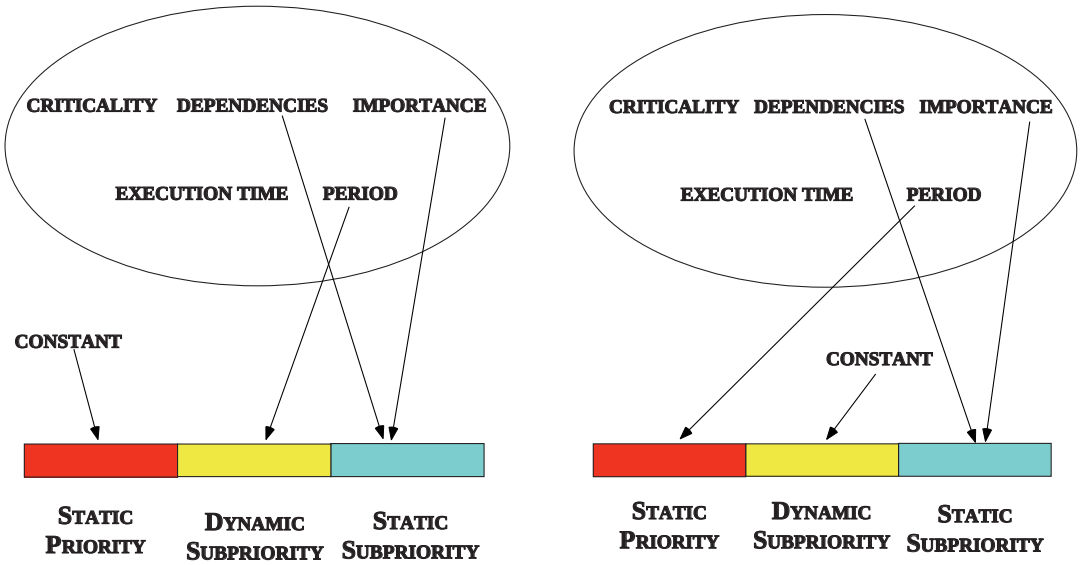

Figure 4.6: Input Mappings: (A) EDF (B) RMS

Defining a platform-independent mapping for EDF: Kokyu provides a platform-independent mapping from operation characteristics onto urgency for EDF as shown in Figure 4.6(A). Like the MLF mapping, the EDF mapping also assigns a zero value to the static priority of each operation. Moreover, the EDF strategy assigns the dynamic subpriority of each operation according to its *time-to-deadline*, which is a function of its period, its arrival time, and the time of evaluation.

Defining a platform-independent mapping for RMS: Kokyu provides a platform-independent mapping from operation characteristics onto urgency for RMS as shown in Figure 4.6(B). The RMS mapping assigns the static priority of each operation according to its *period*, with higher static priority for each shorter period. The period for aperiodic

execution must be assumed to be the worst case. In RMS, all operations are critical, so the minimum critical priority is the minimum static priority in the system. The RMS strategy assigns a constant (zero) value to the dynamic subpriority of each operation.

Defining a platform-independent mapping for RMS+MLF: In addition to strategies for MUF, RMS, EDF, and MLF, the RMS+MLF [15] strategy has been implemented in the Kokyu scheduling framework. The input mapping for the RMS+MLF strategy is shown in Figure 4.7. In this strategy, the user-supplied criticality of each

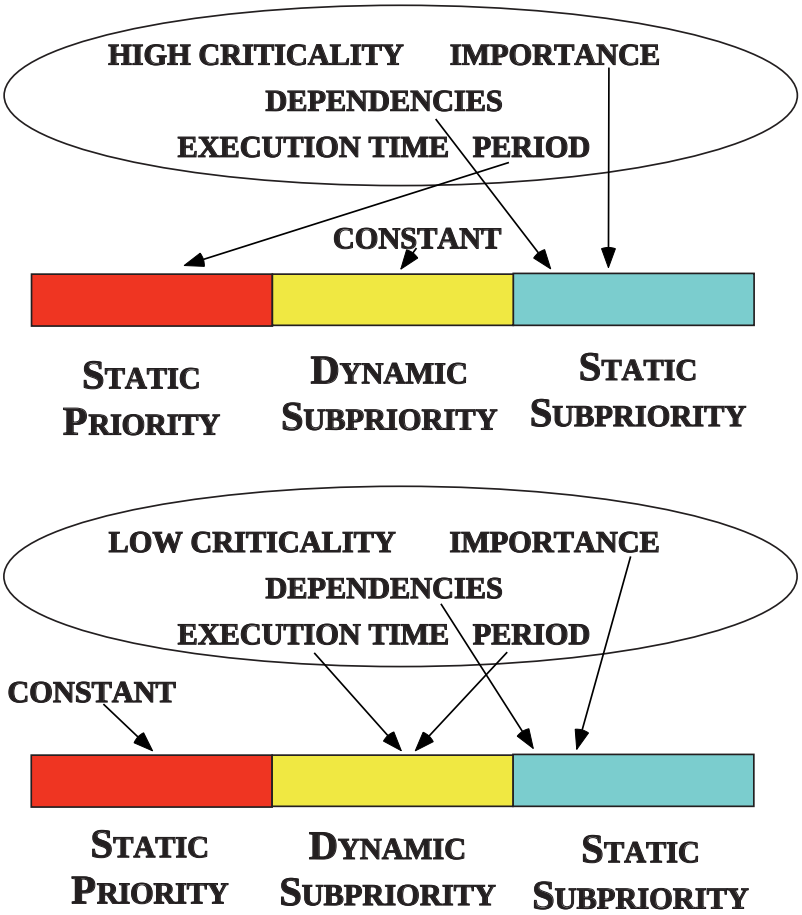

Figure 4.7: RMS+MLF Input Mapping

operation is used to distinguish between operations that are in the critical set and those that are not. The complete input mapping for the RMS+MLF is partitioned according to these sets:

• **RMS+MLF critical set:** The critical set is mapped according to the mapping used in the RMS strategy. Decreasing static priority is assigned according increasing period for operations in the critical set. Dynamic subpriority is not used for operations in the critical set, and is assigned 0. Static subpriority is assigned as in all the other strategies, according to importance and then according to dependency ordering, to operations in the critical set.

• **RMS+MLF non-critical set:** The non-critical set is mapped according to the mapping used in the MLF strategy. An additional static priority, lower than all static priorities assigned to operations in the critical set, is assigned to all operations in the non-critical set. Dynamic priority is is assigned to operations in the non-critical set, according to laxity. Static subpriority is assigned to operations in the non-critical set as it was for operations in the critical set.

4.4 Output Mappings Implemented in Kokyu

Each scheduling strategy assigns platform-independent urgency values to operations, which must then be used to dispatch operations using each endsystem's OS-specific dispatching model. The key design issues for platform-specific dispatching, and how the Kokyu scheduling framework resolves them, are as follows:

Enforcing priority through platform-specific dispatching: The input mappings described in Section 4.3 specify priorities and subpriorities for operations. However, there is no mechanism to enforce these priorities, independent of the platform-specific dispatching models. Therefore, each scheduling strategy must provide a mapping from platform-independent urgency values into platform-dependent dispatching priorities and subpriorities.

Defining platform-specific values for Kokyu's dispatching modules: As described in Chapter 5, operations are distributed to priority dispatching queues in the ORB according to their assigned dispatching priority. Operations are ordered within priority dispatching queues according to their designated dispatching subpriority. The scheduling strategy's output mapping assigns dispatching priority and dispatching subpriority to operations as a function of the urgency values specified by the scheduling strategy's input mapping. In each of Kokyu's scheduling strategies, an output mapping transforms the platform-independent priority and subpriority values into dispatching priority and

subpriority requirements that can be enforced by the specific dispatching models in real systems. Figure 4.8 illustrates the output mapping used by the scheduling strategies implemented in Kokyu. Each part of the mapping is described below.

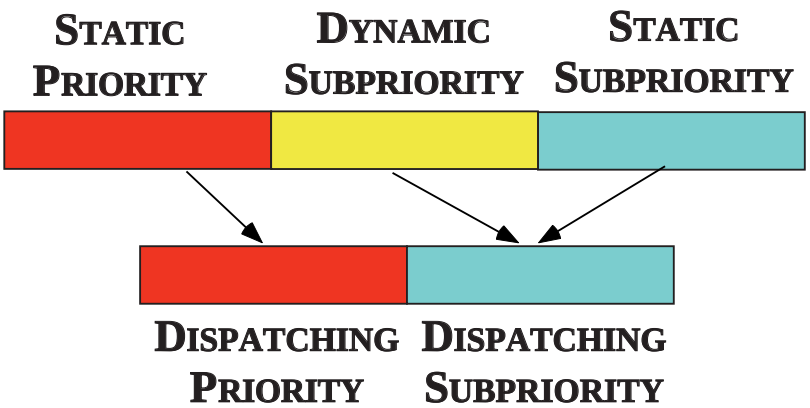

Figure 4.8: Output Mapping Implemented in Kokyu

• **Dispatching priority:** In this mapping, static priority maps directly to dispatching priority. This mapping corresponds to the priority band dispatching model described in Section 4.4. Each unique static priority assigned by the input mapping results in a distinct thread priority in Kokyu's dispatching modules, which are described in Chapter 5. Thus, an operation with higher static priority will always preempt one with lower static priority.

• **Dispatching subpriority:** Dynamic subpriority and static subpriority map to dispatching subpriority. The Kokyu scheduling framework performs this mapping efficiently at run-time by transforming both dynamic and static subpriorities into a binary representation.

Because the range of dynamic subpriority values and the number of static subpriorities are known prior to run-time, a fixed number of bits can be reserved for each. Dynamic subpriority is stored in the m highest order bits, where $m = \lceil \lg(ds) \rceil$, and ds is the number of possible dynamic subpriorities. Static subpriority is stored in the next n lower order bits, where $n = \lceil \lg(ss) \rceil$, and ss is the number of static subpriorities.

Kokyu's preemption subpriority mapping scheme preserves the ordering of operation dispatches according to their assigned *urgency* values. Operations with the same static priority are ordered first by dynamic subpriority and second by static subpriority.

4.5 Operation Dependency Graph

The TAO Scheduling Service was enhanced with a reconfigurable scheduler implementation. Previously, all schedule computation was performed off-line, and the static results used to configure a static run-time scheduler. Adaptive scheduling capabilities were supported only at the level of dispatching queues, which could be configured to use either dynamic or static queue ordering. The reconfigurable scheduler implementation allows the same scheduler to be used for schedule computation and dispatching priority assignment, all at run-time. In this section we describe several enhancements to the previous generation static scheduler.

Graph Algorithms: While the previous generation scheduler implemented a number of the same algorithms, it was structured for recursive traversal of the dependency graph. As noted in Section 3.3, the reconfigurable scheduler constructs operation dependency graphs based on `RT_Info`s registered with it by the application. Nodes that have outgoing edges but no incoming edges in the dependency graph are called *consumers*. Consumers are dispatched after the nodes on which they depend. Nodes that have incoming edges but no outgoing edges are called *suppliers*. Suppliers correspond to distinct threads of execution in the system. Nodes with incoming *and* outgoing edges can fulfill both roles. The reconfigurable scheduler identifies threads of execution by examining the terminal nodes of these dependency graphs. The reconfigurable scheduler can then infer information induced by the dependency graph, such as the effective periods of execution of dependent operations.

Modular Functionality: The requirements for diverse additional algorithms drove the Kokyu framework design toward a model of iterative traversal by visitors over the graph. Furthermore, explicit representation of graph algorithm preconditions and postconditions as separate methods in the visitors allowed efficient re-use of methods across an inheritance hierarchy. Functions for priority assignment, operation sorting, and other algorithm steps are performed via one or more traversals across the operation descriptors in the graph. Functions can be composed, so that a single pass can accomplish several logical algorithmic steps in a single visit to an operation descriptor. This allows the scheduler implementation to be extended in a modular way, while preserving its performance characteristics.

Efficient Data Structures: While the previous To achieve reasonable run-time performance, the reconfigurable scheduler uses highly efficient data structures, notably Hash Maps and Red-Black Trees. A Hash Map performs O(1) storage and lookup of operations given a unique key, which in this case is the `RT_Info` descriptor handle. A Red-Black Tree is used to hold names of operations, though names are rarely used once an operation is registered and its handle returned. As future work, this could be optimized for the special case where all names are known in advance, and a perfect hashing generator [114] could be used with a Hash Map to provide O(1) lookup of operations by name as well as by handle.

Scheduling as Strategized Sorting: If we have prior knowledge of all possible values of key operation characteristics used by a particular scheduling heuristic, *e.g.*, rate and criticality, then we can provide optimizations of the scheduling infrastructure. In particular, we note the ability to assign priorities through different forms of sorting, and apply the most efficient sorting algorithm possible for each case. As future work, C++ generic programming mechanisms [8] such as templates and traits, used as in the C++ Standard Template Library [125], could be applied to the Kokyu scheduling framework to *automatically* associate the most efficient algorithm with a particular set of operation characteristics.

Radix vs. Comparison Sorting: When values of operation characteristics *are* known in advance, O(n) radix sorting can be achieved by creating a hashing function, inserting the operations into a Hash Map parameterized with the hash function, and then simply iterating across the buckets in the Hash Map to obtain the ordering. For example, the rates of invocation for operations in the experimental application discussed in Chapter 8 were 40 Hz, 20 Hz, 10 Hz, 5 Hz, and 1 Hz. A simple C++ hash function, $y = (x/5 > 0)\ ?\ log_2(x/5) + 1 : 0$; would convert those rates into hash table indices 4, 3, 2, 1, and 0 respectively, providing a hash implementation of RMS priority ordering. A similar function could be constructed for RMS+MLF, and the hash function for MUF is simply the criticality value.

4.6 Simulating Critical Instant Behavior

As described in Section 4.2, two of our research goals are (1) to increase effective CPU utilization while (2) preserving scheduling guarantees for critical operations. We first

use simulation to examine the extent to which these goals can be met under conditions of overload. This section presents the results of a simulation that visualizes the behavior of canonical scheduling strategies under overload conditions. Our focus is on the *critical instant*, which occurs in a preemptive schedule when all operation requests arrive simultaneously [70].

In real-time systems, the distribution of when operation requests arrive is important. For example, a given set of operations may be feasibly schedulable if requests for the operations are distributed evenly across a given time frame, but cannot all be scheduled if all requests arrive simultaneously.[5] In order to ensure that a set of operations is schedulable under any pattern of requests, a scheduling strategy must be able to manage the critical instant. Simulating our strategized scheduling service framework's behavior after the critical instant illustrates how it performs for a given set of periodic operations under a worst-case request dispatching scenario. The remainder of this section:

1. describes the simulation design,
2. compares simulation results for the elements of different scheduling strategies in terms of latency, laxity, and missed deadlines, and
3. presents conclusions supported by the simulation results.

These simulation results indicate the feasibility of achieving our research goals and motivate our empirical studies described in Chapter 8.

4.6.1 Simulation Design

We instrumented an early version of the Kokyu scheduling framework to generate timelines for the dispatching and preemption order of the operations after the critical instant. To characterize this behavior, operation dispatches were simulated over a one second time frame, from the critical instant. Each simulation was run until the last operation finished executing.

For each scheduling strategy, the simulator took the priority output of the scheduler and used it to construct a complete preemption timeline. The simulator assumed all operations were elegible to run at the critical instant, and then used the period field

[5] This is called a *critical instant*, which is the worst case request pattern [70].

of each operation to determine when subsequent requests for each operation would be eligible.

To present a fair comparison of canonical scheduling strategies (*i.e.*, MUF [126], MLF [126, 71], EDF [70], and RMS [70]) and provide insight into other strategies composed of them (*e.g.*, RMS+MLF [15]), our simulation employs a *preemptive-by-urgency* dispatching model, as discussed in Section 5.2. This model always executes the highest eligibility operation that is ready to execute at a given time, preempting any lower eligibility operation already running when a higher eligibility operation arrives. Strategies like EDF and MLF, which rely entirely on dynamic prioritization of operations, would otherwise exhibit a disproportional number of priority inversions. Moreover, the canonical definition of EDF [70] specifies that it is dispatched in a fully preemptive manner.

In our simulations, we used a set of operations spanning a range of criticality and period values. The combined utilization of these operations exceeded the maximum schedulable bound, which is the maximum percentage of the CPU that can be utilized, while the combined utilization by critical operations was below the maximum bound. Table 4.2 summarizes the characteristics of each operation in the simulation.

Table 4.2: Characteristics of Simulated Operations

operation	period Hz	worst-case execution time, msec	Criticality	Importance
“low_1”	1	18	LOW	HIGH
“low_5”	5	18	LOW	HIGH
“low_10”	10	18	LOW	HIGH
“low_20”	20	18	LOW	HIGH
“high_1”	1	18	HIGH	LOW
“high_5”	5	18	HIGH	LOW
“high_10”	10	18	HIGH	LOW
“high_20”	20	18	HIGH	LOW

Each scheduling strategy emphasizes different static and dynamic operation characteristics. Our simulations were designed to examine the effects of simple variations in operation characteristics on the scheduling behavior of the various strategies. We have varied only those parameters necessary to demonstrate meaningful differences between the strategies, while holding the others constant. In particular, we do not vary the worst-case execution times of the operations because the variations in period already produce

variations in laxity and time-to-deadline. To avoid unnecessary complexity in experimental parameters, all operations possessed the same execution time: 18 milliseconds.

The latency and laxity of each operation dispatch were calculated from the simulation timelines. Operations with negative laxity at the time they were dispatched were marked as having missed their deadlines. Operations with shorter periods had more dispatches over the frame. To compare operations that execute at different rates, values for average latency and the fraction of deadlines missed were calculated for each operation.

Under these conditions, the results for the RMS+MLF strategy were identical to those for the MUF strategy. Therefore, the figures omit the latency, laxity, and missed deadlines plots for the RMS+MLF strategy. The simulation results and conclusions we draw for the MUF strategy apply equally to the RMS+MLF strategy, albeit RMS+MLF and MUF perform differently with variation in the execution times of operations, as we examine in detail in the empirical studies in Chapter 8.

4.6.2 Comparing Operation Latency in the Scheduling Strategies

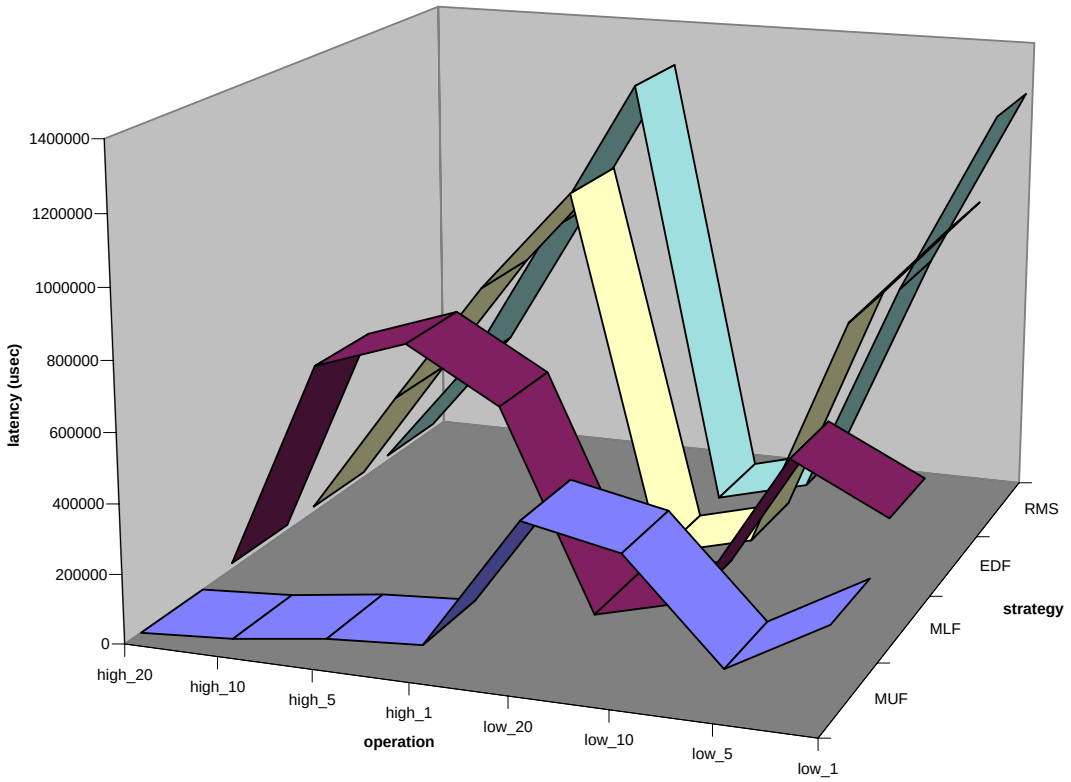

Figure 4.9: Latency of Operations for each Strategy

Figure 4.9 depicts the average latency values for the operations using each of the scheduling strategies in the simulation. Only the MUF strategy minimized the latency of critical operations, as shown in the left half of the figure. In addition, MUF detected which operations will fail to meet their deadlines. This resulted in an overall decrease in both latency and laxity of operations that could meet their deadlines in overloaded conditions.

In contrast, the other scheduling strategies did not fare as well. RMS minimized the latency of operations with shorter periods, while increasing the latency of operations with longer periods. EDF behaved similarly since time-to-deadline is a function of an operation's period. MLF also minimized the latency of operations with shorter periods, but detected which operations would fail to meet their deadlines, thereby showing better overall latency than RMS or EDF.

Upward spikes in the latency graph in Figure 4.9 show which operations incurred high average latency under each strategy. Where MLF, EDF, and RMS showed latency spikes for both critical and non-critical operations, MUF (and RMS+MLF) showed a latency spike only in the non-critical set. Maximum average laxity was lowest for MUF, (RMS+MLF,) and MLF, which considered both the worst-case execution time and time-to-deadline. Maximum average laxity was higher for EDF, which only considered time-to-deadline. It was higher still for RMS, which did not consider any dynamic characteristics.

4.6.3 Comparing Operation Laxity in the Strategies

The laxity of an operation is defined as its time-to-deadline minus its remaining execution time. Figure 4.10 shows the average laxity values for the operations for each scheduling strategy. As with Figure 4.9, only the MUF strategy protected the set of critical operations. The other strategies had negative average laxities for the critical operations with rates less than 20 Hz.

Operations that have negative laxity when they complete execution have missed their deadlines. Conversely, operations that have non-negative laxity when they complete their execution have met their deadlines. Another way to visualize the operation behavior with respect to laxity is to plot the fraction of all dispatches of an operation that miss their respective deadline. Figure 4.11 depicts this graph for the simulated operations and strategies.

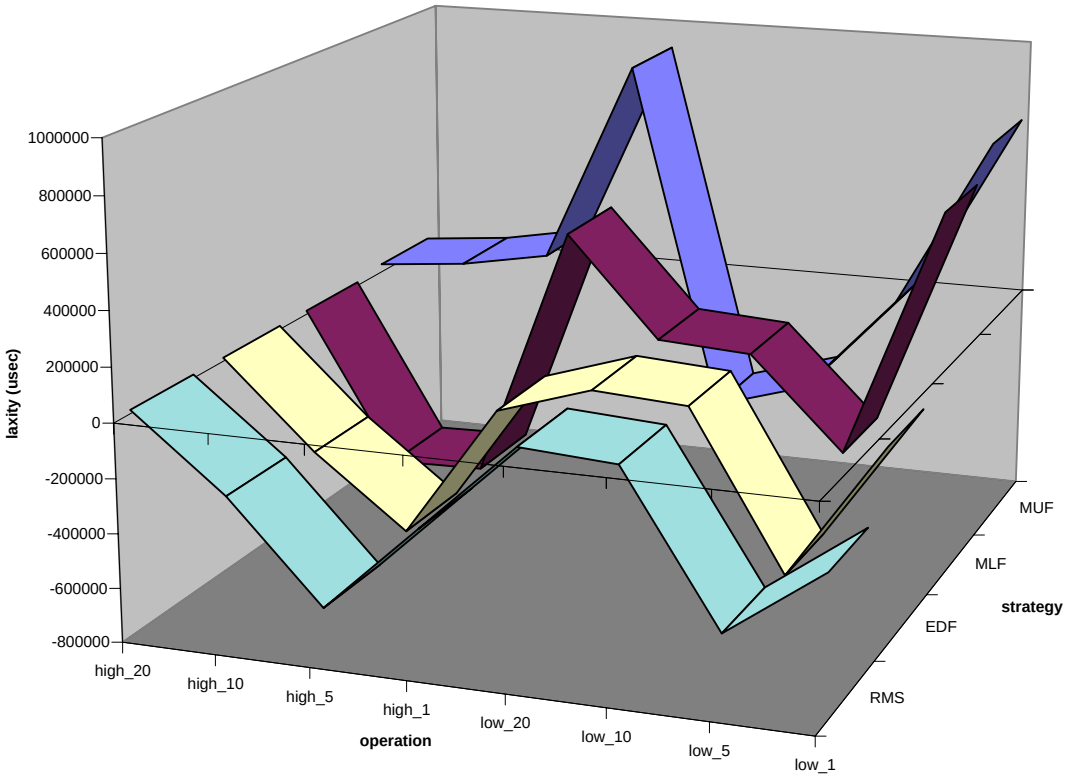

Figure 4.10: Laxity of Operations for each Strategy

The MUF (and RMS+MLF) strategy prevented the critical operations from missing their deadlines. It did so at a cost of missed deadlines in the non-critical set. However, MUF minimized the overall percentage of missed deadlines better than the other strategies.

The other strategies missed deadlines for the critical operations with rates less than 20 Hz. The MUF and MLF strategies detected scheduling failures prior to deadline. They preempted operations with negative laxity in favor of operations with positive laxity, and thus allowed more operations to meet their deadlines.

4.6.4 Analysis of Simulation Results

Our simulation results illustrate that the characteristics considered by each scheduling strategy significantly affects operation latency, laxity, and percentage of deadlines missed. These results, grouped by the operation characteristic, are summarized below:

Criticality: Under conditions of overload, only the MUF strategy reduced latency and preserved the deadline guarantees for operations in the critical set. The MUF and

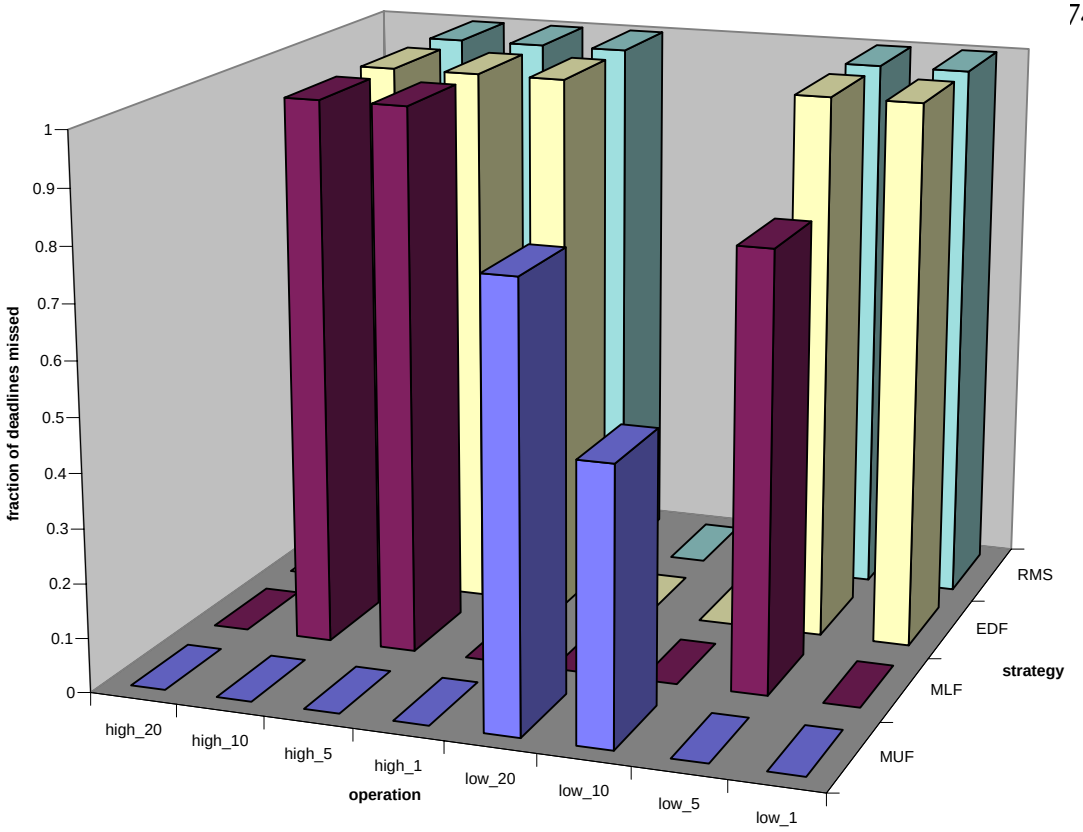

Figure 4.11: Fraction of Deadlines Missed for each Strategy

RMS+MLF strategies consider operation criticality in assigning priority, so operations in the critical set make their deadlines in preference to non-critical operations in MUF and RMS+MLF. The EDF, MLF, and RMS strategies do not consider criticality when assigning priority. Neither do they preserve deadline guarantees for operations in the critical set under conditions of overload.

Execution time: The MUF and MLF strategies, which consider time-to-deadline and worst-case execution time, reduced the impact of scheduling failures on other operations by detecting failure prior to deadline. In addition, they showed lower average latency per-operation than the other scheduling strategies.

Period: All strategies consider operation period. When all other factors are equal, each strategy shows differences in missed deadlines for operations with different periods. Among the non-critical operations in the MUF strategy simulation, the low criticality, 20 Hz period operation has lower initial laxity, because it has a closer deadline. However, is also more likely to miss its deadline as a result of preemption by critical operations. The MUF, MLF, and EDF strategies, which consider time-to-deadline, show

lower maximum and overall latency than the RMS strategy, which does not consider any dynamic operation characteristics.

Importance: Operations with higher criticality values were given lower importance values. Thus, for strategies that do not consider criticality, operations with higher importance values had fewer missed deadlines, all other factors being equal.

4.6.5 Conclusions from Simulation Experiments

The following conclusions can be drawn from comparing the results for the scheduling strategies used in the simulation:

Characteristics considered: Varying which operation characteristics a scheduling strategy considers has a significant impact on scheduling behavior. For example, only MUF considers operation criticality, and thus only MUF can selectively protect critical operations from missed deadlines.

Combinations of characteristics: Considering certain *combinations* of operation characteristics, may have an additional impact. For instance, MUF and MLF consider execution time in combination with period, which gives them the ability to detect deadline failures early and reallocate resources.

Breadth of characteristics: Strategies that consider more of the available information about static and dynamic operation characteristics generally exhibit an advantage over strategies that use less information. For example, MUF considers criticality, execution time, and period, and shows

1. lower latency,
2. fewer missed deadlines, and
3. no missed deadlines for critical operations.

This is in contrast to RMS, MLF, and EDF, each of which consider fewer operation characteristics and fails to meet at least one of these criteria, particularly in conditions of overload.

Chapter 5

Kokyu Dispatching Framework Implementation

5.1 A More General Dispatching Infrastructure

The ability to manage quality of service (QoS) dynamically in distributed real-time systems requires several enhancements to current systems including

1. flexible “modeless” execution,
2. support for variable period tasks,
3. dynamic adjustment to varying loads over longer time scales, and
4. rapid local adaptation to variable system resource availability.

To realize these benefits, systems must provide greater flexibility to support reconfiguration and adaptation, both in the application framework and in the underlying middleware.

5.2 Alternative Dispatching Models

The scheduling strategies implemented in the Kokyu framework strike a balance between preemption granularity and run-time overhead. This design is appropriate for the hard real-time avionics applications we have developed. However, it is important to consider the consequences of the specific output mapping described in Section 4.4 to

evaluate the uses and implications of alternative output mappings. Key design issues, and how the Kokyu strategized scheduling framework addresses them, are as follows:

Adapting to alternative OS dispatching models: The Kokyu scheduling architecture is designed to adapt to the needs of a range of applications, not just hard real-time avionics systems. Different types of applications and platforms may require different resolutions of key design forces.

For example, an application may run on an OS platform that *does not* support preemptive multi-threading. Likewise, other platforms do not support thread preemption and multiple thread priority levels. In such cases, the Kokyu scheduling framework assigns all operations the same constant dispatching priority and maps the entire urgency tuple directly into the dispatching subpriority [126]. This mapping correctly assigns dispatching priorities and dispatching subpriorities for a non-preemptive dispatching model. On a platform without preemptive multi-threading, the application could thus dispatch all operations in a single thread of execution, from a single priority queue.

Another application might run on a platform that *does* support preemptive multi-threading and a large number of distinct thread priorities. Where thread preemption and a very large number of thread priorities are supported, one alternative is a dispatching model that is preemptive by *urgency*. This design may incur higher run-time overhead, but can allow finer preemption granularity. The application in this second example might accept the additional time and space overhead needed to preemptively dispatch operations by urgency, in exchange for reducing the amount of priority inversion incurred by the dispatching module.

Depending on (1) whether the OS supports thread preemption, (2) the number of distinct thread priorities supported, and (3) the preemption granularity desired by the application, several dispatching models can be supported by the output interface of the Kokyu scheduling framework. Below, we examine three canonical variations supported by the Kokyu framework, which are illustrated in Figure 5.1 and described below:

- **Preemptive-by-urgency:** The output mapping currently implemented in the Kokyu Framework only supports preemption *between* static priority levels. Thus, a newly arrived operation will not be dispatched until the operation executing currently at its same preemption priority level has run to completion, even if the new operation has greater urgency. By assigning dispatching priority according to urgency, all scheduling strategies can be made fully preemptive, modulo OS dispatch latency overhead [51].

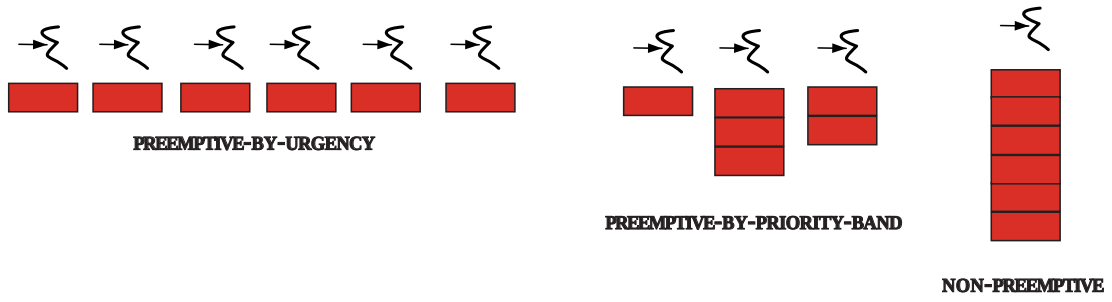

Figure 5.1: Dispatching Models Supported in the Kokyu Framework

This model incurs greater complexity in the dispatching module implementation, including locking during enqueue and dequeue operations, which in turn increases run-time overhead.

This dispatching model maintains the invariant that the highest urgency operation that is able to execute is executing at any given instant, modulo the OS dispatch latency overhead [51]. This model can be implemented only on platforms that (1) support fully preemptive multitasking and (2) provide at least as many distinct real-time thread priorities as the number of distinct operation urgencies possible in the application.

The preemptive-by-urgency dispatching model can achieve very fine-grained control over priority inversions incurred by the dispatching modules. This design potentially reduces the time bound of an inversion to the thread context switch overhead, plus any switching overhead introduced by the dispatching mechanism itself. Preemptive-by-urgency achieves its precision at the cost of increased time and space overhead, however. Although this overhead can be reduced for applications whose operations are known in advance, using techniques like perfect hashing [114], overhead from additional context switches will still be incurred.

• **Preemptive-by-priority-band:** This model divides the range of all possible urgencies into fixed priority bands. It is similar to the non-preemptive dispatching model used by message queues in the UNIX System V STREAMS I/O subsystem [107, 60]. This dispatching model maintains a slightly weaker invariant than the preemptive-by-urgency model: at any given instant, an operation from the highest fixed-priority band that has operations able to execute is executing.

This dispatching model requires thread preemption and at least a small number of distinct thread priority levels. These features are now present in many operating systems.

The preemptive-by-priority-band model is a reasonable choice when it is desirable or necessary to restrain the number of distinct preemption levels. For example, a dynamic scheduling strategy can produce a large number of distinct urgency values. These values must be constrained on operating systems, such as Windows NT [109], that support only a small range of distinct thread priorities. Operations in the queue are ordered by a subpriority function based on urgency. The strategies implemented in the Kokyu scheduling framework use a form of this model, described in more detail in Section 5.3.

- **Non-preemptive:** This model uses a single priority queue and arriving requests do not preempt a running request. It maintains a still weaker invariant: the operation executing at any instant had the greatest urgency at the time of last dispatch. As before, operations are ordered according to their urgency within the single dispatching queue. Unlike the previous models, this model can be used on platforms that lack preemptive multi-threading.

5.3 Selected Dispatching Model

Applications can benefit from strategized scheduling at a variety of points along an end-to-end request-response path. Kokyu's dispatching modules can be integrated at a number of different points within an ORB endsystem architecture. This section (1) motivates the key design issues, (2) shows how the dispatching modules fit within TAO's ORB endsystem architecture, (3) describes the internal queueing mechanism of the Kokyu dispatching modules, and (4) discusses the issue of run-time control over dispatching priority within these dispatching modules.

Supporting Adaptation Locally and End-to-End: As noted in Section 5.1, one of our key research challenges is to implement dispatching modules that support adaptation to varying loads, to maintain end-to-end QoS assurances and overall performance. By designing dispatching modules that can enforce dispatching priority and dispatching subpriority of operations at arbitrary points in the TAO ORB endsystem architecture, we have increased TAO's ability to adapt to varying QoS enforcement capabilities of the endsystem OS platforms along an end-to-end request-response path.

Supporting alternative dispatching module configurations: The output interface of the Kokyu scheduling service is designed to work with dispatching modules in one or more layers in the TAO ORB endsystem architecture. For example, TAO's real-time

extensions to the CORBA Event Service [41] use the scheduler output interface, as does its I/O subsystem [60]. Figure 5.2(A) illustrates a configuration where a dispatching module resides in TAO's real-time Event Service [41]. In Figure 5.2(A), the client ap-

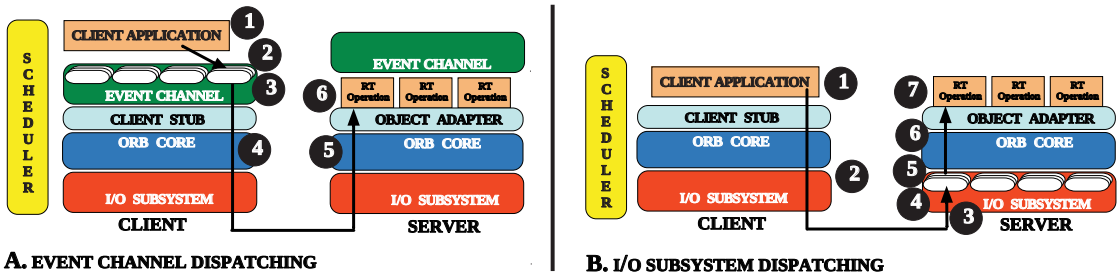

Figure 5.2: Alternative Placement of Dispatching Modules

plication **(1)** pushes an event to TAO's Event Service. The Event Service's dispatching module **(2)** enqueues events and **(3)** dispatches them according to dispatching priority and then dispatching subpriority. Each dispatched event results in **(4)** a flow of control down through the ORB layers on the client and **(5)** back up through the ORB layers on the server, where **(6)** the operation is dispatched.

Figure 5.2(B) illustrates an alternative configuration where a dispatching module would reside in TAO's I/O subsystem [60]. The client application **(1)** makes direct operation calls to the ORB, which **(2)** passes requests down through the ORB layers on the client and **(3)** back up to the I/O subsystem layer on the server. The I/O subsystem's dispatching module **(4)** enqueues operation requests and **(5)** dispatches them according to their dispatching priority and dispatching subpriority, respectively. Each dispatched operation request results in **(6)** a flow of control up through the higher ORB layers on the server, where **(7)** the operation is dispatched.

Figure 5.2 illustrate two alternatives for configuring a dispatching module within a TAO ORB endsystem. However, Kokyu supports other configurations, as well. For example, a TAO ORB endsystem can be configured with dispatching modules in *both* the I/O subsystem and the Event Service. This configuration, which might be implemented in conjunction with TAO's Real-Time CORBA features [102], helps avoid priority inversions and allows early dynamic queue management on server ORB endsystems via *early demultiplexing* [60] of events to prioritized threads in the I/O subsystem. Likewise, it

helps avoid priority inversions on client ORB endsystems via prioritized dispatching of events in a collocated client-side Event Service.

Internal architecture of a Kokyu dispatching module: Figure 5.3 illustrates the general queueing mechanism used by the dispatching modules in the Kokyu dispatching framework. In addition, this figure shows how the output information provided by the

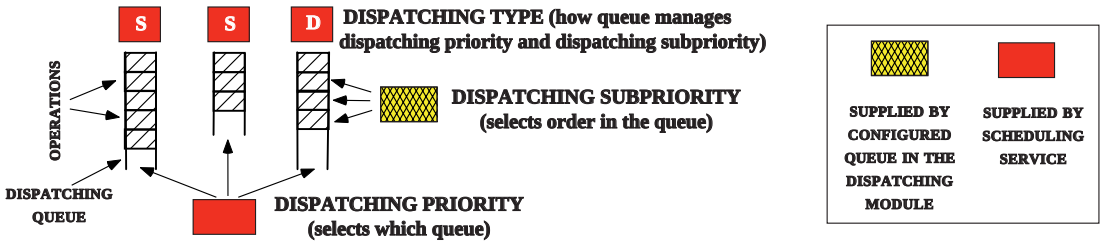

Figure 5.3: Example Queueing Mechanism in a Kokyu Dispatching Module

Kokyu scheduling framework is used to configure and operate a dispatching module.

During system initialization, each dispatching module obtains the thread priority and dispatching type for each of its queues from the scheduling service's output interface. Next, each queue is assigned a unique dispatching priority number, a unique thread priority, and an enumerated dispatching type. Finally, each dispatching module has an ordered queue of pending dispatches per dispatching priority.

To preserve QoS guarantees, operations are inserted into the appropriate dispatching queue according to their assigned dispatching priority. Operations within a dispatching queue are ordered by their assigned dispatching subpriority. To minimize priority inversions, operations are dispatched from the queue with the highest thread priority, preempting any operation executing in a lower priority thread [41]. To minimize preemption overhead, there is no preemption within a given priority queue.

The following three values are defined for the dispatching type:

- STATIC_DISPATCHING**:** This type specifies a queue that only considers the static portion of an operation's dispatching subpriority.

- DEADLINE_DISPATCHING**:** This type specifies a queue that considers the dynamic and static portions of an operation's dispatching subpriority, and updates the dynamic portion according to the time remaining until the operation's deadline.

- LAXITY_DISPATCHING: This type specifies a queue that considers the dynamic and static portions of an operation's dispatching subpriority, and updates the dynamic portion according to the operation's laxity.

The deadline-based and laxity-based queues update operation dispatching subpriorities whenever an operation is enqueued or dequeued.

5.3.1 Run-time Dispatching Priority

Run-time control over dispatching priority can be used to achieve the preemptive-by-urgency dispatching model discussed in Section 5.2. However, this model incurs greater complexity in the dispatching module implementation, which increases run-time overhead. Therefore, once an operation is enqueued in Kokyu's dispatching modules, none of the queues specified by the above dispatching types exerts control over an operation's dispatching priority at run-time. This greatly simplifies the dispatching module implementation since queues need not maintain references to one another or perform locking to move messages between queues.

As noted in Section 5.2, all the strategies implemented in the Kokyu framework map static priority directly into dispatching priority. Compared with strategies that modify an operation's dispatching priority dynamically, this mapping simplifies the dispatching module implementation since queues need not maintain references to one another or perform locking to move messages between queues. In addition, Kokyu's strategy implementations minimize run-time overhead since none of the queues specified by its dispatching types update any dynamic portion of an operation's dispatching priority. These characteristics meet the requirements of real-time avionics systems to which TAO has been applied [67, 41, 117, 118].

It is possible, however, for an application to define strategies that *do* modify an operation's dispatching priority dynamically. A potential implementation of this is to add a new constant to the enumerated dispatching types. In addition, an appropriate queue must be implemented and used to configure the dispatching module according to the new dispatching type. Supporting this extension is simplified by the flexible design of the Kokyu framework.

5.4 Scheduling Overhead in TAO's Real-Time Event Service

The conditions under which we ran the simulations in Section 4.6 were somewhat idealized. In particular, factors such as run-time overhead for dynamic scheduling mechanisms and OS dispatch latency [51] significantly affect the scheduling behavior of these strategies in actual systems. Therefore, the empirical benchmarks described in Chapter 8 are needed to validate the simulation results. To ensure that the Kokyu framework is efficient and predictable, we first performed initial measurements of:

1. the minimal latency added by dynamic scheduling in TAO's Event Channel, and
2. the basic dispatching overhead for the three types of queues used in the Kokyu framework.

The first measurements, described in Section 5.4.2, determined the run-time cost of dynamic dispatching for end-to-end performance. The second measurements, described in Section 5.4.3, assessed the potential increase in dispatching overhead as varying loads were placed on the dispatching queues described in Section 5.3. These tests demonstrate that Kokyu's dispatching modules can enforce dynamic end-to-end QoS requirements within acceptable levels of overhead.

5.4.1 Comparing Run-Time Performance

Several metrics are suitable for comparing the run-time performance of various scheduling strategies. Latency, latency jitter, laxity, missed deadlines, and CPU utilization can all be used to compare different scheduling strategies for a given set of operation characteristics. These metrics are defined as follows.

Latency: Latency is the amount of time an operation is delayed. It can be calculated by subtracting the CPU time used by the operation from the time between when it was requested and when it finished executing.

Jitter: Jitter is the amount of variation in the latency of an operation from invocation-to-invocation, relative to its average latency over time. Jitter for an invocation can be calculated by subtracting the latency for that invocation from the average latency. The absolute values of these measurements can be averaged to give the average latency jitter for an operation or a group of operations.

Laxity: An operation's laxity is the amount of slack time remaining after it completes its execution. Laxity can be calculated by subtracting an operation's completion time from its deadline.

Missed deadlines: Negative laxity indicates a missed deadline. The number of missed deadlines can be determined by computing the laxity of each operation invocation and counting the number of invocations that complete execution with negative laxity.

CPU utilization: CPU utilization is defined as the percentage of total CPU time that is used to execute an operation or a group of operations. CPU utilization for ORB endsystems can be subdivided into time used by operations, time used by the ORB and OS, and unused time.

The remainder of this section (1) describes an experiment to measure the minimum achievable end-to-end overhead for both static and dynamic scheduling strategies using TAO's Event Service over the TAO ORB, (2) describes an experiment to measure the overhead for static and dynamic dispatching queues as the load on these queues increases, and (3) draws conclusions about dynamic scheduling from the results of these experiments.

5.4.2 End-to-End Overhead

Levine conducted an initial experiment to quantify the actual overhead of the dynamic queues in our dispatching infrastructure, when applying the Kokyu scheduling framework to the TAO Event Service [42], shown in Figure 5.4. This experiment consisted of a single high-priority supplier/consumer pair, and a varied number of low-priority event supplier/consumer pairs, ranging from 1 to 1,000 pairs. By varying the number of low-priority suppliers and consumers, this experiment measured

1. the effect of increasing low-priority load on high-priority performance, and
2. the minimum relative overhead associated with dynamic operation dispatching.

He measured latency, which is the amount of time an operation is delayed, using time stamps. The run-time overheads for the static and dynamic scheduling strategies can be compared based on this measured latency. He measured the latency in event delivery between the high-priority supplier and consumer. This latency included

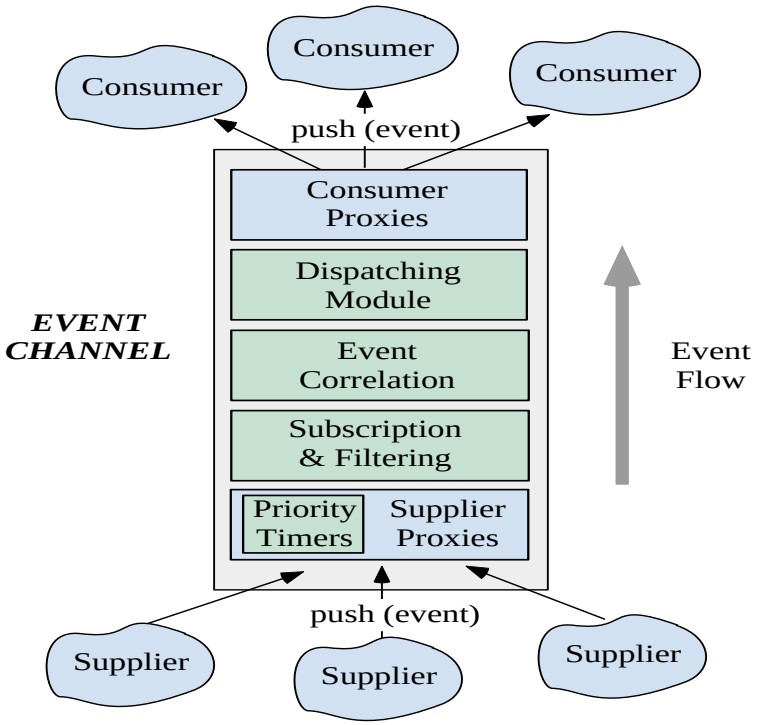

Figure 5.4: TAO's Event Service Architecture

- the time required for the TAO table-driven run-time scheduler to satisfy the Event Service dispatch module scheduling request, plus
- the time the request spent enqueued in the dispatch module.

The test was run for two different scheduling strategies on a Sun Ultra 30 uni-processor 300 MHz UltraSPARC CPU with 256 MB of memory, running SunOS 5.5.1 and using the real-time (RT) scheduling class [59].

An earlier version of the Kokyu scheduler was configured with an off-line RMS strategy and a run-time scheduler with $O(1)$ table lookup was used to provide the generated schedule at run-time. The dynamic strategy used MUF and therefore required an additional run-time laxity calculation. The high-priority supplier and consumer were paced so that each high-priority operation was dequeued before the next was enqueued. This design removed any queueing effect from the high-priority queue, so its minimum relative overhead could be measured accurately.

The results of this experiment are shown in Figure 5.5. This figure illustrates

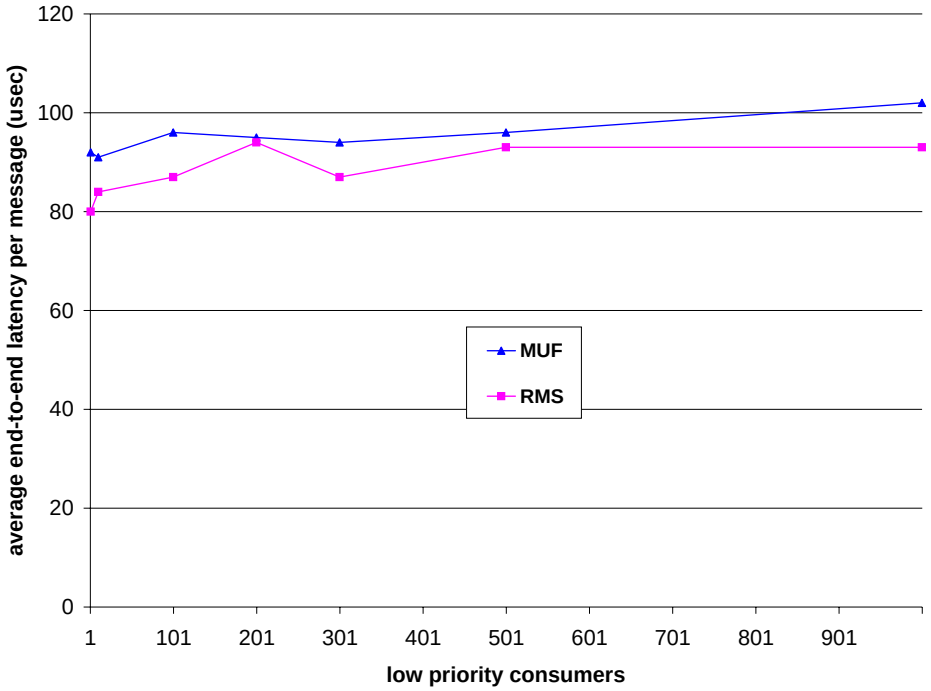

Figure 5.5: End-to-end Run-time Overhead of Dynamic Scheduling

that there was no significant change in high-priority performance with increasing low-priority load. Likewise, there appears to be only a small ($<$ 10 percent) overhead end-to-end for dynamic dispatching with no queueing effect. In addition, the absolute overhead was between 80 and 100 μsecs.

5.4.3 Overhead of Dispatching Primitives

The experiment described in Section 5.4.2 established the minimum relative end-to-end overhead for dynamic scheduling using the Kokyu dispatching primitives. Our second experiment gauged the potential impact of an increasing number of enqueued messages on this overhead. To measure this queueing effect accurately, we eliminated as many sources of constant overhead as possible. For instance, the queues were tested in isolation from TAO's Event Service and only the overhead of the enqueue and dequeue operations was measured.

The test was run on Windows NT 4.0 (SP3), in the real-time scheduling class on a dual-CPU Intel 333 MHz Micron Powerdigm with 256 MB of memory. The test used time stamps to measure the latency added by enqueue and dequeue operations for

an increasing number of messages in the queue. A separate iteration of the test was run for each of an increasing number of enqueued messages. Messages were enqueued in random order. The same order was used for all queues in a given test iteration.

The test was run with three different kinds of dispatching queues. We tested static, deadline-based, and laxity-based queues. The static queue, which was used by the RMS and Rate Monotonic Scheduling (RMS)+Minimum Laxity First (MLF) scheduling strategies, used a $O(1)$ table lookup at run-time. The deadline-based queue, which was used by the EDF scheduling strategy, required an additional deadline calculation at run-time. The laxity-based queue, which was used by the MUF, MLF, and RMS+MLF scheduling strategies, required an additional laxity calculation at run-time.

The dequeue overhead for the laxity-based queue was highest, followed by the deadline-based queue, and then the static queue. As shown in Figure 5.6, there was

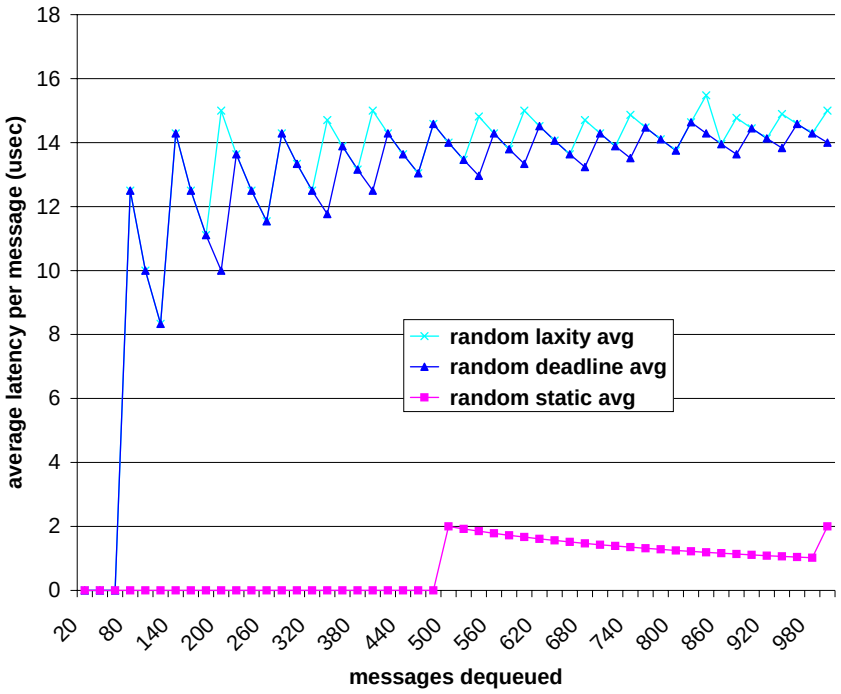

Figure 5.6: Average μsec/Dequeue

an initial increase in overhead for dequeue operations in the laxity and deadline-based queues as the number of enqueued messages increases. However, the overhead per-dequeue operation rapidly saturated at $\sim$14 μsecs per operation for these queues. Thus, as the number of enqueued operations increased, the overhead for dequeue operations

for the laxity- and deadline-based queues remained within a constant factor of seven times the overhead of the static queue.

The overhead for randomly ordered enqueue operations was highest for the laxity-based queue, followed by the overhead for deadline-based queue, and last for the static queue. As shown in Figure 5.7, the overhead per-enqueue operation increased linearly

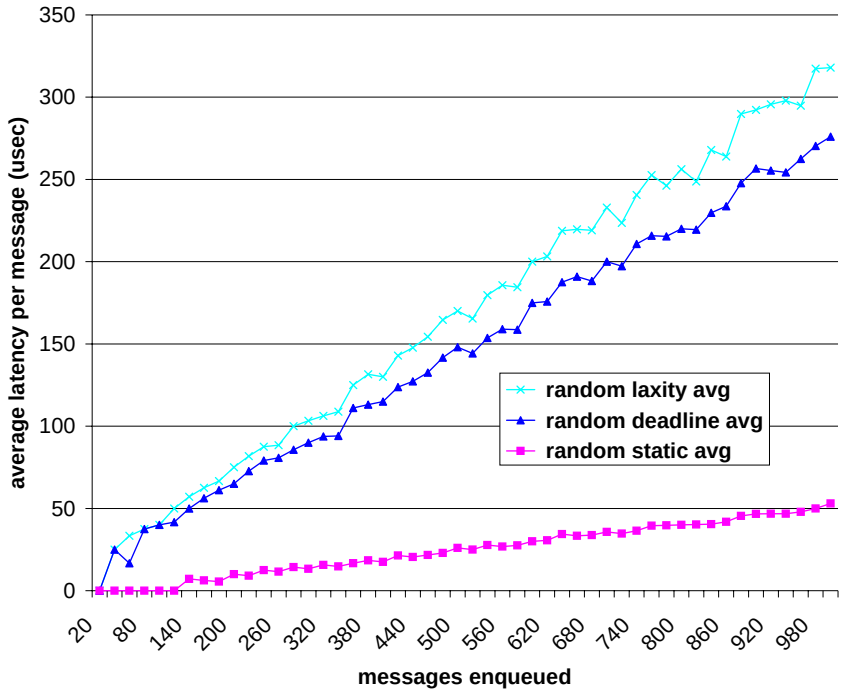

Figure 5.7: Average μsec/Enqueue

with the number of enqueued operations for all three kinds of queues. The overhead for enqueue operations for the laxity- and deadline-based queues remained within a constant factor of roughly six of the static queue overhead as the number of enqueued operations increased.

The tests described here and in Section 5.4.2 were run independently and in different experimental settings. Taken together, their results confirm empirically that dynamic scheduling strategies can be used effectively in real-time systems. Further, these results identify potential targets for optimization in cases where application requirements, such as heavy queue loading, may degrade performance.

Moderately-loaded systems: We now consider the implications of these results for systems with either moderate or heavy queueing, and discusses alternative dispatching

implementations and the conditions under which each may be preferable. Figure 5.5 shows that the minimal end-to-end latency for the laxity-based MUF scheduling strategy was only slightly higher than for the static RMS scheduling strategy. For systems where the maximum number of messages that can be enqueued at one time remains very small, the additional end-to-end overhead for dynamically scheduled dispatching should be relatively low.

If the number of messages that can be enqueued at one time increases, however, the effects of dynamic queue management become more prevalent, assuming a randomized enqueueing order. This dynamic queue management overhead is distributed between the enqueue and dequeue operations, so the measured overhead for both must be considered.

As shown in Figure 5.6, the overhead for dequeue operations does not appear significant for systems with fewer than 50 messages enqueued at one time. As the number of enqueued messages reached 100 messages, however, the overhead per-dequeue operation jumped to $\sim$12 μsecs. Even with a large number of enqueued messages, this overhead remained around 14 μsecs per dequeue operation, roughly a factor of six times the overhead per-dequeue operation in the static queue. Thus, the overhead from dequeue operations in the laxity- and deadline-based queues remains reasonable, even as the number of enqueued operations increases significantly.

As shown in Figure 5.7, the overhead for laxity-based and deadline-based enqueue operations does not appear to be significant if fewer than 20 messages are enqueued at one time. As the number of enqueued messages reached 50, the overhead per-enqueue operation for the dynamic queues jumped to $\sim$20 μsecs. Although the laxity-based and deadline-based enqueue overhead remained within a constant factor of six times the static enqueue overhead when more than 50 messages were enqueued, the significance of this constant factor increased with the number of enqueued messages.

Heavily-loaded systems: Depending on the characteristics of the specific application, the overhead for laxity- or deadline-based dispatching may reach unacceptable levels as the number of enqueued messages increases. Figure 5.7 shows that as the number of enqueued messages reached 1,000, the average overhead *per enqueue operation* exceeded 300 μsecs for messages enqueued in randomized order. Thus, the total CPU time needed to enqueue these 1,000 messages was above 0.3 seconds.

For systems with such a large queueing effect, the overhead from dequeue operations will be minimal compared to the overhead for enqueue operations in the dispatching queues. Section 5.4.4 discusses two alternative dispatching priority queue implementations and describes when each are optimal for different numbers of enqueued messages and different application characteristics.

The following conclusions can be drawn from the empirical results of our initial experiments:

Minimal end-to-end overhead: The minimal end-to-end overhead for the dynamic scheduling strategies is comparable to that for the static scheduling strategies, with only a small increase due to dynamic priority computations. This indicates that dynamic end-to-end QoS requirements can be enforced within acceptable levels of overhead, assuming other sources of system overhead are minimized.

Range of acceptable performance: The range of acceptable performance is sustained for dynamic scheduling strategies, up to a load of $\sim$150 messages enqueued at one time. The Kokyu scheduling and dispatching infrastructure can adapt flexibly to alternative queueing implementations, so that for heavier loads, heap-based queues may be preferable.

The empirical results presented here validate the simulation results presented in Section 4.6, and provide preliminary evidence for the efficacy of our dynamic scheduling approach. We expand this evidence with the empirical studies described in Chapter 8.

The overhead of enforcing dynamic end-to-end QoS requirements remains within acceptable limits for systems with light to moderate queue loading. Further, the empirical results suggest alternative queueing implementations to give optimal performance under increasing loads. Thus, dynamic scheduling using the Kokyu framework can be achieved both efficiently and predictably.

Qualitative comparisons of the steady state optimizations we propose for integrating predictable execution of the Real-Time Adaptive Resource Manager (RTARM) with mandatory and optional tasks were discussed in Section 3.5.1. We support these indications of the benefits of our approach with results from our previous work [32]. For example, figures 5.7 and 5.6 show respectively the average time spent on each enqueue and dequeue operation as a function of the number of messages enqueued in random arrival order, for each of the queueing disciplines in our framework [32].

These figures show that where possible (*e.g.*, for mandatory operations with harmonically related rates) scheduling a partition using static dispatching, such as is available in RMS, can lower overhead of operation dispatches. Depending on the granularity of periods and task execution times, this overhead reduction may bring significant benefits. On the other hand, for partitions where required schedulability is sub-optimal with RMS (*i.e.*, for mandatory operations with highly unrelated rates), using a dynamic queueing discipline such as deadline or laxity will incur greater overhead, but can help optimize other criteria such as the schedulable utilization bound.

5.4.4 Dispatching Infrastructure Extensions

The dispatching queues described in Section 5.3 are implemented as linked lists. This minimizes the dequeue overhead for the static, deadline-based, and laxity-based dispatching queues, even as the number of enqueued messages becomes large. For the statically dispatched queues, the dispatching overhead remains reasonable as well, even as the number of enqueued messages approaches 1,000. However, for the laxity- and deadline-based queues, the enqueue overhead grows significantly as the number of enqueued messages increases.

One alternative to a linked list message queue implementation is to use a *heap*. A heap is a partially-ordered, almost-complete binary tree that ensures the average- and worst-case time complexity for enqueueing or dequeueing is $O(\lg n)$. The trade-off is that in the linked list priority queue implementation, enqueue operations are $O\left(n\right)$ and dequeue operations are $O\left(1\right)$. Conversely, in the heap-based priority queue implementation, both enqueue and dequeue operations are $O\left(\log n\right)$.

Switching from a linked list implementation to a heap implementation can reduce the cost of enqueue operations while raising the cost of dequeue operations. Therefore, the selection of a dispatch queue implementation depends on application characteristics. For example, even with a large number of messages enqueued, a laxity-based queue may show $O\left(1\right)$ enqueue overhead if all messages have nearly identical execution times and times to deadline. Such idealized characteristics occur infrequently, however. Therefore, in systems where there is a larger queueing effect, heap-based implementations for laxity- and deadline-based queues may be preferable.

5.5 Configuration-Driven Dispatching Module Factory:

We conclude this chapter by describing how the dispatching infrastructure can be configured automatically using the scheduler output described in Section 3.4. We are currently modifying an existing version of the The ADAPTIVE Communication Environment (ACE) Object Request Broker (ORB) (TAO) Real-Time Event Channel (RTEC) dispatching infrastructure within the Bold Stroke avionics middleware to be configurable with such a factory. Four major segments of the Kokyu dispatching framework can be configured using information automatically generated by the Kokyu scheduling framework.

Dispatching Module: Priority lanes in the dispatching module itself can be collected either using a Native C++ array as in the existing TAO RTEC implementation [41], or in a more flexible data structure such as a hash map. The former approach is slightly more efficient in storage footprint and per-access time overhead, so for environments where in particular footprint is of extreme concern, this may be the preferable implementation. We have retained this approach in converting the existing TAO RTEC dispatching infrastructure within the Bold Stroke avionics infrastructure. The latter approach provides greater flexibility for adaptive reconfiguration without undue overhead for memory reallocation, and is our preferred approach for a new dispatching module implementation we plan to add to an enhanced and optimized version of the TAO event channel [99].

Message Queues: Two kinds of message queues are currently supported, one that provides static ordering of messages according to a fixed subpriority field, and one that allows arbitrary ordering of messages according to a pluggable strategy. The static queues are used to implement first-in-first-out (FIFO) or subpriority-ordered queues in RMS. Strategies for laxity and deadline ordering are currently provided for the strategized queue class, and are used to implement MLF and Earliest Deadline First (EDF) queues respectively.

Timers: Several kinds of timers are provided in ACE and the Kokyu dispatching infrastructure can be configured to use any of these. Choice of timer type(s) can be made for each application according to the number and pattern of timer intervals and the performance requirements of the application, particularly for predictability and overhead of timer dispatches and registration.

Concurrency Mechanisms: ACE also provides several concurrency mechanisms that can be used to manage priority isolation for timers, queues, or both. We consider three mechanisms here: tasks, reactors, and proactors.

The rest of this section considers the last three segments of the dispatching infrastructure in detail. Section 5.5.1 describes the Common Object Request Broker Architecture (CORBA) Interface Definition Language (IDL) configuration specification produced by the Kokyu scheduling framework. Section 5.5.2 describes the message queues provided by ACE. Section 5.5.3 describes the kinds of timers that can be configured. Finally, Section 5.5.4 discusses the task, reactor, and proactor concurrency mechanisms.

5.5.1 IDL Configuration Specification:

The `dispatch_configuration()` method of the Kokyu scheduling framework output interface provides a sequence of dispatch configuration descriptors, as described in Section 4.2.2. Figure 5.8 shows the CORBA IDL definitions provided by the RtecScheduler.idl file, as part of the `RtecScheduler` module interface for the Kokyu scheduling framework: As shown in Figure 5.8, the `Preemption_Priority_t` and `OS_Priority` types are used respectively to identify

1. the platform-independent priority level that also serves an index to that particular *priority lane*, and

2. the platform-dependent operating system thread priority used at that level to enforce priority isolation from other levels.

These types are used to configure the priorities of the various kinds of concurrency mechanisms described in Section 5.5.4.

The `Period_t` type is for time values in units 100 nanoseconds, corresponding to the period or interval of invocation of an operation, and used to set the expiration of periodic timers in the Kokyu dispatching framework as described in Section 5.5.3. Depending on the strategy with which the Kokyu scheduling infrastructure is configured, a particular priority level may have more than one associated period for its assigned operations. The `Period_Set` type declares a seqence of the periods associated with a particular priority lane.

The `Dispatching_Type_t` enumerated type identifies the kind of strategy to be used by a dispatching queue, as described in Section 5.5.2. Values of this type are

```
typedef long Preemption_Priority_t;
typedef long OS_Priority;
typedef long Period_t;
typedef sequence<Period_t> Period_Set;

enum Dispatching_Type_t
{
    STATIC_DISPATCHING,
    DEADLINE_DISPATCHING,
    LAXITY_DISPATCHING
};

struct Config_Info
{
    Preemption_Priority_t preemption_priority;
    OS_Priority thread_priority;
    Dispatching_Type_t dispatching_type;
    Period_Set timer_periods;
};

typedef sequence<Config_Info> Config_Info_Set;
```

Figure 5.8: IDL for Dispatching Module Configuration Descriptors

currently limited to `STATIC_DISPATCHING` for FIFO and static subpriority queues, and `DEADLINE_DISPATCHING` and `LAXITY_DISPATCHING` for deadline-ordered and laxity-ordered queues, respectively.

The `Config_Info` structure aggregates these types into a single descriptor for each priority lane:

- `preemption_priority` – platform-independent priority level, lane index
- `thread_priority` – OS priority of the thread(s) at that level
- `dispatching_type` – kind of dispatching queue
- `timer_periods` – timer periods associated with the priority level

Finally, the dispatching configuration descriptors can be collected as a sequence, of the type `Config_Info_Set`.

5.5.2 Message Queues:

ACE provides several forms of message queues, with which the Kokyu dispatching framework can bbe configured to produce different kinds of request ordering. Class `ACE_Message_Queue_Base` provides common enumerations and other types uesd by derived classes. Class `ACE_Message_Queue` extends class `ACE_Message_Queue_Base` and provides basic FIFO and static subpriority-ordered queueing. We added derived class `ACE_Dynamic_Message_Queue` to ACE, to extend base class `ACE_Message_Queue` with strategized dynamic ordering of messages. Any type that is derived from base class `ACE_Dynamic_Message_Strategy` can be used to configure message ordering in class `ACE_Dynamic_Message_Queue`. Class `ACE_Deadline_Message_Strategy` is derived from class `ACE_Dynamic_Message_Strategy` and implements deadline-ordered queueing. Class `ACE_Laxity_Message_Strategy` provides laxity-ordered queueing and is also derived from class `ACE_Dynamic_Message_Strategy`.

5.5.3 Timers:

ACE provides several kinds of containers for timer management. For each container, an iterator is defined, so that not only may the closest timer expiration be determined, but the entire sequence of registered timer expirations may be inspected. For efficient dispatching the former capability is used, though the ability to look ahead in the timer schedule appears useful for the kinds of adaptive reconfiguration at QoS transitions described in Section 9.3.

Class `ACE_Timer_Queue_T` provides features for a basic timer container. Classes derived from class `ACE_Timer_Queue_T` provide variations on the kind of container, particularly with respect to how the timers are stored. Examples provided by ACE include: `ACE_Timer_List_T`, `ACE_Timer_Heap_T`, `ACE_Timer_Wheel_T`, and for constant time access `ACE_Timer_Hash_T`. In addition, ACE provides two kinds of adapters for timer queues: class `ACE_Async_Timer_Queue_Adapter` extends class `ACE_Event_Handler`, and class `ACE_Thread_Timer_Queue_Adapter` extends class `ACE_Task_Base`.

5.5.4 Concurrency Mechanisms:

Finally, we consider three ways to provide prioritized concurrency management to timers and queues in the Kokyu dispatching framework. *Tasks* associate message queues and

threads according to the Active Object pattern [64]. *Reactors* provide synchronous event demultiplexing according to the Reactor pattern [115, 119]. *Proactors* provide asynchronous event demultiplexing according to the Proactor pattern [103, 119]. We now examine the ACE classes that provide each of these mechanisms.

Tasks: Class `ACE_Task` provides a complete implementation of an active queue abstraction, in which a consumer thread is associated with a message queue. Other threads post messages to the queue, and the consumer thread removes the message from the queue and processes it. By associating an event push upcall *function object* (also known as a *functor*) with a message, the RTEC [41] applies this abstraction to operation dispatching. As in the previous-generation static scheduling approach, we allow the priority of the task thread to be set to the assigned OS prioriry. We extend that capability by providing additional queue types and strategies with which to configure the task.

Reactors: Class `ACE_Reactor` provides an interface that can be implemented by any type derived from class `ACE_Reactor_Impl`, according to the Bridge pattern [28]. Implementation classes provided by ACE include:

- `ACE_Select_Reactor` – uses the UNIX `select ()` system call semantics
- `ACE_WFMO_Reactor` – uses the `WaitForMultipleObjects ()` system call semantics

Additional kinds of reactors are derived from those classes, and could be used to configure similar dispatching infrastructure in other settings: `ACE_XtReactor` (X Toolkit), `ACE_TP_Reactor` (thread pools), `ACE_QtReactor` (Qt Library), `ACE_FlReactor` (FL Toolkit), and `ACE_Msg_WFMO_Reactor`.

Several models are possible for configuring reactors in the Kokyu dispatching framework. For example, a single reactor could be used, or a reactor per priority level, each with its own thread of execution. In addition to the priority of the thread in which the reactor is run, reactors my use the semantics of the `ACE_Event_Handler` priority member to distinguish prioritization of dispatches. For example, priorities associated with handlers are inspected by class `ACE_Priority_Reactor` during its dispatch upcall.

Proactors: In addition to the synchronous event demultiplexing approaches currently used in the dispatching infrastructure, *asynchronous* event demultiplexing capabilities are becoming increasingly available in modern operating systems. ACE provides a set of

Proactor abstractions that encapsulate these capabilities. Class `ACE_Proactor` provides an interface implemented by any type derived from class `ACE_Proactor_Impl`, which is in turn derived from class `ACE_Event_Handler`.

Class `ACE_WIN32_Proactor` is derived from class `ACE_Proactor_Impl` and is implemented using the semantics of the system calls for win32 I/O completion ports. Class `ACE_POSIX_Proactor` is derived from class `ACE_Proactor_Impl` and is implemented using the semantics of the `aio_` system calls. Base class `ACE_POSIX_Proactor` is extended by `ACE_POSIX_AIOCB_Proactor` using Asynchronous I/O Control Blocks to notify or obtain status of `aio_` system calls. Class `ACE_POSIX_SIG_Proactor` is also derived from class `ACE_POSIX_Proactor` and is implemented using signals.

Finally, several helper classes are provided to integrate timer classes into the Proactor model. Class `ACE_WIN32_Asynch_Timer` defines the type posted to the completion port when a timer expires on Win32 platforms and is derived from the base class `ACE_WIN32_Asynch_Result`. Derived class `ACE_POSIX_Asynch_Timer` extends class `ACE_POSIX_Asynch_Result`, and defines the type posted to the completion port when a timer expires on UNIX platforms. ACE provides functors used that are used by class `ACE_Proactor_Handle_Timeout_Upcall` to allow parameterized management of timer expiration, cancellation, and removal.

We have not experimented with using Proactors instead of Reactors in the Kokyu dispatching infrastructure, though timer and priority configuration issue appear similar. As future work we plan to extend the Kokyu dispatching infrastructure to use Proactors for asynchronous event demultiplexing.

Chapter 6

Adaptive Rate-Selection Implementation

Section 3.5 discussed the idea of adaptation as a sequence of transitions between stable *operating regions*, as illustrated in Figure 3.5. Within each operating region the system parameters remain unchanged from the perspective of the application, other than minor variations in performance. In each transition between operating regions, some kind of adaptive transformation of the system is needed to maintain necessary invariants both during and after the transition. Two major forms of adaptation are examined in this dissertation: self-adaptation, and adaptive reconfiguration under the guidance of a higher-level resource manager. Chapter 8 establishes an empirical foundation for self-adaptation, and Chapter 9 suggests preliminary models and open problems in that area. In this chapter we focus on integrating the Kokyu scheduling framework more closely with a higher-level adaptive resource manager, particularly with respect to the issues described in Section 3.5.2.

This chapter is structured as follows. Section 6.1 describes our original approach to multi-layer adaptive resource management that connected an adaptive resource manager to the Kokyu scheduling framework through a controls-like sensitivity interface. Section 6.2 describes preliminary studies of performance of that sensitivity approach, and motivates the need to improve that performance. Finally, Section 6.3 describes optimizations to the Kokyu scheduling framework to remove the sensitivity interface and thus reduce the algorithmic overhead of adaptive transitions mediated by a higher-level resource manager.

6.1 Multi-Layer Adaptive Resource Management

Figure 6.1 shows an overall architecture for a middleware resource management infrastructure developed [25] under the Adaptive Software Test Demonstration (ASTD) program, a contract research and development (CRAD) project 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, under the Weapon System Software Technology Support (WSSTS) contract, number F33615-97-D-1155. In

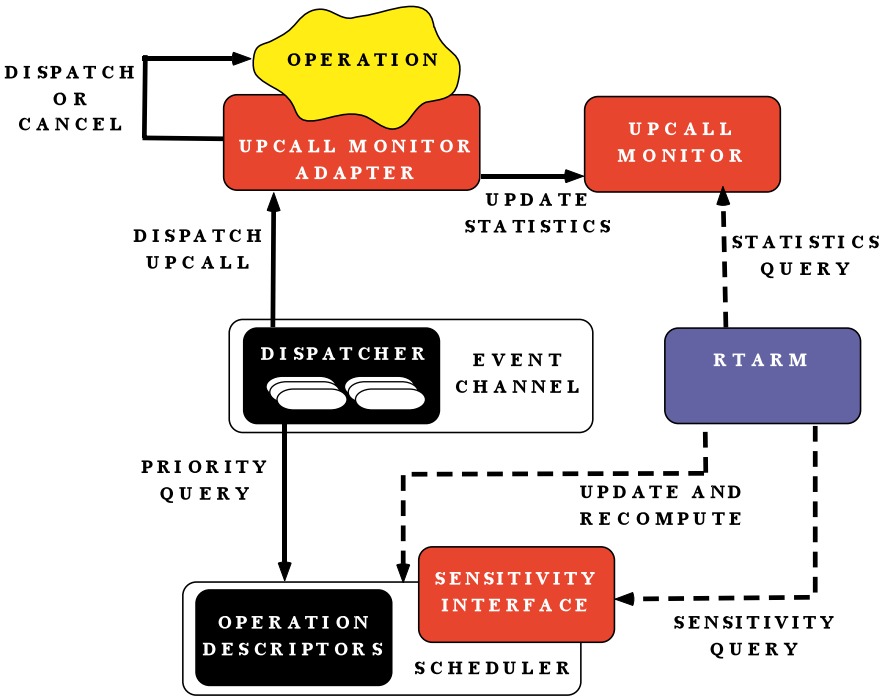

Figure 6.1: RTARM and Scheduler: Initial Integration

this architecture, an early version of the Kokyu framework was integrated with a Real-Time Adaptive Resource Manager (RTARM) [44] through a query interface designed to provide *sensitivity* information, thus enabling a form of closed-loop control over the quality of service (QoS) experienced by the application. The senstivity interface included the `operation_set_utilization_value` call, which returned the utilization level for a particular assignment of rates to operations by the RTARM, and the `operation_sensitivity` call, which returned a value indicating the sensitivity of the current utilization level to changes in the proposed rates.

The scheduler sensitivity interface provided access to algorithms needed by the RTARM to assess operation utilization and schedule sensitivity. Utilization is the portion of the total CPU time that is consumed by each operation. The RTARM can assess current utilization levels based on existing operation settings, or pose scenarios for utilization feasibility assessment by the scheduler. Sensitivity is a measure of how far an operation's rate can vary within the *feasibility* constraints imposed by the CPU resource and the demands of other operations. Because utilization by critical operations must not exceed the feasible bound, the scheduler uses the hard upper limit determined by the particular scheduling strategy. However, non-critical operations can be overscheduled in the worst case, to achieve improved utilization overall. For this reason, the scheduler allows the RTARM to specify a secondary, soft utilization bound that is greater than the theoretical limit, within which it will assess sensitivity for non-critical operations.

In contrast to dynamic scheduling, which adapts to jitter on a narrow time scale, adaptive resource management (ARM) adapts to longer-term variations in load and execution times. Figure 6.1 shows the *in-band* dispatching path in solid arrows:

1. from the dispatcher to the scheduler to obtain the upcall's priority,
2. from the dispatcher to the upcall monitor adapter to push the event,
3. from the upcall monitor adapter to the operation (unless the operation upcall is cancelled) to invoke the operation, and then
4. from the upcall monitor adapter to an upcall monitor to update stored dispatch deadline statistics.

Every event dispatch follows this path. The dashed arrows in Figure 6.1 show an *out-of-band* resource adaptation path:

1. from the RTARM to the upcall monitor to obtain statistics,
2. from the RTARM to the scheduler to query schedule sensitivity, and
3. to the scheduler to update operation characteristcs and recompute priority assignments.

In the architecture shown in Figure 6.1, the RTARM provided two types of adaptation:

- contraction and expansion of feasible QoS regions defined in terms of the set of enabled operations and their selected rates, and

- adjusting the operating point within a QoS region based on run-time feedback of actual QoS.

For example, in a system of rate-adaptive periodic operations, QoS contraction decreases maximum operation rates, whereas feedback adaptation varies operation rates within the currently active min-max range.

The RTARM and scheduler collaborated to perform a kind of admission control for operations and operation rates. Operations were categorized as being either hard real-time (HRT) or soft real-time (SRT), with the HRT operations at a higher criticality level than the SRT operations. A schedule (a particular assignment of rates and priorities to a set of operations) was considered feasible if the maximum utilization by all the HRT operations in the schedule fit within the CPU bound. A schedule was considered optimal if it was feasible and all operations in the schedule fit within the total bound. The RTARM was designed to use various algorithms for deciding which operations would be scheduled, and at which rates. To perform its admission control algorithm, the RTARM iteratively extended a set of $< operation, rate >$ bindings, adding new bindings and updating existing ones based on responses from the scheduler to feasibility and sensitivity queries.

6.2 Performance of the Sensitivity Approach

Qualitative comparisons of the adaptive transition optimizations for integrating predictable RTARM execution with critical and non-critical operations were discussed in Section 3.5.2. We support these indications of the benefits of our approach with results measuring the behavior of the previous generation RTARM and scheduler during adaptive transitions over a small number of operations. We conducted a simple experiment to measure overhead factors for the original sensitivity-based adaptive rescheduling approach described in Section 3.6. This experiment was conducted on a single CPU (300MHz Pentium) machine, running the Windows NT Workstation 4.0 operating system. Since the experiment was designed to assess the order of complexity and constant overhead factors of adaptive rescheduling using the original sensitivity-based approach, and not the precise values of the overhead curves in the target platform environment, we measured these results directly in the Windows NT desktop environment used for development and functional testing of target platform applications.

In this experiment, we emulated the *operating regions* described in Section 3.5 and illustrated in Figure 3.5, as a sequence of *mission states*. A mission state consisted of a definition of the operations that were enabled in that state and the range of available rates for each enabled operation. When a mission state transition changed which operations should run, and a what available rates, the RTARM would perform an admission control algorithm to determine which operations could be feasibly (and hopefully optimally) scheduled, assigning each operation one of its available rates. Figures 6.2 and 6.3

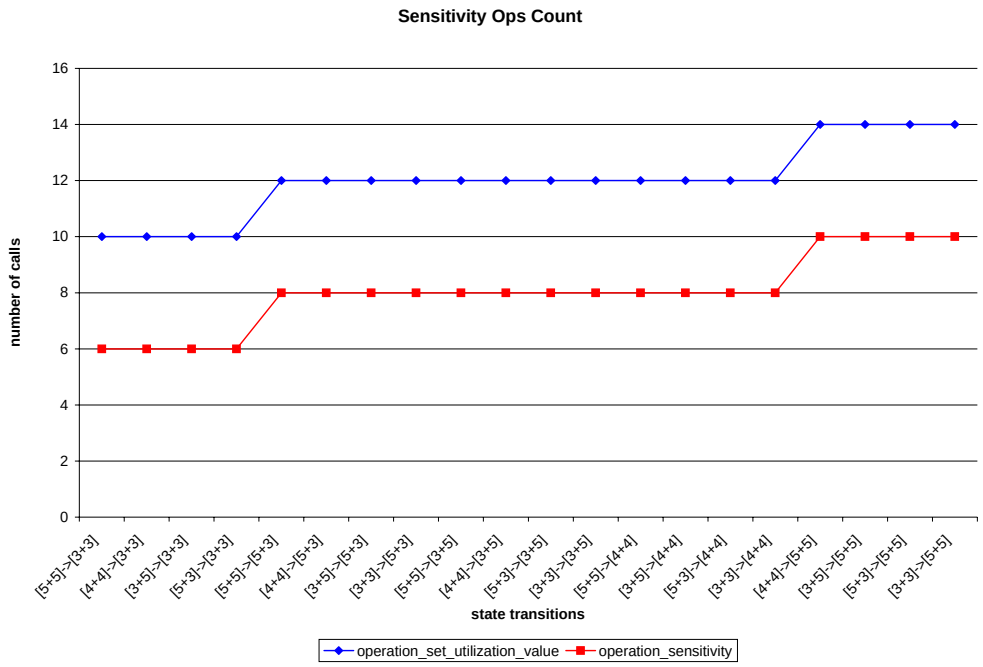

Figure 6.2: Adaptation Method Call Counts

show respectively the number and average duration of calls to our scheduler by the earlier-generation RTARM during an adaptive transition. In each figure, we plot data for the `operation_set_utilization_value` and `operation_sensitivity` calls.

Each horizontal axis label describes an adaptive transition. The first bracketed numbers in each label show the number of critical and then non-critical operations in the mission state before the adaptive transition, and the second bracketed numbers show the number of critical and then non-critical operations in the following mission state. Both the average time and number of sensitivity interface calls give plots that are linear in the

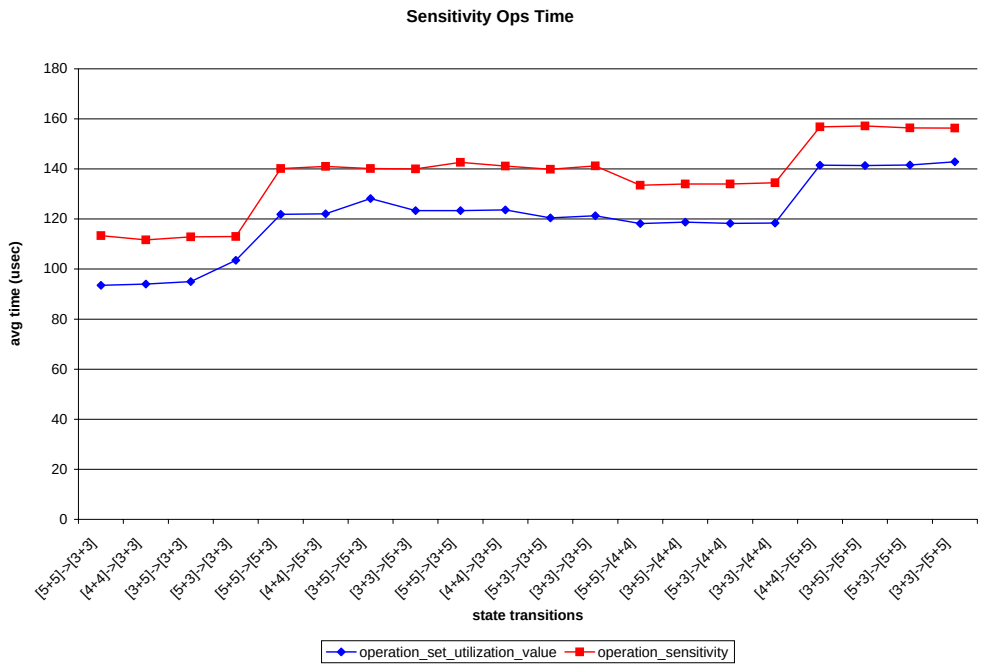

Figure 6.3: Average μsec/Call

total number of operations in the *destination* state during an adaptive transition, for an expected resulting adaptive transition time that is a *quadratic* function of the number of operations in the destination state.

These results motivate the comparison sort and radix sort optimizations to adaptive rescheduling described in Section 6.3. In particular, we expect that constants in the order complexity equations (*i.e.*, $C_0, ..., C_6$) to be of similar magnitude, according to the following reasoning. First, the constant overheads in the original sensitivity based case were proportional to two function calls (feasibility and sensitivity) per operation and an inspection of each operation per function call. In the comparison sorting case, there is only one function call (sort), but at worst $nlog(n)$ comparisons of two operations, where n is the number of operations. Finally, in the radix sorting case, there is again one function call (sort), and a single light-weight hash computation per operation.

6.3 Adaptive Optimizations

The structure of our solution by sorting is one of

1. de-normalizing the RT_Info and available rate information about each information into a set of flat structures,
2. decorating these structures with derived information that facilitates sorting and utilization bounds checking,
3. encapsulating sorting algorithms to enforce different admission control policies using the Strategy design pattern ,
4. performing one or two $O(nlog(n))$ sorts, depending on the relative stability of the rate and priority assignment sorts, and finally
5. performing a single $O(n)$ traversal to check utilization bounds and select the final operation rates.

Having motivated the *need* for optimization to the RTARM and scheduler interaction in Section 6.2, we now expand on the list of adaptive optimizations noted in Section 3.6, and illustrated in Figure 3.8. Each optimization serves to reduce the latency and jitter, and improve the accuracy, of adaptive transitions.

A. De-normalized operation descriptors: We de-normalize the available rate set and fixed characteristics for each operation into a sequence of flat tuples of characteristics (containing *e.g.*, the operation handle, a particular rate, the execution time at that rate). We then derive information that facilitates sorting for and utilization bounds checking. For example, we specify the index of a tuple within an operation's ordered set of rates, and the utilization difference for an operation between each pair of its consecutively indexed tuples. This optimization can help meet our goal to trade performance of individual elements (*i.e.*, rate of execution) for overall performance objectives (*i.e.*, maximizing the number of feasible operations).

B. Rate and priority sorting: We recast rate and priority assignment as a sorting problem over operation characteristics, with at worst an $O(nlog(n))$ bound on worst-case performance, and an $O(n)$ bound on worst-case performance in certain special instances of the more general problem. Since our scheduling approach applies to arbitrary collections of operation characteristics, for some combinations of operations and

scheduling strategies an $O(nlog(n))$ comparison sort may be needed. For our target avionics application, however, all operations are known in advance and the value spaces of the characteristics of interest (*e.g.*, whether an operation is critical, its available periods) are small, so the more efficient $O(n)$ radix sorts are applicable in many cases.

C. Assignment policies: We encapsulate specific sort ordering strategies as policies for rate assignment, much as we have done previously for scheduling policies [32]. We present two canonical strategies for rate selection, based on two different views of fairness: *Fair Assignment by Indexed Rate* (FAIR), and *Criticality-Biased FAIR* (CB-FAIR), described in detail in Section 6.3.2.

Other policies and other criteria besides the FAIR and CB-FAIR strategies are possible, and seem likely to be beneficial. In particular, strategies that consider the dependencies between operations seem promising for the inter-component rate dependencies of complex avionics mission computing applications [41]. As with our approach to scheduling [32], our approach to rate selection strives to combine flexibility and efficiency, to support optimized performance for a range of distributed real-time and embedded applications.

D. Rate Selection: Once the tuples are sorted, we perform a single $O(n)$ traversal of the tuples to select the rate of each operation and determine expected utilization values based on the rates selected and the advertised execution times. As we iterate through the sorted tuples, we maintain variables for (1) the total utilization by critical operations, and (2) the total utilization by all operations, based on the tuples selected so far. A tuple is selected if and only if the additional utilization, compared to the utilization for the previously admitted tuple for that operation, will still fit within the utilization threshold associated with that tuple. The highest rate of any tuple selected for an operation becomes the assigned rate for that operation. As described above, critical and non-critical operations may have different associated utilization thresholds. Due to a particular sorting order and the relative utilization differences between tuples, with this approach it is possible for a tuple to be skipped for an operation, but for a later tuple for another operation to be admitted.

6.3.1 Recasting Admission Control as a Sorting Problem

These optimizations can help meet our goal to perform adaptive resource reallocations within firmly bounded time-scales. For example, consider a realistic application with

64 schedulable operations, each of which has (1) one of a fixed small set of criticality values, and (2) an associated set of available invocation periods chosen from a fixed similarly small set of period values. If we applied the previous-generation sensitivity-based approach, we would expect adaptive rescheduling to occur in time bounded by:

$$C_0 + C_1 n + C_2 n^2 = C_0 + 64C_1 + 4096C_2 \tag{6.1}$$

If we instead applied a comparison sorting strategy for combined rate and priority assignment, we would expect a tighter bound on adaptive rescheduling:

$$C_3 + C_4(nlog_2(n)) = C_3 + 384C_4 \tag{6.2}$$

Finally, if we instead applied a radix sorting strategy for combined rate and priority assignment, we would expect a still tighter bound on adaptive rescheduling:

$$C_5 + C_6 n = C_5 + 64C_6 \tag{6.3}$$

The constant overheads for the sensitivity, comparison sorting, and radix sorting approaches are expected to be similar, as follows.

Discrete rates: First, operation rates are discrete, due to the frame-based nature of the OFP. This means that the operation characteristics described in the fields of an operation's RT_Info, combined with the operation's list of possible rates in a particular mission state, can be de-normalized initially into a set of discrete $< handle, period >$ tuples.

Small number of rates: Second, the number of possible discrete rates per operation is very small, being at most 5: 40Hz, 20Hz, 10Hz, 5Hz, and 1Hz. This means that the de-normalization of operation characteristics can be achieved with only a constant worst-case increase in the amount of space required per operation.

Ordered rates: Third, the rates themselves have an inherent numerical ordering. This means that each tuple for an operation in a mission state can be tagged with an index, with the lowest rate having the lowest index (0), the next higher available rate having the next higher index (1), and so on. This helps to define and enforce arbitrary partial orderings on tuples based on the available rates, which are essential to policies like FAIR and CB-FAIR.

Fixed execution times: Fourth, the advertised worst-case execution times of the operations are constant, at least per rate. This means that each tuple can be further decorated with a $\Delta utilization$ value reflecting the increase in utilization from that consumed by all other pairs with that same handle but with a lower rate (longer period). This helps transform the activity of checking utilization limits and selecting final rates for operations into a linear traversal of the sorted tuple set.

Derived characteristics: Fifth and last, the operation characteristics described in the RT_Info, together with the available rate information, are sufficient to compute additional derived information for each tuple, such as the mean rate of all available rates for an operation, that can be used to implement specific admission control policies as sorting strategies. Each tuple can also be tagged with any additional characteristics from the operation's RT_Info that matter to any of the admission control algorithms, such as the criticality of the operation. To implement the FAIR and CB-FAIR policies, the resulting tuple must hold at least the following five fields:

$$< handle, criticality, index, mean\,rate, \Delta utilization >$$

Therefore, we anticipate that experiments currently in progress to measure these factors precisely in all three cases, using a realistic application with around 64 schedulable operations on the target platform, will show adaptive rescheduling overhead reductions on the order of:

- **90%**, *i.e.*, a ten-fold reduction – going from the sensitivity approach to the comparison sorting approach.
- **98%**, *i.e.*, a fifty-fold reduction – going from the sensitivity approach to the radix sorting approach.

6.3.2 Sorting Strategies

It is thus possible to sort the de-normalized tuples to implement efficiently a particular rate admission control policy. The Strategy design pattern helps structure the design for flexible substitution of admission control policies. This in turn allows both employment of different sorting algorithms such as comparison or radix sorts, and implementation of different admission control policies such as FAIR or CB-FAIR to be encapsulated together and plugged into a more general framework, thus increasing overall software reuse.

The key insight is that FAIR and CB-FAIR differ only in the way they arrange the top-level branches in their decision trees. This property holds whether they are modeled as partial ordering functions for comparison sorts, or hash functions for radix sorts. Therefore, we can define a suitable strategy class in each case. Table 6.1 summarizes the criteria in the decision trees and their order of evaluation (from top to bottom) for the FAIR and CB-FAIR strategies. The remainder of this section examines the decision

Table 6.1: Ordered Sorting Criteria for FAIR and CB-FAIR

FAIR	**CB-FAIR**
rate-index	criticality
criticality	rate-index
mean rate	mean rate
handle	handle

trees in greater detail for each of these policies.

FAIR Strategy: The *Fair Assignment by Indexed Rate* (FAIR) strategy is illustrated on the left side of Figure 6.4. It emphasizes fairness across all operations, ordering the collection of tuples:

1. by ascending rate index, then
2. by descending criticality, then
3. by mean rate, and finally
4. by descriptor handle.

This strategy selects the lowest rate for each operation, first for critical operations and then non-critical operations, then the next rate for each critical operation and the each non-critical operation, and so forth. The mean rate and descriptor fields are used to break ties in both rate index and criticality. In essence, the FAIR strategy introduces a kind of rate allocation fairness across all operations.

CB-FAIR Strategy: The *Criticality-Biased FAIR* (CB-FAIR) strategy, illustrated on the right side of Figure 6.4, emphasizes criticality partitioning first, and orders tuples:

1. by descending criticality, then

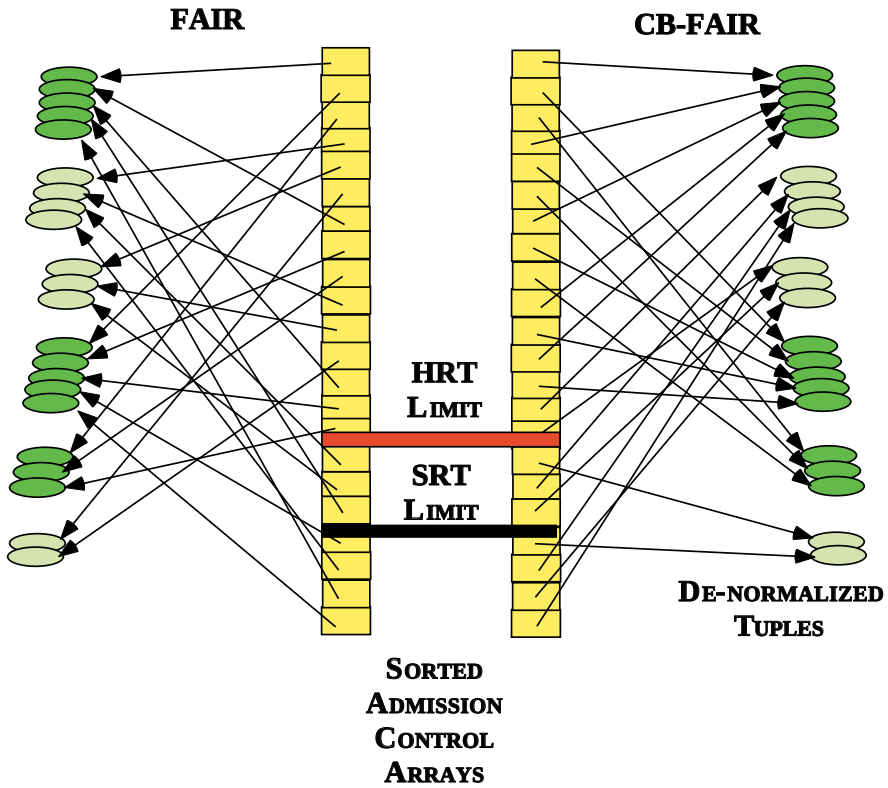

Figure 6.4: FAIR and CB-FAIR Rate Selection Strategies

2. by ascending rate index index, then

3. by mean rate, and finally

4. by descriptor handle.

The CB-FAIR strategy selects the lowest rate for each critical operations, then the next rate for each critical operation, until all critical operations are at their highest rates and then repeats the process for the non-critical operations. As in the FAIR strategy, the mean rate and descriptor fields are used to break ties in both rate index and criticality.

6.3.3 Iterative Admission Control

Once the tuples are sorted, it is then possible to iterate once through the sorted tuples, maintaining a variable with the total utilization by all tuples admitted so far . A tuple is admitted if and only if the addition of its $\Delta utilization$ will still fit within the utilization threshold associated with that tuple. Note that it is possible, due to the sorting order and

the relative sizes of $\Delta utilization$ values, for a HRT tuple to be skipped, but a later HRT tuple with a smaller $\Delta utilization$ value to be admitted (and similarly for SRT tuples).

As each tuple is admitted, the period of its corresponding RT_Info is set to that tuple's period. This results in each operation having the highest rate of any of its admitted tuples, because the tuples of a particular operation are always sorted by period index, with tuples for that operation having shorter periods falling after tuples (for that operation) having longer periods. For some SRT operations, it is possible that no tuples could be admitted to a feasible schedule. In this case, the operation itself should be omitted from the final schedule. This can be achieved by initially marking all operations inactive, and when a tuple is admitted marking its operation active. The set of active operations (with their final rates and other characteristics) and the total HRT and SRT utilization values are the final outputs of the admission control algorithm.

One final note is that scheduler functions such as priority assignment are needed to turn an admitted set of operations with their rates into a schedule that can be dispatched. For some combinations of admission control and scheduling strategies for the architecture, it may be possible to perform a priority ordering of tuples as part of the same sorting pass that orders the tuple set for admission. That is, the priority assignment sort is *stable* [18] with respect to the rate sort. For such combinations of policies and sorting algorithms, it would then be possible also to perform priority assignment to tuples (and transitively to operations) as part of the same pass that admits tuples.

However, this is not always possible in general. In particular, the order in which criteria are considered by the admission control ordering and priority ordering may differ, and the criteria considered may overlap. For example, CB-FAIR would place a 10 Hz HRT tuple before a 20 Hz SRT tuple, while Rate Monotonic Scheduling (RMS) would reverse their order. This means that in some cases a second separate sorting of tuples and/or operations must be performed for priority assignment. However, this does not increase the order of the complexity bound for the scheduling framework overall: the bound is preserved within a constant factor for admission control and priority assignment performed either in the same pass or separate passes.

Chapter 7

Metrics Feedback Framework Implementation

To measure, assess, and visualize real-time performance of rate-based distributed real-time systems, a coherent infrastructure for data collection, storage, transfer, and visualization is needed. Figure 7.1 illustrates such a framework, which has been applied and subsequently evolved under each of three research programs: Adaptive Software Test Demonstration (ASTD), Adaptive Software Flight Demonstration (ASFD), and Weapon Systems Open Architecture (WSOA), all managed by The Boeing Company under contract to the Air Force Research Labs (AFRL).

The three major areas of this framework are:

1. instrumentation and monitoring of real-time latencies and operation deadline statistics as described in Section 7.1,

2. consistent time frame management as described in Section 7.2, and

3. remote logging and visualization infrastructure as described in Section 7.3.

We note important similarities between segments of the Kokyu metrics framework and previously published work by Dr. Douglas Niehaus and his research group at the University of Kansas. In particular, the frame manager described in Section 7.2 serves a similar role to the UTIME temporal resolution work in KURT Linux [122] and the Proteus project in that the question of time consistency is a central theme. Our approach differs in its explicit representation of *time frames*, a worthy first-class abstraction for rate-based systems, which are the predominant target of our research.

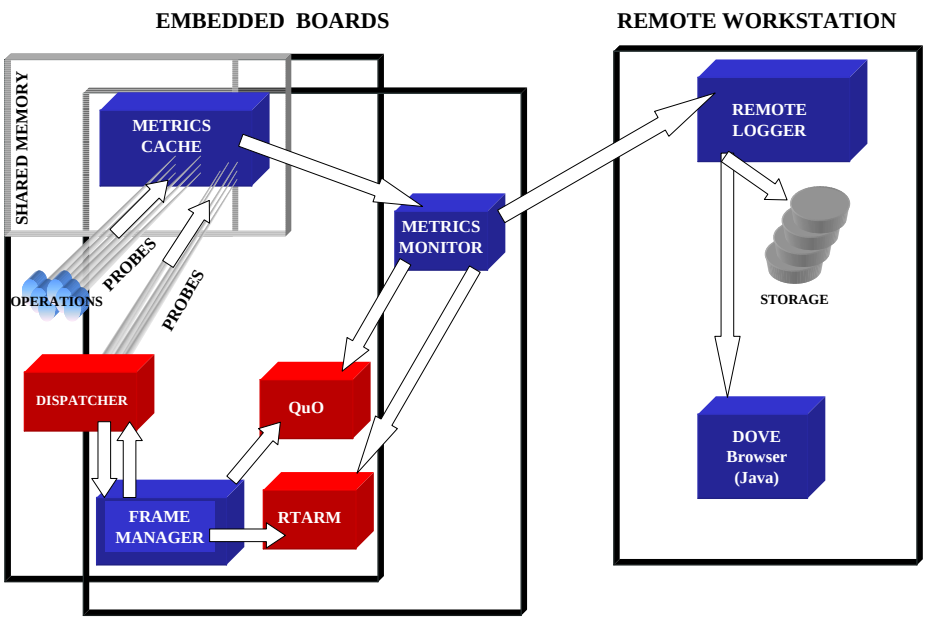

Figure 7.1: Metrics Framework

Furthermore, the data streams kernel interface (DSKI) [88] offers flexible and efficient real-time performance data collection and monitoring, in a way similar to the approach described in Section 7.1. Our approach is similarly customized to the specific target environment we have studied, and in partular provides a set of dynamically configurable C++ classes that can be used within shared memory, such as that provided by the VME backplane described in Section 8.1.3 and illustrated in Figure 8.2. As future work we plan to examine these respective infrastructures carefully, and eliminate any areas of unnecessary overlap from the Kokyu metrics framework, focusing instead on integration with the University of Kansas work so that the Kokyu metrics framework addresses issues in other dimensions, and is an entirely complementary technology.

7.1 Instrumentation and Monitoring

Four main areas of instrumentation and monitoring are needed to provide a complete measurement capability for experiments on the scale of those decribed in Chapter 8.

1. instrumentation of the dispatching module to profile queueing latency as discussed in Section 8.3.1,

2. instrumentation of the upcall monitor adapter to profile operation execution latency as discussed in Section 8.4.2,

3. the common collection point for data in both these areas within a shared-memory capable metrics data cache, and

4. coordination of the upcall monitor adapter with the upcall monitor to profile operation deadline success as discussed in Section 8.3.2.

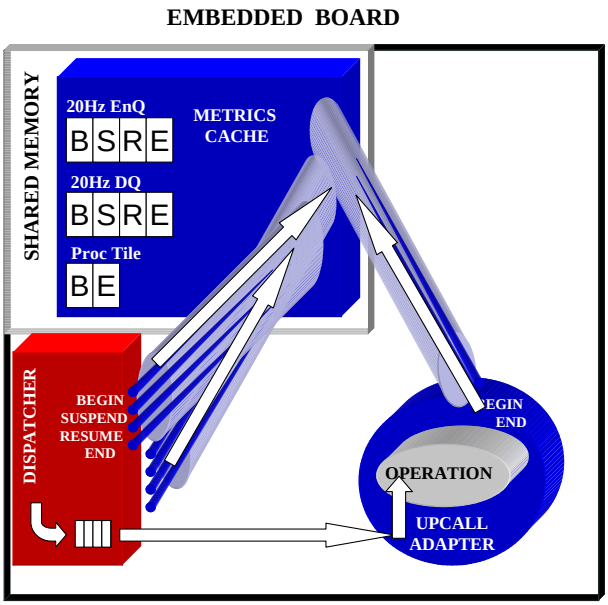

Figure 7.2: Metrics Cache

Figure 7.2 illustrates the first three of these areas. We consider each area separately, as follows.

Queueing latency: The overhead for dequeing (and enqueing) operations in both the static and dynamic queues was measured using light-weight time probes. Because the dynamic queues may perform re-ordering before trying to wait on a “not empty” condition variable, and then dequeue the operation after acquiring the appropriate lock, it was necessary to suspend and resume the metric, so that only the CPU time actually consumed by the dynamic queue was measured. This was achieved by extending the

time probe class provided by ACE to log suspend and resume time probe events, and to assess total overhead accordingly.

Execution latency: Just before it passes the thread of control into the operation by pushing an event to its enclosing component, the upcall monitor adapter takes a *begin* time stamp When the thread of control returns from the component, the adapter immediately takes an *end* time stamp. The sequence of time stamps is stored in the metrics cache, and a later iteration through those time stamps simply subtracts each begin stamp from its corresponding end stamp to obtain the specific execution time. These time stamps are taken using the high resolution timer tied to one of the following, in decreasing degree of resolution:

- a specilized hardware timer,
- a timer feature of a general purpose hardware element such as the Pentium high-resolution clock tick counter, or
- a general purpose operating system call such as `::gethrtime ()`.

For the systems we have studied, microsecond resolution is mostly sufficient, so unless we require a finer-granularity we use the `ACE_Time_Value` class provided by ACE to offer

- efficient built-in operators for convenient and flexible time calculations,
- built-in conversions to and from other time types, and
- use by and with a high-resolution timer class,

all while preserving microsecond resolution.

A cancellation capability was also implemented in the upcall monitor adapter during the ASTD program. For each operation upcall, the upcall monitor adapter can also look at the begin stamp prior to pushing the upcall event to the component, and determine based on the deadline, that time stamp, and the operation's worst case execution time whether or not the operation is at risk. If an operation is determined to be at risk and cancellation is acceptable at its criticality level, the adapter can cancel its dispatch upcall in favor of operations that are more likely to meet their deadlines. For ASTD, we added the constraints that the entire cancellation feature can be enabled or disabled, and that only non-critical operations can be cancelled.

Metrics cache: In the ASFD program experiments described in Chapter 8, we significantly evolved the metrics data collection and storage capabilities of the Kokyu metrics framework. We removed virtual functions and other features that conflicted with our goal of caching data in shared memory to reduce latency and improve access during narrow windows of execution so that we could:

1. perform metrics data collection and distribution in as small a memory footprint and in as narrow a time window as possible,

2. avoid overflows in either time or space, and by doing so,

3. minimize the impact of data collection and distribution on the rest of the application.

We were not able to complete all refactoring of the metrics cache during the ASFD program, so two additional evolutions were performed during the WSOA program:

1. replacing native C++ pointers with `ACE_Based_Pointer`s, which are smart pointers that remember the offset *at which they were allocated* in shared memory, and re-thunk their target address based on their `this` pointer at each dereference, and

2. adding template support for parameterized allocators, which can then be used to ensure integrity of dynamically allocated data structures in shared memory.

Upcall monitor and adapter: In addition to integrating the scheduler and Real-Time Adaptive Resource Manager (RTARM) as described in Section 6.1, a second major development effort under the ASTD scheduling framework was to provide feedback on operation progress to the RTARM. A special component adapter, called an *upcall monitor adapter*, was applied to each event consumer containing an operation we wished to monitor. The upcall monitor adapter served as a proxy for the event consumer, both to the dispatching module and to an *upcall monitor* that records statistics about the real-time success, failure, and cancellation of dispatch upcalls to operations.

The application registered each of its components' upcall monitor adapter as an event consumer with the Event Channel, in place of the component. The Event Channel then pushes all events destined for the component to its adapter instead. When the adapter receives an event upcall, it may make a decision to cancel the upcall but otherwise pushes to the component.

When the thread of control returns from the component operation upcall, the adapter compares the completion time to the event's deadline. If the deadline expired before the completion time, the operation had missed its deadline. Otherwise, the deadline was made. The adapter passed the success or failure of the deadline to the upcall monitor via a low-overhead inline reporting method. The RTARM periodically polled the upcall monitor to determine the progress of various operations, and to adjust its adaptive behavior accordingly.

7.2 Time Frame Manager

Two phenomena observed during the development phase leading up to the experiments described in Chapter **??** motivated the development of a common manager for time frames and deadlines. First, we noticed the special case that the end of frame timeout event described in Section 8.1.3 had been wrapped with an upcall monitor adapter and was dispatched at the end of one frame, but executed in the beginning of the next frame. Therefore, under the convention of an event's deadline being the end of the frame in which it was dispatched, we were seeing a consistently reported deadline miss for that hard real-time (HRT) event. As the observed behavior

1. was in fact correct for that event, and

2. was more similar to internal event handling than application-level event handling,

we simply removed the upcall monitor adapter for that event consumer, and only considered *application* operation deadlines in our experimental protocol. However, we also interpret this case as representative of a particular class of applications where operations may be dispatched with arbitrary phasing of frames, and which motivated development of a separate manager to maintain frame and deadline information.

The second phenomenon we observed that also motivated development of a distinct *frame manager*, was due to the use of Event Filter Discriminator (EFD) [41] performance optimizations to support filtering and correlation among events at the same priority level, which therefore could bypass the dispatching module itself. Because each event was stamped with its deadline by the dispatching module, when an EFD event reached its upcall monitor adapter, it had an uninitialized value in its deadline field needed for deadline reporting and cancellation calculations. In the experiments

described in Chapter 8, we worked around the problem by having the upcall monitor adapter compute the deadline if it was not already assigned. Clearly, proliferation of deadline calculations throughout the system could lead ultimately to unacceptable maintanance costs and brittle systems infrastructure. Therefore, this case also motivates the development of a single, separate metrics frame manager class.

Figure 7.3 illustrates the frame manager class we have developed for the Kokyu metrics framework. Various points in the system architecture, *i.e.*, the application, the

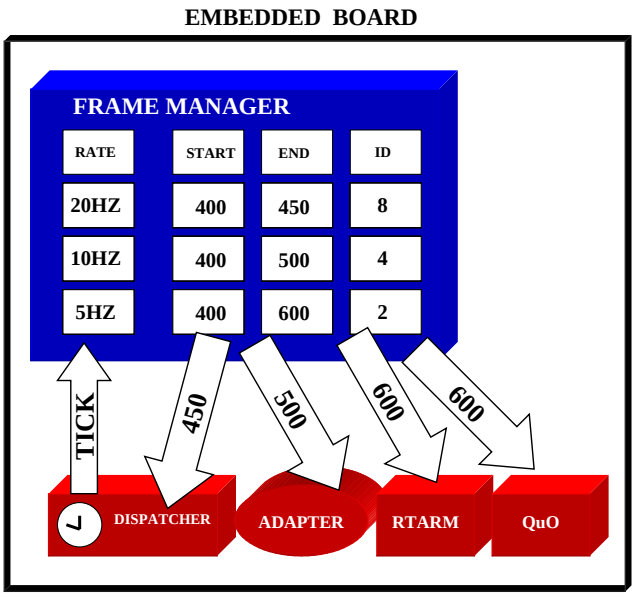

Figure 7.3: Metrics Frame Manager

scheduler, the dispatcher, and the upcall monitor adapter, are able to:

1. obtain access to the frame manager through a singleton wrapper class,
2. register rates with the frame manager,
3. query a uniqueue frame id for a given rate,
4. query the start or end time of the current frame at a given rate, and
5. ask the frame manager to update its frames with respect to the current time or a time given to the frame manager.

The ability to wrap a single instance of the frame manager class as a singleton allows a common and consistent representation of time frames to be shared across dispatching threads, system layers, and epochs of execution. Supporting registration of arbitrary rates makes the frame manager more portable across kinds of real-time applications. The start and end times, maintained on-demand for each rate, provide precise information about deadlines and release times, without undue overhead. Finally, the use for frame ids supports both local and distributed transition protocols of the kinds described in Chapter 9.

7.3 Integration with Remote Logging and Visualization

Many sensor-driven systems, such as those for avionics mission computing and for manufacturing process control, have stringent timing requirements for processing sensor data. Furthermore, many of these systems must manage multiple sources of sensor data simultaneously. The results in Chapter 8 demonstrate that sensor-driven systems can be implemented efficiently and predictably using a real-time CORBA Event Service. This approach allows designers of real-time systems to leverage the benefits of flexible and open distributed computing architectures, such as those defined in the CORBA specification, while still meeting real-time requirements for efficiency, scalability, and predictability. To build and manage these types of systems, application developers and test engineers must be able to monitor and visualize the systems' real-time behavior.

Rate monotonic analysis and other scheduling strategies have been developed to help ensure that real-time systems achieve this goal. However, scheduling strategies alone offer little guidance during the debugging and testing phases. Traditional debugging techniques do not help either because they can change the behavior of the system. Therefore, a Distributed Object Visualization Environment (DOVE) framework can offer an alternative, less obtrusive, way to observe how a real-time system works at run-time. Thus, it can be a powerful tool in the real-time system development cycle.

This section describes how we have extended a distributed object visualization environment (DOVE) framework to monitor the timing behavior of a real-time application that generates and processes two separate streams of simulated sensor data events. The principal contributions of that effort are:

1. applying the DOVE framework to a realistic sensor-driven application,

2. extending the DOVE framework to support new application requirements, and

3. demonstrating and visualizing quality of service (QoS) control for multiple event streams within a real-time CORBA Event Service.

Software for visualizing real-time system behavior must address the following significant challenges that are not faced when visualizing the behavior of non-real-time systems. First, the visualization framework must not interfere with the correct timing behavior of the real-time system. Second, the framework must be flexible to address diverse system behaviors, particularly when sources of non-determinism appear. Finally, the framework must support both independent and correlated visualizations of distinct event streams. The reaminder of this section examines each of these key design forces.

Unobtrusive Visualization: As noted above, visualizing the behavior of a real-time sensor-driven system is a valuable engineering tool, particularly in the validation phase of the system lifecycle. However, information about real-time system behavior must be collected and displayed without interfering with the overall timing behavior of the system. For example, in an avionics mission-computing systems , monitoring and displaying behavioral information must not cause critical operations to miss their deadlines. Furthermore, excessive impact of the visualization on non-critical operations should be avoided, as well. The visualization mechanisms used to instrument the sensor-driven system must be efficient and relatively deterministic.

To reduce the load on the operational system, the DOVE browser and the visualization components usually run on a separate endsystem from the real-time portions of the system being monitored. To facilitate development and deployment of visualization components in such diverse client environments as monitoring workstations or web browsers, the DOVE browser and visualization components are written in JavaTM. Thus, the DOVE framework must decouple the timing behavior of the browser and visualization components from the timing behavior of the application, so that the application's real-time behavior is not adversely affected by the behavior of the Java Virtual MachineTM.

Visualizing Non-deterministic Behavior: Dynamically scheduled sensor-driven systems can produce non-deterministic behavior for certain operations when they are overloaded. For some dynamically scheduled sensor-driven systems, a low level of overload is an acceptable operating characteristic to maximize utilization of system resources.

Therefore, visualization software should be able to detect and provide a reasonable visualization of the load on the system, as well as the effects of overload on system behavior.

Visualizing Distinct Streams: Multi-sensor systems may produce distinct streams of sensor data. The real-time behavior of these systems often depends on the interactions between multiple streams. For example, two data streams may be processed at different priorities, e.g., processing for the higher priority stream may preempt processing for the lower priority stream. Likewise, there may be dependencies between the streams. Therefore, the visualization framework must consider data streams both individually and in the aggregate.

A significant part of the development effort for the ASFD program within which the experiments described in Chapter 8 were conducted, was the production of a flexible visualization capability for debugging and analysis. We used a DOVE [55, 31] architecture in which the primary components were

1. upcall monitor adapters that wrapped application components,
2. the metrics data cache that stored collected data,
3. a remote metrics logger that forwarded data to
4. an event channel that acted as the visiualization *agent* [55], and
5. a visualization browser and components written in Java^{TM}.

A sample display from the visualization framework is shown in Figure 7.4. This figure shows the effects of the handling of HRT (top) and soft real-time (SRT) (bottom) real-time deadlines by the Maximum Urgency First (MUF) [126] strategy during the flight simulator demonstration following the experiments described in Chapter 8. The panel on the upper left shows the regular pattern of critical deadlines that were successfully met, as expected. The top middle and top right panels show zero missed or cancelled critical deadlines, also as expected. The spikes in the bottom left window show a more variable pattern in the number of non-critical deadlines made, and the bottom middle and bottom right panels show respectively peaks of non-critical deadlines missed or cancelled. We note that this visualization approach allows system developers and testers to implement portable arbitrary combinations of instrumentation and visualization, which may in turn supplement platform-specific tools such as WindView^{TM}.

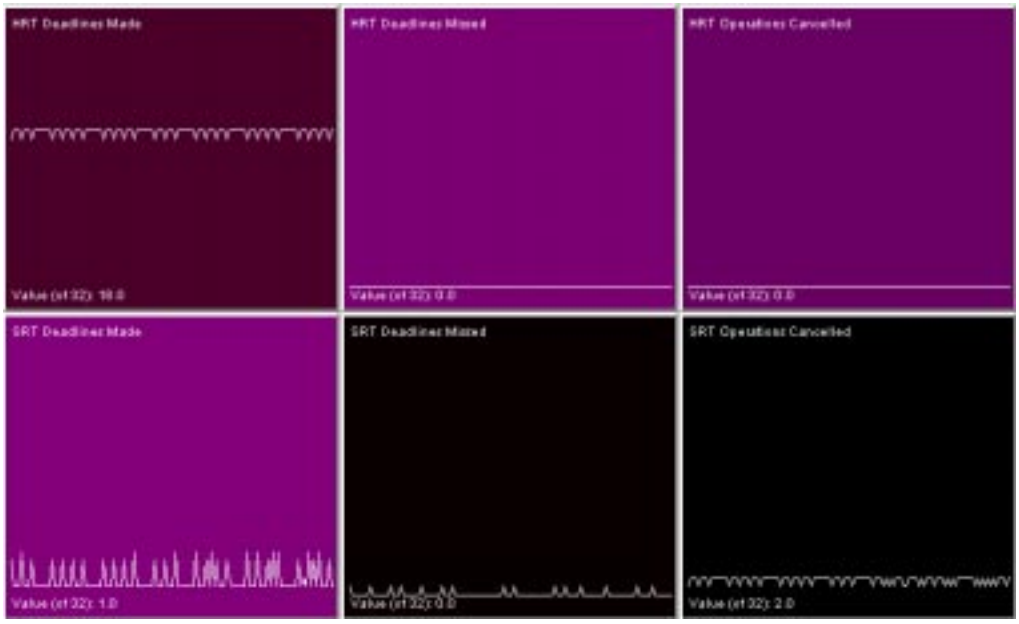

Figure 7.4: Metrics Visualizations

Chapter 8

Empirical Studies

This chapter describes empirical studies that justify an adaptive approach to scheduling. First, we quantify the framework's run-time dispatching behavior under each of several scheduling heuristics, in a realistic application under a range of load and load jitter conditions. We then identify key characteristics of those behaviors as they may be applied to the problem of reflective selection among several scheduling strategies at run-time.

This chapter is organized as follows: Section 8.1 describes the experimental platform used for these studies, including the application, middleware infrastructure, operating system, and hardware configurations. Section 8.2 describes the experimental design itself, including the hypotheses to be tested, the variables that were controlled, and the variables that were observed in these studies. Section 8.3 presents empirical results obtained on the described experimental platform. Section 8.4 examines the correlation between several kinds of information that can be observed or derived at run-time, and the performance of the scheduling heuristics in these studies. Finally, Section 8.5 presents conclusions that can be drawn from these studies, as well as open questions they raise.

8.1 Experimental Platform

The experimental platform, and the studies conducted on it, were supported under the Adaptive Software Flight Demonstration (ASFD) program, a contract research and development (CRAD) project 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, under Delivery Order 003 of the Weapon System Software Technology Support (WSSTS) contract, number F33615-97-D-1155.

The remainder of this section describes the constituent layers of the experimental platform. Section 8.1.1 defines several key terms and symbols. Section 8.1.2 describes the research avionics mission computing application used for these studies, and the middleware infrastructure that hosted the application. Section 8.1.3 describes the underlying operating system and hardware support.

8.1.1 Terminology

For clarity, we define the following terms prior to discussing the experimental platform itself:

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 heuristics using cancellation by a © annotation.

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 naturally implemented as a load chain in the experimental application, with each route segment successfully completed requesting the next segment, up to the length of the chain. In particular, a realistic application might declare the computation of the first one or two legs to be critical operations, that must be completed and cannot be cancelled, while subsequent route legs would likely be declared non-critical.

Bold Stroke: an avionics mission-computing domain-specific infrastructure built by Boeing on Common Object Request Broker Architecture (CORBA)-compliant Object Request Broker (ORB) middleware – *i.e.*, ADAPTIVE Communication Environment (ACE) and The ACE ORB (TAO) – on which the experimental application was hosted.

replication service: a middleware service provided by the Boeing Bold Stroke infrastructure for replicating data across mission-computing processors. Operation deadlines in the experimental application 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 experimental 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. In addition, following these experiments, the experimental application, middleware, and hardware was demonstrated in an AV-8B flight simulator at Boeing, which included an AV-8B cockpit and hardware remote terminals.

8.1.2 Experimental Application and Middleware

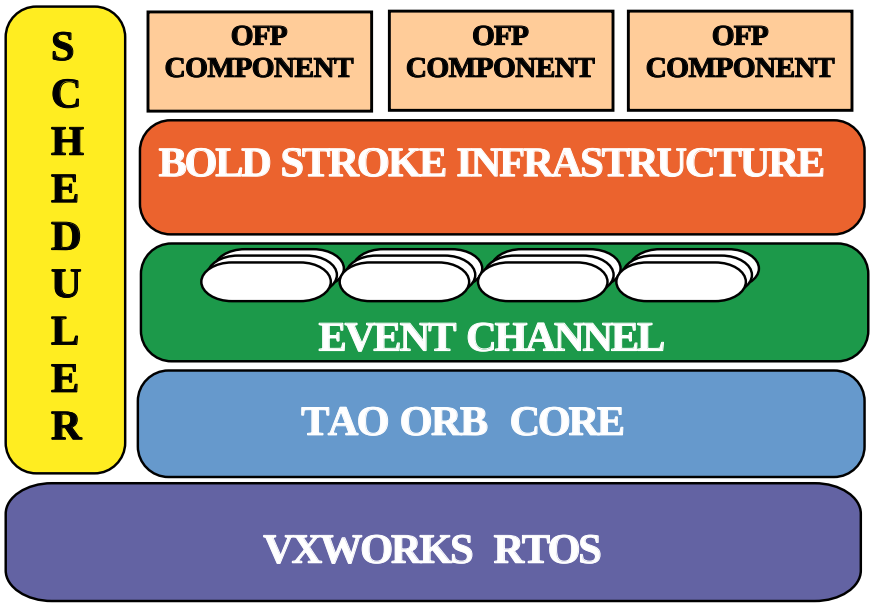

Figure 8.1: Application and Middleware Layers

We now consider the software architecture of the experimental application and supporting middleware infrastructure. As Figure 8.1 illustrates, the experimental application was hosted on middleware consisting of:

- the Bold Stroke avionics domain infrastructure [140, 120, 121, 24]
- TAO [13]

- the TAO Real-Time Event Channel (RTEC) [41]
- a strategized scheduler [32]

Application components, within which all avionics mission computing operations are performed, were hosted on the Bold Stroke infrastructure. Bold Stroke uses TAO's RTEC on the TAO ORB core to communicate both between components on the same endsystem requiring event-mediated interactions, and between components distributed across different endsystems. The scheduler maintains information required for priority-preserving dispatching, which in the experimental system was performed in dispatching queues within the TAO RTEC. The entire software architecture was hosted on the VxWorks [139] Real-Time Operating System (RTOS) on embedded hardware as described in Section 8.1.3.

The application with which these experiments were conducted is a research operational flight program (OFP) for avionics mission computing in an AV-8B aircraft. The baseline version evolved under the Open Systems Avionics Technology (OSAT) and OSAT-II [61] programs 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 (OSCAR) [16] airframe and the Boeing Bold Stroke middleware infrastructure [140, 120, 121, 24].

All of the major OFP components are implemented as periodically invoked operations, executed by event consumers. These operations were divided into two criticality equivalence classes: hard real-time (HRT) for critical operations, and soft real-time (SRT) for non-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, while 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. In these experiments we do not study adaptation among alternative operation rates, as described in Chapter 6, but rather compare scheduling heuristics over a given fixed assignment of rates. For the ASFD experimental platform, new SRT 20 Hz

functions were added to the OFP, including routes and steering components, as well as a digital map display.

To study the potential benefits and consequences of

1. supporting alternative scheduling strategies and
2. more importantly, working toward the ability to perform beneficial adaptation among them at run-time,

we ran identical trials of the experiment using each of the following three canonical scheduling heuristics: Rate Monotonic Scheduling (RMS) [70], Maximum Urgency First (MUF) [126], and RMS+Minimum Laxity First (MLF) [15]. RMS 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 is a hybrid static/dynamic strategy that assigns static priorities by operation criticality, and schedules within each static priority by minimum laxity. RMS+MLF is a further refinement of MUF, which schedules critical operations according to rate and non-critical operations at lower priority according to laxity.

We selected these strategies as most applicable to the chosen experimental application's requirements to support both HRT and SRT operations under the range of load and load jitter conditions studied. For applications with other characteristics, performing similar experiments with other scheduling strategies, instead of or in addition to the strategies we studied, might be indicated.

For each trial, the scheduler was configured with the appropriate strategy as described in Chapter 4, and the dispatcher was configured statically for that strategy as described in Chapter 5. In addition, separate trials for each strategy were run both with and without the operation cancellation capability described in Chapter 7. The metrics infrastructure described in Chapter 7 was also used to

1. capture time stamps,
2. record deadline success, failure, and cancellation counts during system execution,
3. deliver those data during open frames described in Section 8.2.1 to an external logger,
4. write collected data to disk from the logger, and

5. provide visualization events from the logger to a Distributed Object Visualization Environment (DOVE) [55, 31] browser.

8.1.3 OS and Hardware Configuration

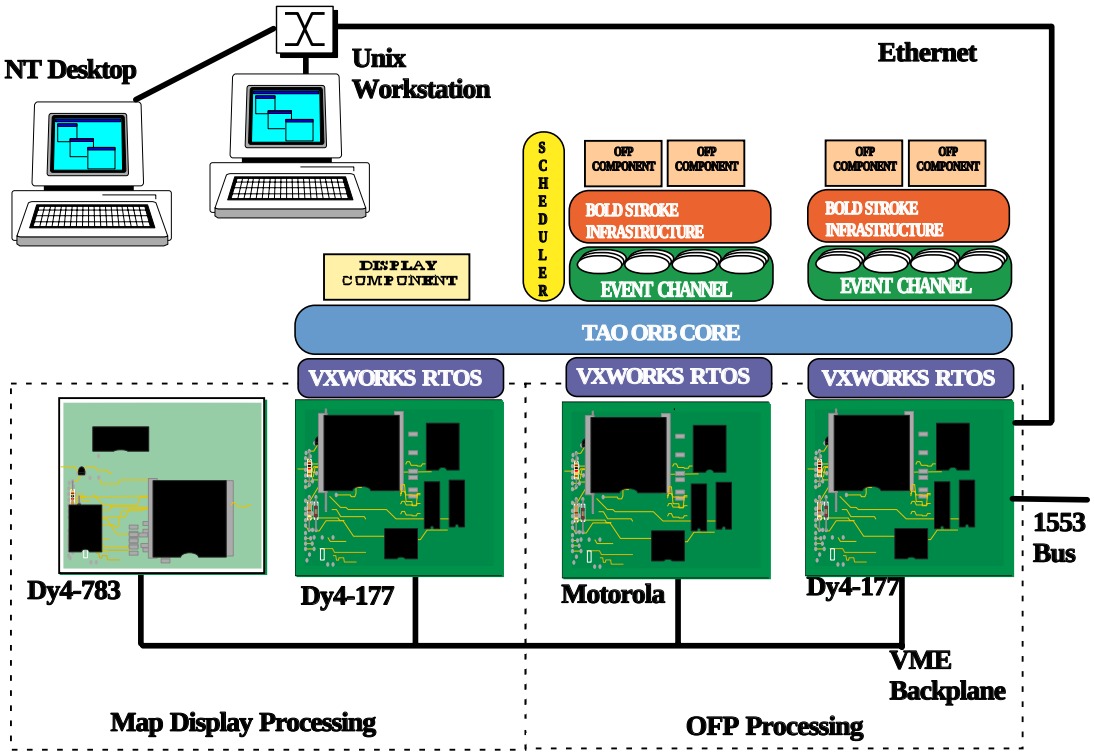

Figure 8.2: Hardware and Software Configuration

As Figure 8.2 illustrates, the ASFD demonstration hardware consisted 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 was used to gather metrics data and present visualizations of processor utilization and deadline successes, failures, and cancellations. The Unix workstation was used to load the executable programs onto the boards in the VME chassis, and as a file server for the digital map display.

Two processor cards, a Dy4-783 and a Dy4-177, performed the map display function. The Dy4-783 card had a memory-mapped display processor. The Dy4-177 card

hosted an application component that ran the map display algorithms and communicated with the OFP application components using the TAO ORB core.

The OFP application was distributed across the remaining two processor cards. The first card was a 200 MHz, PowerPC 604, Motorola card, which ran the experimental application and middleware described in Section 8.1.2 on the VxWorks [139] RTOS. The Motorola card was responsible for much of the OFP's logical operation, including computing route legs for navigation. This card also contained tasks used to simulate dynamic variability in the OFP, as described in Section 8.2.1, and was scheduled in each experimental trial using one of the scheduling heuristics described in this chapter.

The Motorola card used as many as eight individual threads to carry out the execution of the OFP. There were up to five threads in the TAO RTEC, used to service the appropriate dispatching queues configured for the scheduling strategy in each particular trial. For MUF, two threads in descending priority order were used for the HRT and SRT operation dispatches, respectively. For RMS, four threads in descending priority order were used for the 20 Hz, 10 Hz, 5 Hz, and 1 Hz dispatches, respectively. For RMS+MLF, five threads in descending priority order were used for the 20 Hz HRT, 10 Hz HRT, 5 Hz HRT, and 1 Hz HRT, and SRT dispatches, respectively.

The other three threads were a 20 Hz ORB thread, a 40 Hz reactor thread and the main thread. The main thread was used for remote metrics reporting. The threads' priorities were set with the 40 Hz reactor thread at highest priority, the 20 Hz ORB thread and highest priority dispatching thread having the same (next highest) priority, and then subsequent dispatching threads at descending priorities, and finally the main metrics thread at the lowest priority.

The second 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.

At each 20 Hz frame, a one-way call was used to send an event from the Dy4-177 OFP card, indicating arrival of a new set of 20 Hz input data via the 1553 bus, from sensors throughout the aircraft. The 20 Hz ORB thread on the Motorola card received this event and was responsible for starting execution of operations in that frame. The

data was unpacked into a local database and local data ready events were pushed to the event channel dispatching threads.

A 40 Hz reactor thread on the Motorola card was used to detect each end of frame timeout. When the timer expired, data produced by the operations on the Motorola card for that frame was flushed to the replication service and an event indicating availability of that data was sent back to the 100 MHz PowerPC Dy4-177 card. In addition to the middleware-level threads described above, the VxWorks networking task was set to the highest priority of all, so that when network communication was possible it would occur immediately, *i.e.*, when the data was flushed to the replication service, and the data ready event was sent to the Dy4-177 card.

Therefore, the end-of-frame deadline represented a hard deadline for completion of all operations whose output is critical to the correct operation of the system. Any operation whose output is not critical could miss this deadline without compromising the system, but its newest output simply would not be available at the end of that frame. Depending on the kind of non-critical output missing, a previous value could be used, or processing that depended on the output could be deferred until the output became available. A key implication of this style of processing is that critical processing must never depend on the output of processing whose completion before that deadline is assured.

8.2 Experimental Design

We now describe the experimental design itself, including the hypotheses tested, the variables that were controlled, and the variables that were measured in these studies. Two hypotheses were explored in these trials:

1. that efficiency and effectiveness of any given scheduling heuristic are functions of run-time factors, *i.e.*, load and load jitter, and

2. that reflection on measurable information at run-time can yield derived information of potential use to guiding adaptation between scheduling heuristics.

The remainder of this section is structured as follows: Section 8.2.1 describes the variables that were controlled in these experiments, notably

- the number of operation dispatches requested

- the jitter in the offered load

- whether or not cancellation was enabled for SRT operations

Section 8.2.2 describes the variables that were measured, and how those measurements were integrated into the experimental design to avoid interference with control of the experiment itself.

8.2.1 Controlled Variables

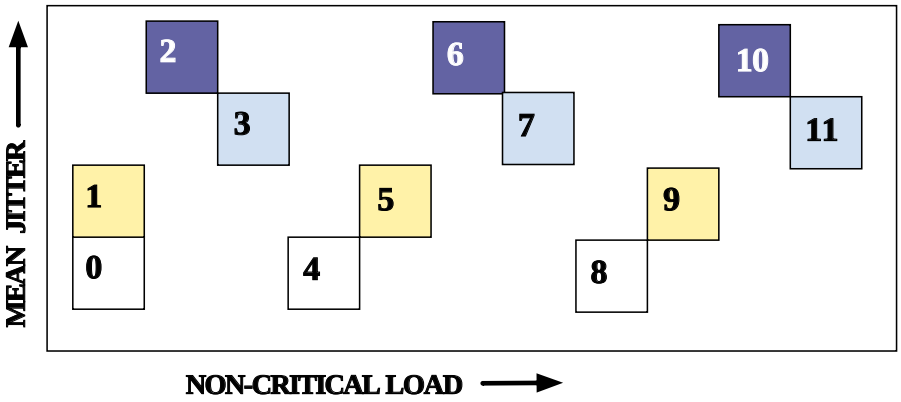

Figure 8.3: Operating Regions

To examine effects of varying load and load jitter in a realistic avionics mission computing environment, we added operations to a sequence of twelve epochs of operation, each representing what we term a distinct *operating region* [72], numbered 0–11, as shown in Figure 8.3.

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 to each operating region, with the length of the chain of requests varying to move from lowest to highest *fundamental* non-critical load from region 1 to region 11, while keeping the fundamental critical load constant across operating regions. 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,

an additional HRT event consumer was added 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 kinds of 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)

We kept the same fundamental load in region 0 and region 1 to create a case where we could examine the effects of varying jitter, with load held constant. We note that with the cycling of jitter ranges every four operating regions, there were at least two cases for each jitter range where load varied but the jitter range was the same.

Execution time variability within each range was implemented using a pseudo-random sequence initialized using the same seed for each heuristic. Also, the system was set to move to the next operating region every 150 seconds in each trial. Thus, the same profile of load and load jitter was applied for each heuristic, allowing direct comparisons of trials for different heuristics. Table 8.1 shows how the HRT execution

Table 8.1: Loads For Each Operating 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

variability and additional SRT loads were combined in 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). Finally, 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). Thus, total variability was 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. Furthermore, the *range* of variability was lowest in regions 0, 4, and 8, was comparable in regions 1, 3, 5, 7, 9, and 11, and was highest in regions 2, 6, and 10.

As Section 8.1.2 described, each of the scheduling heuristics 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 (WCET) exceeded the time remaining before its deadline at the point of upcall, as described in Chapter 7.

The route leg update operation was registered as both an event consumer and event supplier. When the route leg update event consumer routine is called, it updates one route leg and then if there are remaining steps in its computation chain (according to the chain length for the current region, as described in table 8.1), pushes a SRT event to be consumed if needed. Therefore, if a SRT event to the route leg update consumer is cancelled, additional SRT events are not pushed to the event channel even if the mode indicates that there should be additional updates.

Since 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 and any subsequent route legs computed in that chain would likely have erroneous inputs and outputs. Shedding the route leg load chain at the first missed deadline thus removes operations that would otherwise consume CPU time without adding utility. Therefore, the above cancellation policy 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.

8.2.2 Measured Variables

To measure the effects of varying load and load jitter described in Section 8.2.1, we instrumented the application and middleware using an earlier version of the metrics infrastructure described in Chapter 7. We collected three kinds of information:

- latency of dispatching enqueue and dequeue actions
- counts of missed, made, and cancelled operation deadlines
- 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 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, and
3. neither collection nor extraction of data interferes with the real-time constraints of the system itself.

To achieve this, we have first optimized the data probes and cache for both efficiency and flexibility, as described in Chapter 7. Second, we have leveraged the existing phasing of application operations to provide regular windows of reduced contention for the CPU, in which to extract collected data. Figure 8.4 shows the resulting framing of operations in the executing OFP. In general this framing is engineered for improved real-time behavior, as it might be in a production OFP:

- frame periods are harmonic
- initiation of requests is staggered to reduce contention (*i.e.*, avoiding the canonical critical instant for as many operations as possible).

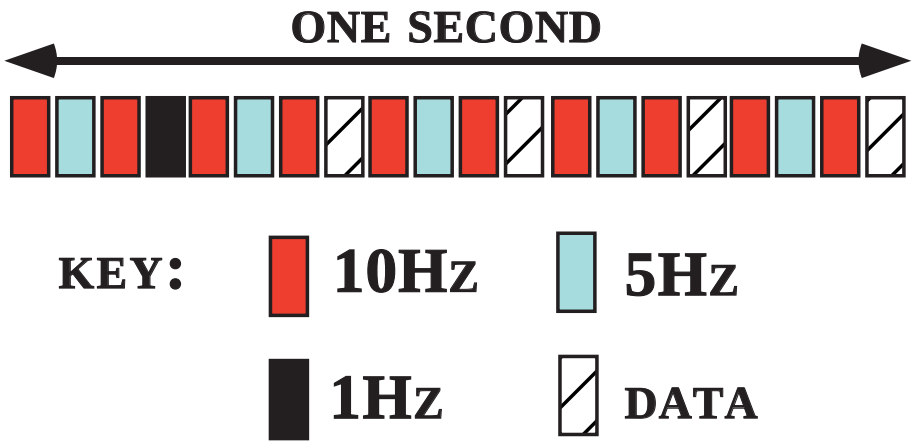

Figure 8.4: Framing of Operation Requests and Metrics Data Extraction Points

8.3 Observed Results

The collected data reveal five key areas of information for adaptive and reflective scheduling in middleware:

1. the dispatching load and the latency of the dispatching infrastructure itself
2. the total number of operation deadlines missed, made, and cancelled for each of the six heuristics examined (*i.e.*, RMS, MUF, and RMS+MLF each with and without cancellation of SRT operations)
3. canonical examples of missed HRT deadlines across several heuristics
4. differences in efficiency and effectiveness of the heuristics, particularly with and without cancellation
5. observable characteristics that are correlated with the effectiveness of particular scheduling heuristics in specific operating regions

This section discusses each of these kinds of information. Section 8.3.1 examines the measured dispatching load and latency, and presents an analysis of deduced overhead. Section 8.3.2 presents measured operation deadline success, failure and cancellation data, and compares the behavior of the different strategies. Section 8.3.3 makes a closer examination of the canonical cases where HRT deadlines were missed. Section 8.3.4

compares dispatching efficiency and effectiveness, particularly for strategies where cancellation had an effect, and compares effectiveness of all strategies across all operating regions. Finally, Section 8.3.5 summarizes these results.

8.3.1 Dispatching Load and Overhead

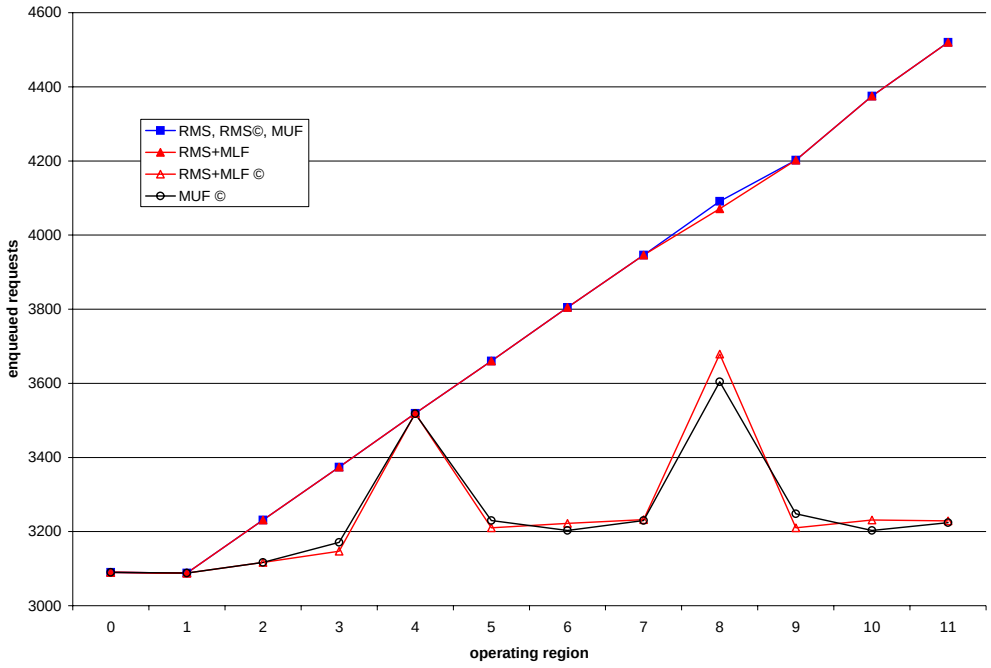

Figure 8.5: Total Requests Enqueued

Figure 8.5 shows effective load on the system with each scheduling heuristic (*i.e.*, the total number of requests enqueued), in each of the operating regions. Scheduling heuristics using operation cancellation are indicated by a © annotation. MUF with cancellation and RMS+MLF with cancellation enqueued fewer dispatch requests overall due to the effects of cancellation on the chains of operations described in Section 8.2.1: when one operation of a chain is cancelled, subsequent requests for that operation are not made. The other heuristics, RMS MUF and RMS+MLF all without cancellation, and RMS with cancellation, enqueued a total number of dispatch requests that rose linearly from around 3100 in regions 0 and 1 to above 4500 in region 11.

Figures 8.6 and 8.7 show respectively the total latency of enqueuing and dequeuing requests in each operating region. The MUF and RMS+MLF strategies with

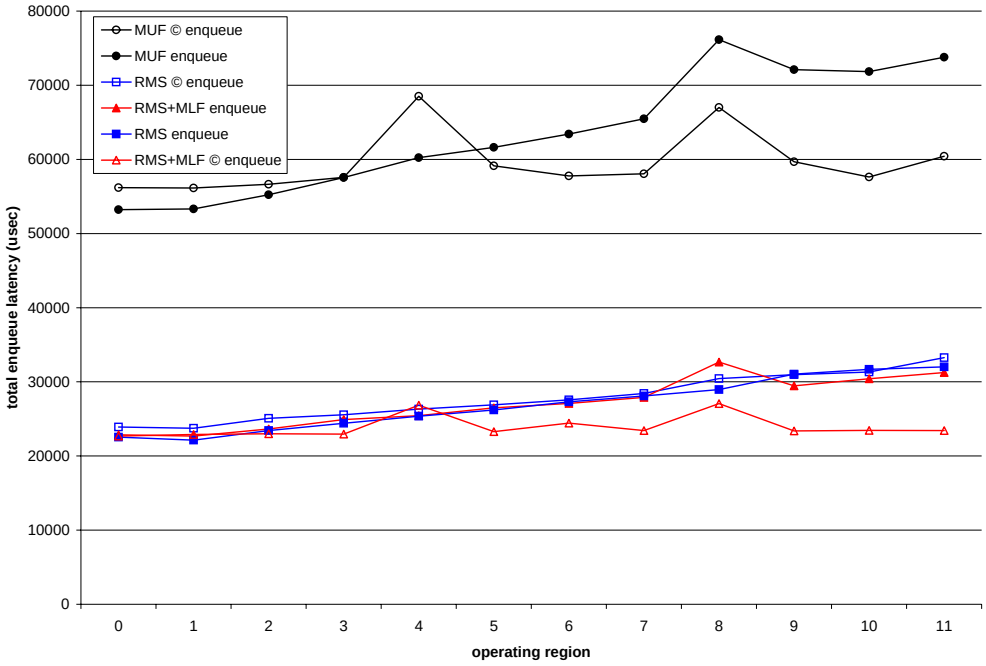

Figure 8.6: Total Enqueue Latency

cancellation limited the number of SRT operations attempted, and thus did not show increasing total latency as the lengths of the route leg chains grew. The other strategies did not perform any cancellation, and showed a gradual increase in total latency with the number of requests enqueued and dequeued.

Figures 8.8 and 8.9 show respectively the mean enqueue and dequeue latencies for each strategy in each of the operating regions. As in Figures 8.6 and 8.7, enqueue calls showed higher latency than dequeue calls. The MUF and MUF with cancellation strategies had the highest mean enqueue and dequeue latencies, with lower latencies for RMS MUF and RMS+MLF all without cancellation, and RMS with cancellation.

The most important feature of these plots is that the mean enqueue and dequeue latencies did not rise significantly with increasing load or variations in jitter, so that including preemption and jitter delays, the combined average queueing latency in each strategy:

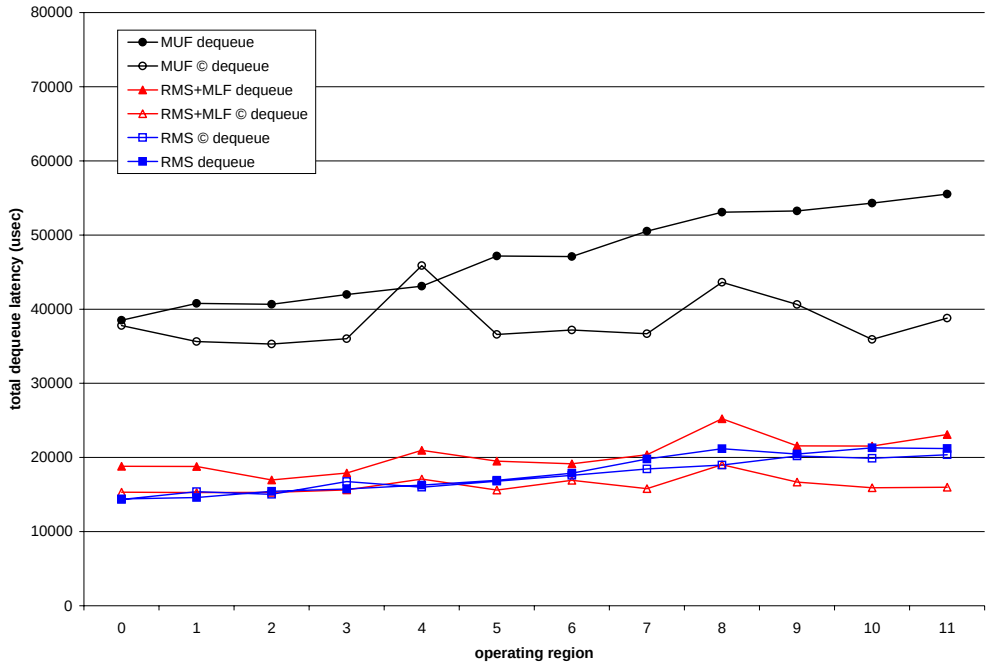

Figure 8.7: Total Dequeue Latency

1. took around 12 μsec per dispatch request for RMS and RMS+MLF, took around 32 μsec per dispatch request for MUF, and

2. for each strategy remained comparable across operating regions.

Although the queue latency values shown in Figures 8.8 and 8.9 include preemption, we can argue that the preemption effect is essentially constant over a sufficient number of data samples, based on the phasing of dispatch requests illustrated in Figure 8.4 above. Therefore, it is reasonable to claim that each latency value is, within a scalar constant that may be different for each strategy and region, reflective of the actual overhead of the enqueue and dequeue method calls. Because the pseudo-random execution jitter introduced in each region is evenly distributed, the effects of this jitter on enqueue and dequeue latency can also be assumed to be constant over a large enough sampling interval. Here, we assume the interval spent in each region is sufficiently large, which is supported by the discussion of *operation* latency in Section 8.4.2.

Figures 5.7 and 5.6 in Section 5.4.3 illustrated several key effects of dynamic queue management and queue length on the overhead incurred when enqueuing or dequeuing requests:

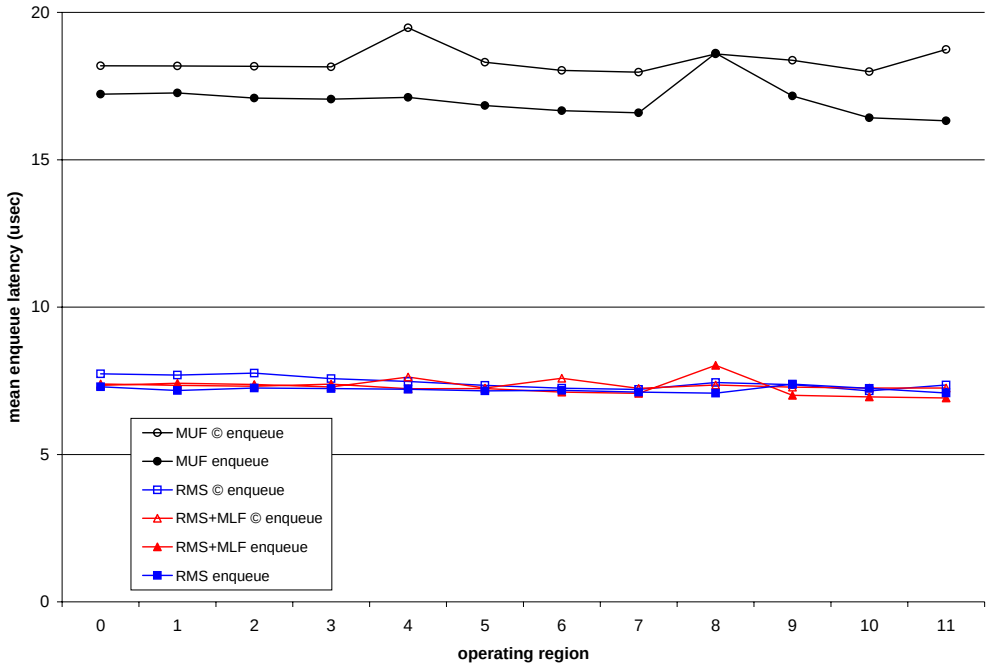

Figure 8.8: Mean Enqueue Latency Per Operation

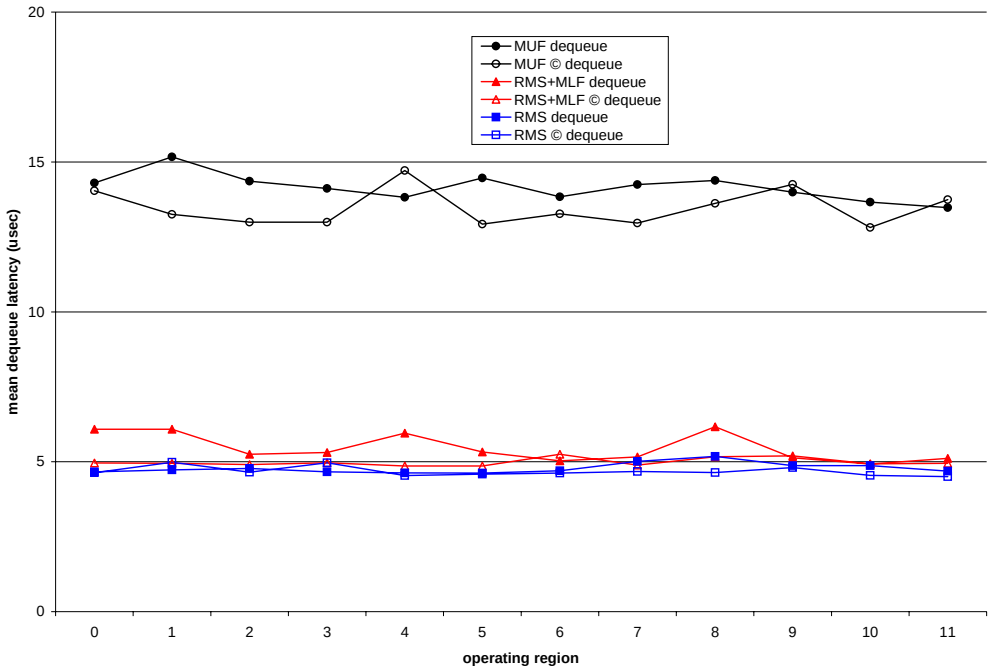

Figure 8.9: Mean Dequeue Latency Per Operation

- enqueue actions were more expensive than dequeue actions in both statically and dynamically managed queues
- queues managed dynamically according to deadline laxity showed greater overhead than statically managed queues for both enqueue and dequeue actions
- the dequeue overhead curve for both static and dynamic queues saturated rapidly and grew very slowly afterward
- enqueue overhead for both static and dynamic queues increased with the number of enqueued operations (*i.e.*, the queue length)
- the difference in enqueue overhead between the static and dynamic queues increased with the queue length as well

To identify where on the overhead curves shown in Figures 5.7 and 5.6 the experimental application was operating, we need a more accurate assessment of the actual overhead imposed on each dispatch request by the queues. Therefore, we would naturally like to factor out the effects of preemption and jitter and obtain the actual overhead imposed by each queue itself.

Because the highest priority dispatching queue in each heuristic is not subject to preemption from operations in other queues, we can limit the effect of preemption to two possible sources: the higher priority VxWorks network task and 40 Hz reactor task described in Section 8.1.3. Furthermore, interruption by the 40 Hz reactor task would be indicated by a missed critical deadline for a dispatch being enqueued or dequeued in the highest priority dispatching queue. While spurious preemption by the VxWorks network task is possible, in general the system is very closed and network traffic is limited to the extent possible – only the VME bus connected to the Motorola card where scheduling occurred could have impacted these measurements.

Figures 8.10 and 8.11 show respectively the enqueue and dequeue latency for the highest priority queue of MUF, RMS, and RMS+MLF in each operating region. We note that in operating regions 0, 4, and 8 there was no added jitter, so the latency measures in those regions accurately reflect the enqueue and dequeue overheads of each highest priority queue. Furthermore, no significant additional latency was observed in the other operating regions, so all data points in Figures 8.10 and 8.11 can be said to reflect accurately the actual overhead incurred in the highest priority queue for each heuristic.

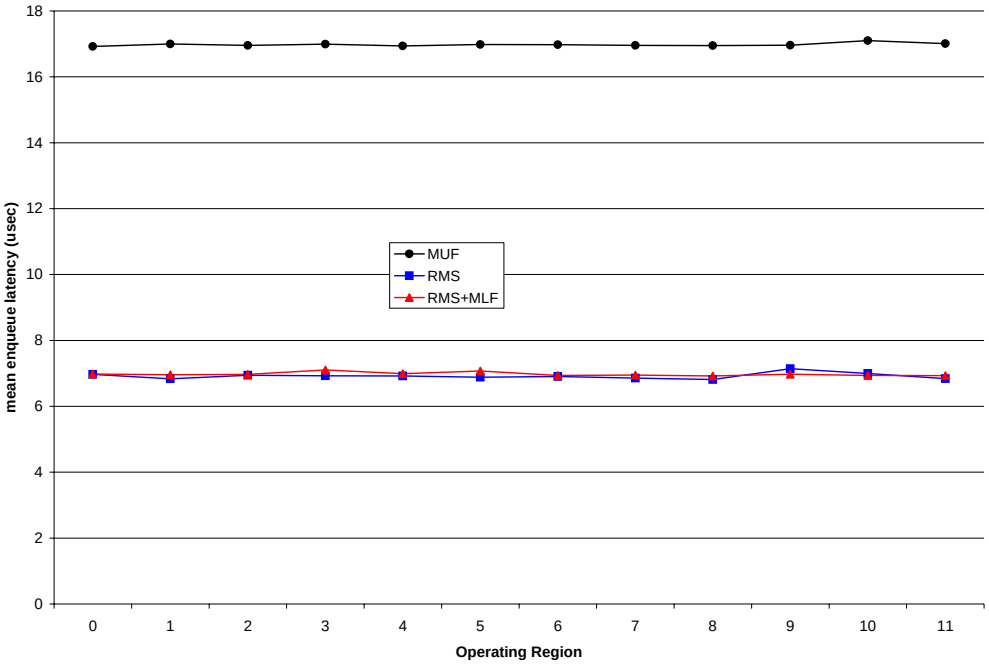

Figure 8.10: Mean Enqueue Latency of Highest Priority Queue

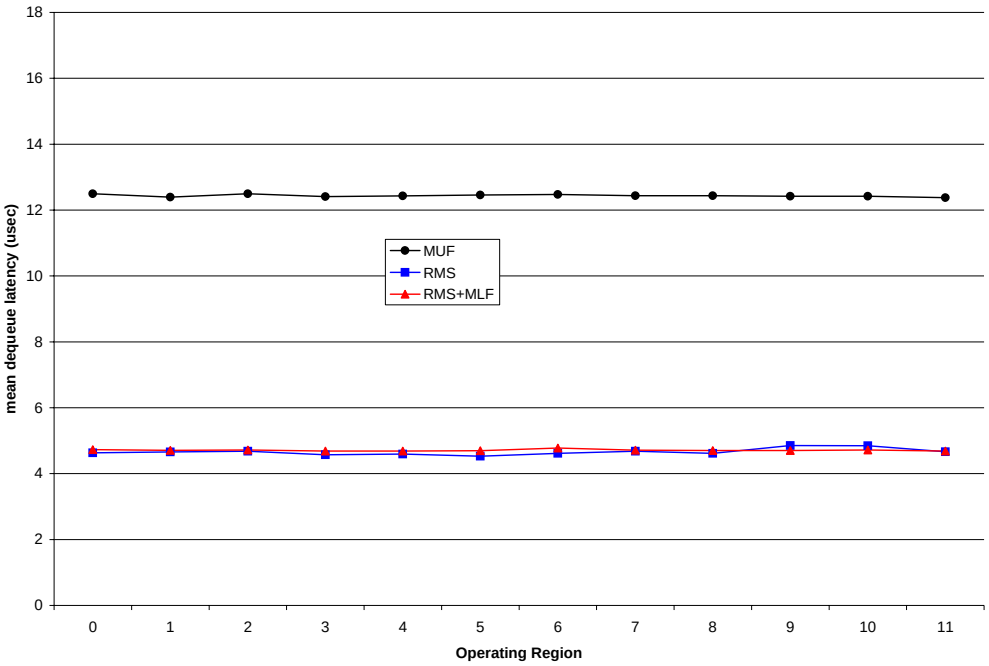

Figure 8.11: Mean Dequeue Latency of Highest Priority Queue

The enqueue methods for statically and dynamically managed queues perform similar functions, that differ only by a scalar value under any particular queue length and insertion position. The static queue uses a fixed subpriority field of each enqueued request to place requests in fixed subpriority order. In the case where static subpriorities all have the same value this simply produces a FIFO ordering of requests.

The dynamic queue on the other hand uses both a deadline field and a WCET field to order operations by laxity, and the case of FIFO ordering although possible by setting all deadline and WCET fields to the same value is trivially improved upon by instead using a static queue. Because the dequeue methods of dynamically managed queues address the problem of requests aging past their latest feasible dispatch times while still enqueued, enqueue methods need only ensure that each new request is added at a correct laxity-monotonic position. Thus, simply subtracting the request's WCET from its deadline produces a correct ordering, without requiring an additional system call to obtain the current time.

Since the highest priority queue in MUF is managed dynamically according to laxity, and the highest priority queue in each of RMS and RMS+MLF is managed statically, we can compare static and dynamic enqueue costs by subtracting the RMS+MLF or RMS latency from the MUF latency. The average additional overhead of subtracting and comparing time fields versus doing simple integer comparisons is thus approximately 10 μsec in each operating region.

While the dequeue method for a statically managed queue simply removes the dispatch request at the head of the queue, a dymanically managed queue must first check that the request is still valid. When requests at the head of the queue have aged past the point where they can be successfully dispatched, a dynamically managed queue must move through those requests to find the first operation that has not aged out. In region zero, no operations were cancelled or missed their deadlines, so it is safe to say none aged out in the queues. Therefore, the measured dequeuing overhead represents the best-case performance for both the static and dynamic queues.

Furthermore, no additional dequeue latency was observed in the other operating regions, so all data points in Figure 8.11 represent best case behavior for the particular strategy's highest priority queue. Again comparing MUF to RMS and RMS+MLF, we identify an increase in best-case average overhead of 8 μsec for dynamically managed queues, compared to statically managed queues.

Notably, although the number of operation requests managed differed between the RMS highest priority queue and the RMS highest priority queue, they showed no significant difference in overhead. RMS managed both HRT and SRT 20 Hz requests in its highest priority queue, while the highest priority queue in RMS+MLF only managed HRT 20 Hz requests. We attribute this effect to a moderate number of enqueued requests at any given time, which is further supported by the fact that highest priority enqueue and dequeue latencies did not rise measurably with increasing load in subsequent operating regions.

We also study the effects of preemption directly in the experimental application by comparing the lowest priority queue in each of the MUF and RMS+MLF strategies. In both MUF and RMS+MLF, all SRT requests are managed by a single lowest-priority queue managed by laxity.

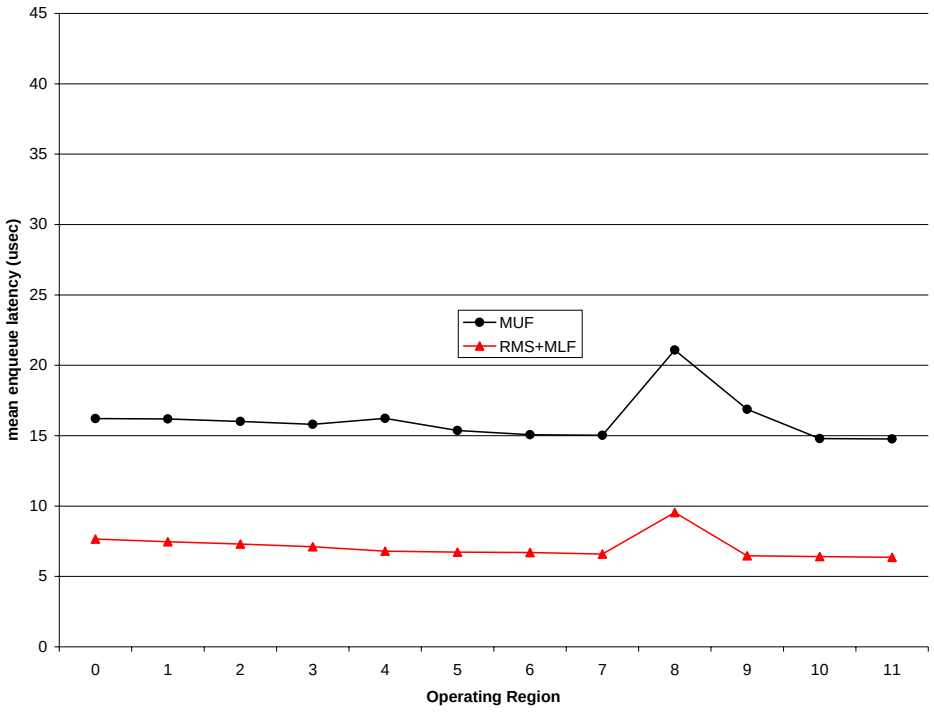

Figure 8.12: Mean Enqueue Latency of Lowest Priority Queue

Figures 8.12 and 8.13 show respectively the enqueue and dequeue latency for the lowest priority queue of MUF and RMS+MLF in each operating region. These plots show an amortization effect as preemption delays are averaged over an increasing

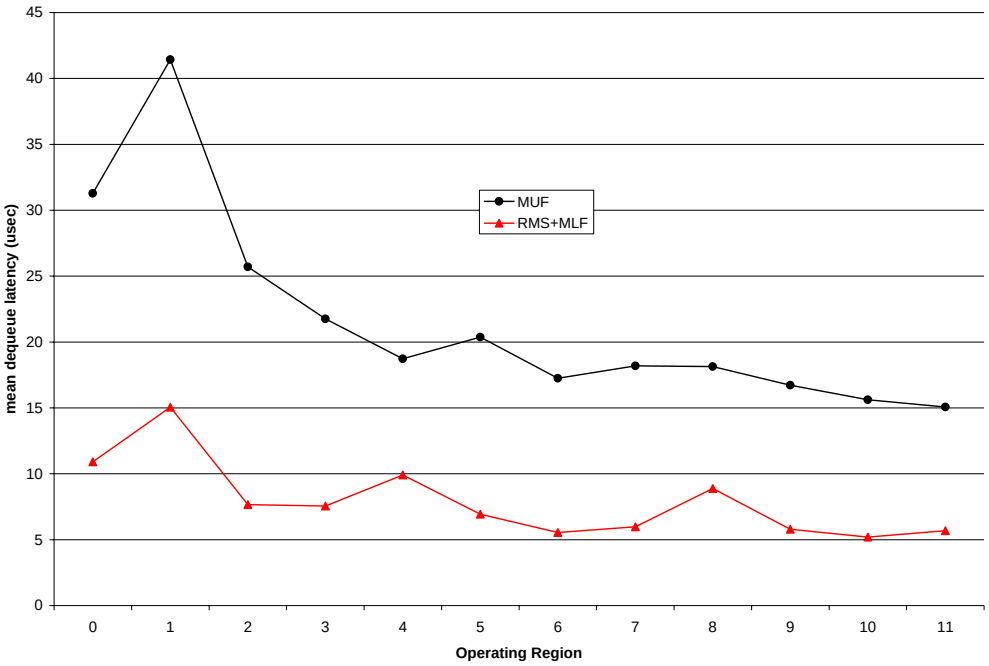

Figure 8.13: Mean Dequeue Latency of Lowest Priority Queue

number of requests in subsequent operating regions. We note that enqueue latency was not significantly affected by preemption, and attribute this effect to the fact that in the experimental application most events for a given frame are enqueued at once at the beginning of the frame, and only the events pushed to subsequent operations in load chains are likely to be preempted by executing operations.

Dequeue latencies on the other hand showed a marked increase in average latency over the fundamental queue overhead for laxity queues: 2–11 μsec for RMS+MLF and 12–37 μsec for MUF. Finally, we note the large difference in latency between for RMS+MLF and MUF with the same kind of queue managing the same number of SRT requests. We attribute this difference to lower overhead incurred in RMS+MLF than in MUF, although it might also be due in part to a possibly more favorable *availability function* [70] for the lowest priority SRT requests, with HRT requests managed in RMS+MLF using RMS rather than in MUF using MLF. Additional experiments recording exact preemption ordering of request dispatches and enqueue and dequeue actions in all queues are thus indicated, though we defer those to future work.

8.3.2 Operation Deadline Success, Failure and Cancellation

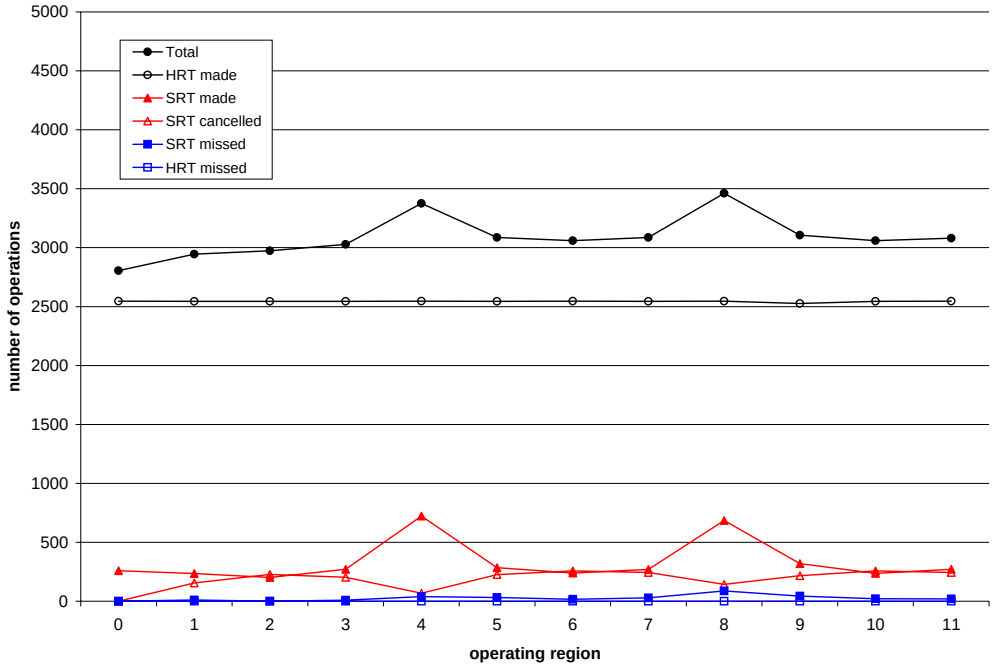

Figure 8.14: MUF Operation Behavior With Cancellation

Figures 8.14 and 8.15 show the total number of HRT and SRT operation deadlines made, missed, and cancelled for the MUF strategy. Figure 8.14 shows MUF *with* cancellation, and Figure 8.15 shows MUF *without* cancellation. It is instructive first to compare the slope of the top curve in each of these graphs, indicating the increase in the total number of dispatch requests in subsequent operating regions. In Figure 8.15 the slope of the total requests curve is similar to that shown in Figure 8.5, though the curve is slightly lower as some dispatch requests are for internal dependency correlations in the event channel, and not for application operations. Without cancellation, the total operation load in MUF was thus proportional to the number of enqueued requests.

In Figure 8.14, the slope of the total requests curve was much less than in Figure 8.15, 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 was very successful in reducing the number of operation deadlines missed though it also resulted in a lower number of operation deadlines made. Both with and without cancellation, MUF

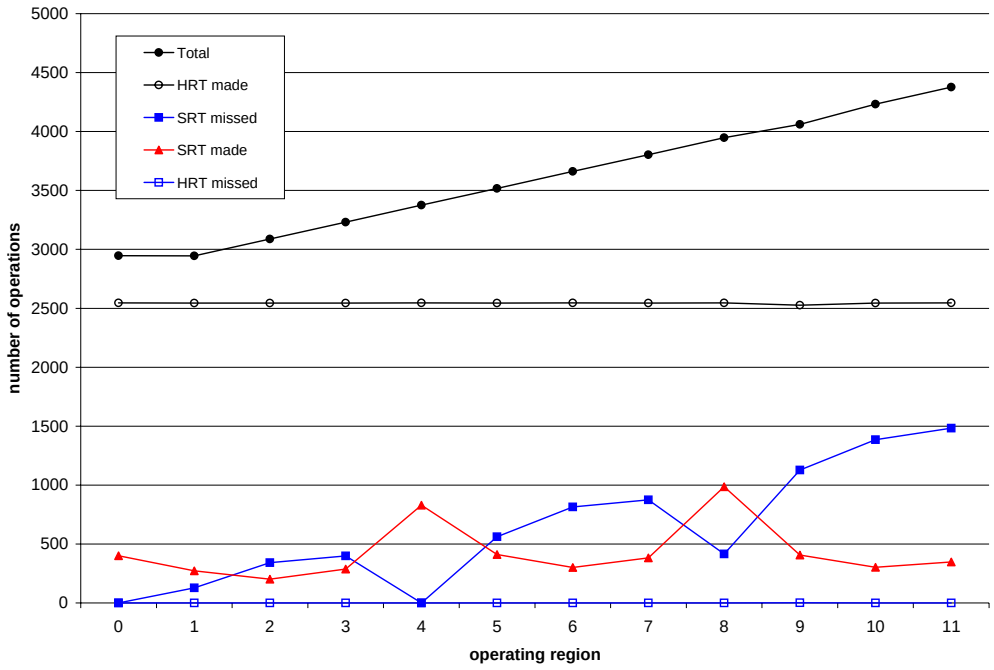

Figure 8.15: MUF Operation Behavior Without Cancellation

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.

Figures 8.16 and 8.17 show the total number of HRT and SRT operation deadlines made, missed, and cancelled for the RMS+MLF strategy. Figure 8.16 shows RMS+MLF *with* cancellation, and Figure 8.17 shows RMS+MLF *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.

Figures 8.18 and 8.19 show the total number of HRT and SRT operation deadlines made, missed, and cancelled for the RMS strategy. Figure 8.18 shows RMS *with* cancellation, and Figure 8.19 shows RMS *without* cancellation. Both RMS with cancellation and RMS without cancellation show a total operation load similar to that of MUF without cancellation and RMS+MLF without cancellation. Both RMS with cancellation

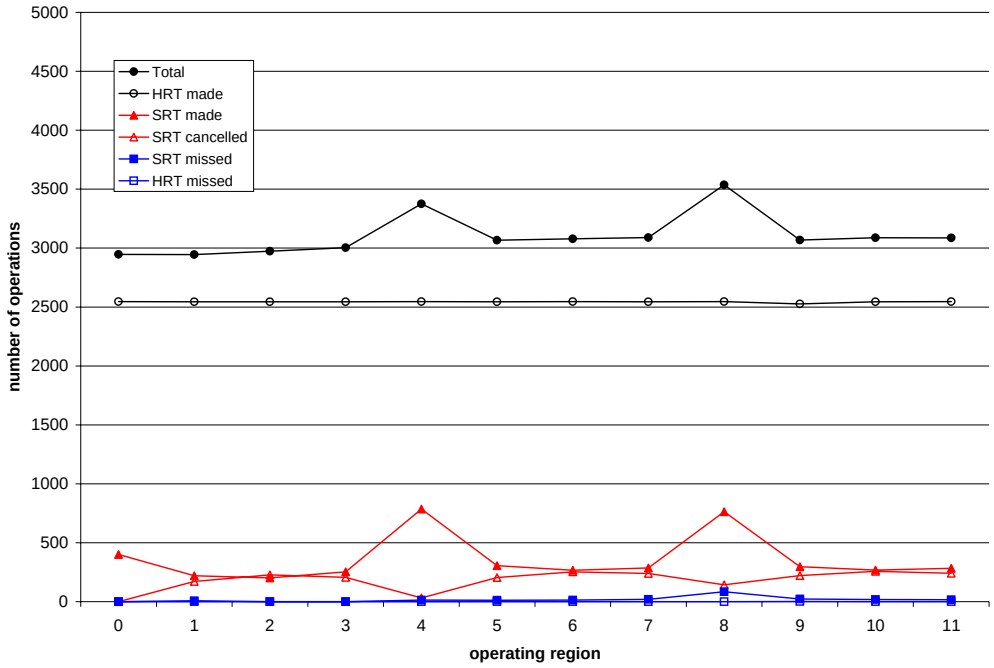

Figure 8.16: RMS+MLF Operation Behavior With Cancellation

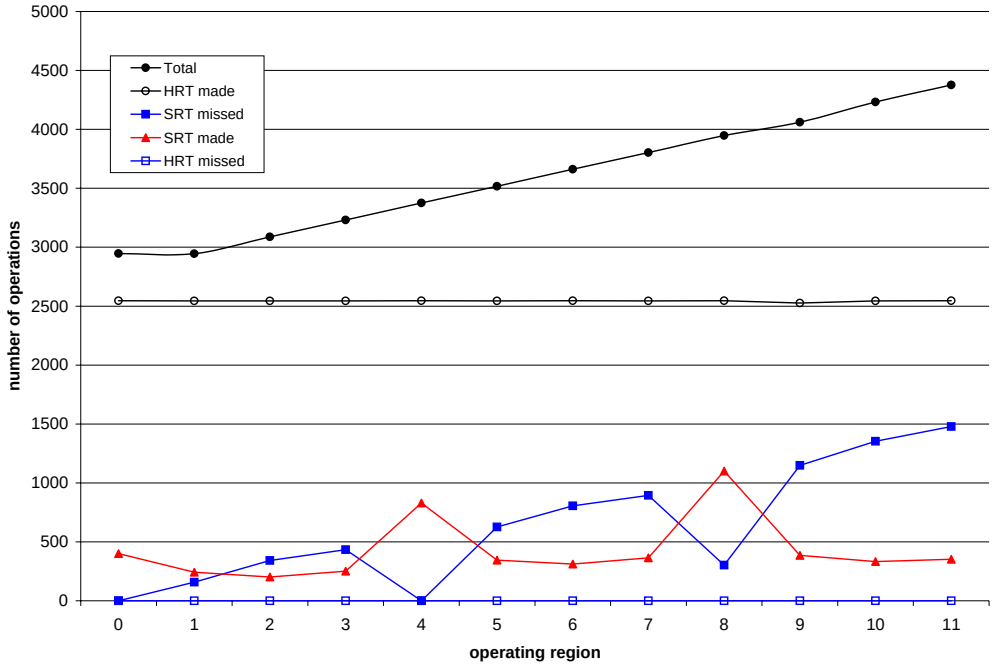

Figure 8.17: RMS+MLF Operation Behavior Without Cancellation

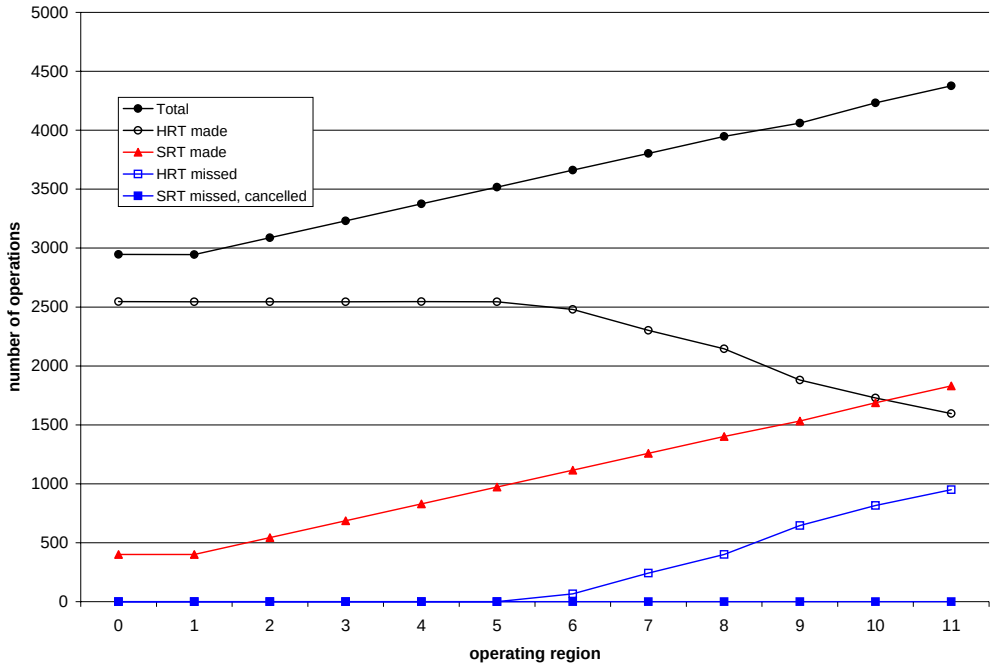

Figure 8.18: RMS Operation Behavior With Cancellation

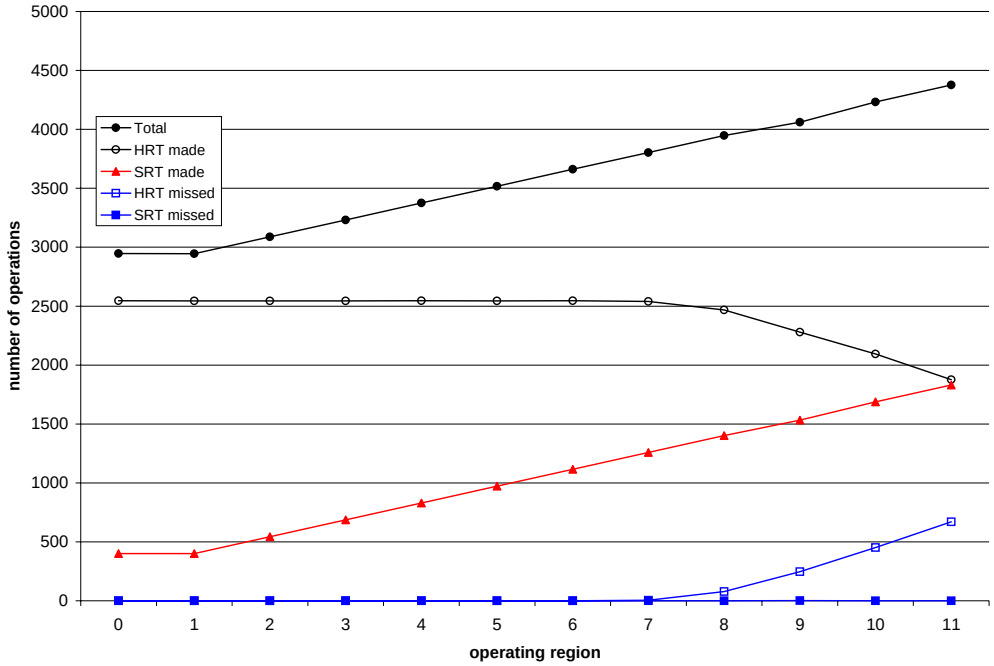

Figure 8.19: RMS Operation Behavior Without Cancellation

and RMS without cancellation show a significant number of missed HRT deadlines 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. We interpret these observations as indicating the impact of cancellation overhead on overall feasibility.

Interestingly, with RMS in this application, adding cancellation had no apparent benefit at all, and in fact showed a greater number of missed HRT deadlines and a lower number of made HRT deadlines, 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.

The *critical instant* [70] simulation results presented in Section 4.6 are instructive, particularly Figures 4.9 and 4.11 which show that 20Hz SRT operations under the RMS priority assignment may be expected to receive preferential quality of service (QoS), with lower latency and few or no missed deadlines compared to other operations. Accordingly, even with the increasing length of the 20 Hz SRT rout leg chains in successive operating regions, RMS with cancellation did not in fact cancel even a single SRT request. Only HRT requests were detected as being in danger of missing their deadlines, but since the cancellation policy did not allow HRT operations to be cancelled, no load shedding was performed. Thus, the only effect of using cancellation in RMS was to add overhead, resulting in the first missed HRT deadlines occurring in an earlier, less loaded region, and more missed HRT deadlines overall.

8.3.3 Missed HRT Deadlines

A closer examination of the missed HRT deadlines in RMS with and without cancellation is instructive. We look first at the transition from feasible to infeasible load conditions, which occurs first for RMS without cancellation, and then for RMS with cancellation, when moving through operating regions 6, 7, and 8. Second we examine the case of a single missed HRT deadline in both the RMS+MLF strategy with cancellation, and the MUF strategy without cancellation.

Each data sample contains a sequence of statistics taken for a number of operations, and a sequence of samples is taken in a particular operating region. We denote:

- the ith operating region by $\mathcal{R}_i$
- the number of samples in operating region $\mathcal{R}_i$ by $|\mathcal{R}_i|$
- the jth sample in operating region $\mathcal{R}_i$ by $\mathcal{S}_{(j,i)}$
- the number of operation statistics in sample $\mathcal{S}_{(j,i)}$ by $|\mathcal{S}_{(j,i)}|$
- the kth operation statistic in sample $\mathcal{S}_{(j,i)}$ by $\mathcal{O}_{(k,j,i)}$

To examine the deadline success of each operation in the experimental application, we note that for a given statistic in a given sample, four factors are important: the criticality of the operation, the number of deadlines made, the number of deadlines missed, and the number of dispatches that were cancelled. We thus represent an operation statistic as a 4-tuple:

$$\mathcal{O}_{(k,j,i)} = \langle critical_{(k,j,i)}, made_{(k,j,i)}, missed_{(k,j,i)}, cancelled_{(k,j,i)} \rangle \tag{8.1}$$

where $critical_{(k,j,i)}$ is a boolean value that is true if and only if the operation is HRT. The values of $made_{(k,j,i)}$, $missed_{(k,j,i)}$, and $cancelled_{(k,j,i)}$ are from the natural numbers[1] and represent the number of deadlines made, missed, and cancelled respectively in operation statistic $\mathcal{O}_{(k,j,i)}$.

We use Iverson's bracketed predicate valuation operator [43, 36] to sum over only the operation statistics of interest, *i.e.*, $[\neg critical_{(k,j,i)}]$ is 0 for a HRT operation statistic and 1 for a SRT operation statistic. We first define helper functions srt_made, srt_missed, $srt\_cancelled$, and $srt\_fraction$ over a sample $\mathcal{S}_{(j,i)}$:

$$srt\_made(\mathcal{S}_{(j,i)}) = \sum_{k=1}^{|\mathcal{S}_{(j,i)}|} made_{(k,j,i)}[\neg critical_{(k,j,i)}] \tag{8.2}$$

$$srt\_missed(\mathcal{S}_{(j,i)}) = \sum_{k=1}^{|\mathcal{S}_{(j,i)}|} missed_{(k,j,i)}[\neg critical_{(k,j,i)}] \tag{8.3}$$

$$srt\_cancelled(\mathcal{S}_{(j,i)}) = \sum_{k=1}^{|\mathcal{S}_{(j,i)}|} cancelled_{(k,j,i)}[\neg critical_{(k,j,i)}] \tag{8.4}$$

[1]The natural numbers are the non-negative integers: fields for these values were implemented in the metrics framework data cache described in Chapter 7 using unsigned long integers.

$$srt\_fraction(\mathcal{S}_{(j,i)}) = \frac{srt\_made(\mathcal{S}_{(j,i)})}{\begin{array}{l} srt\_made(\mathcal{S}_{(j,i)}) + \\ srt\_missed(\mathcal{S}_{(j,i)}) + \\ srt\_cancelled(\mathcal{S}_{(j,i)}) \end{array}} \tag{8.5}$$

We then define a weighted *efficiency* function f over a sample, that is zero if any HRT deadline is missed in the data sample and otherwise is the fraction of SRT deadlines made in that sample:

$$f(\mathcal{S}_{(j,i)}) = \begin{cases} 0 & : \textbf{if } \exists \mathcal{O}_{(k,j,i)} \in \mathcal{S}_{(j,i)} \left| \begin{array}{l} critical_{(k,j,i)} \wedge \\ ((missed_{(k,j,i)} \neq 0) \\ \vee (cancelled_{(k,j,i)} \neq 0)) \end{array} \right. \\ srt\_fraction(\mathcal{S}_{(j,i)}) & : \textbf{otherwise} \end{cases} \tag{8.6}$$

Figure 8.20 illustrates the behavior of the function f shown in Equation 8.6, over each sample $\mathcal{S}_{(j,6)}$ for RMS with and without cancellation in operating region 6, which is loaded very close to the feasible limit for all operations to make their deadlines. RMS without cancellation performed perfectly in region 6, making all SRT and HRT deadlines. RMS both with and without cancellation performed similarly well in regions 0-5. However, the overhead of adding cancellation to RMS in region 6 pushes the system into overload, and deadlines are missed. Furthermore, HRT deadlines are missed in a significant number of the samples, so that RMS with cancellation is not a suitable strategy under these load conditions.

Figure 8.21 shows the behavior of the efficiency function f shown in Equation 8.6, over each sample $\mathcal{S}_{(j,7)}$ for RMS with and without cancellation in operating region 7, which is loaded to just beyond the feasible limit for all operations. RMS with cancellation missed HRT deadlines in all but three data samples, while RMS without cancellation only missed HRT deadlines in five data samples, and met all SRT and HRT deadlines in the other samples. Even without the overhead of cancellation, however, the inability of RMS to protect HRT deadlines under conditions of slight overload means that neither RMS strategy is suitable under those conditions.

Figure 8.22 shows the behavior of the function f shown in Equation 8.6, over each sample $\mathcal{S}_{(j,8)}$ for RMS with and without cancellation in operating region 8, which is further loaded beyond the feasible limit. In region 8, RMS with cancellation misses

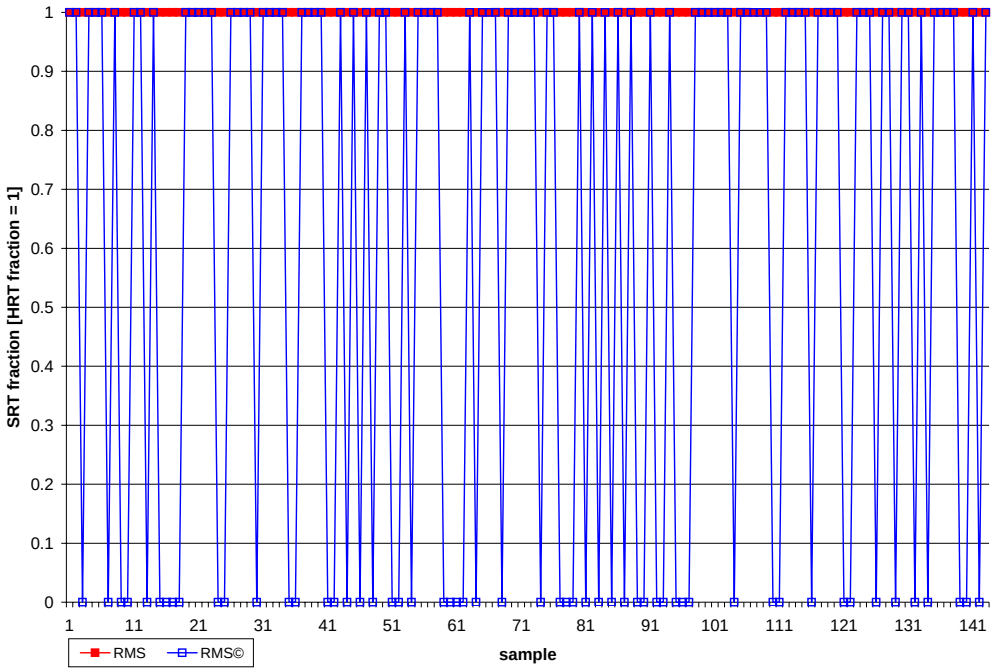

Figure 8.20: Region 6: Missed HRT Deadlines in RMS With and Without Cancellation

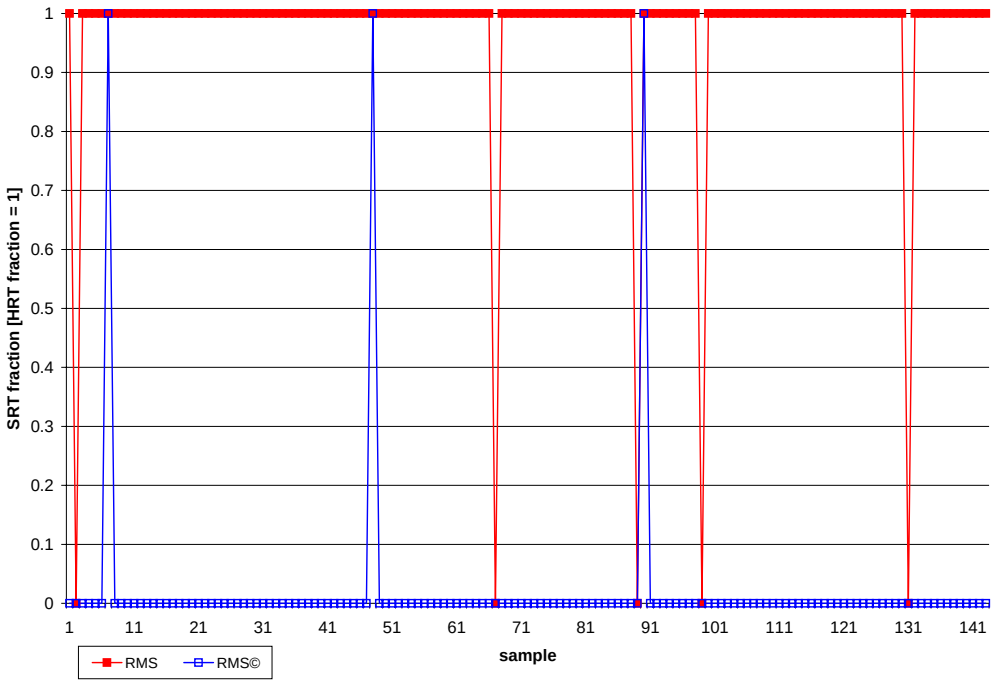

Figure 8.21: Region 7: Missed HRT Deadlines in RMS With and Without Cancellation

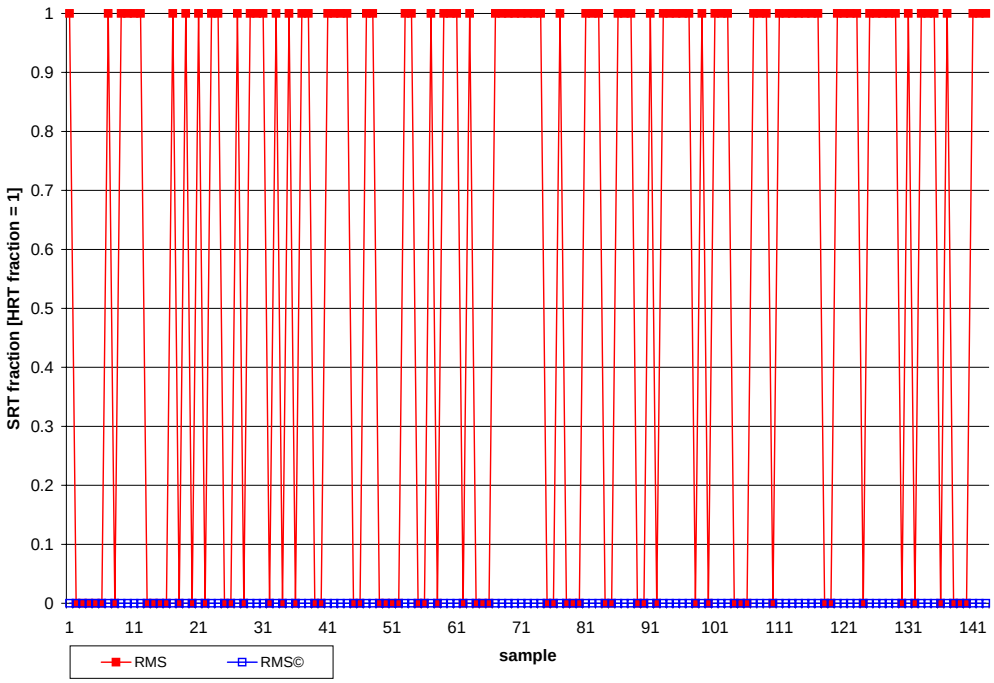

Figure 8.22: Region 8: Missed HRT Deadlines in RMS With and Without Cancellation

HRT deadlines in every sample. RMS without cancellation performs in region 8 similarly to the way RMS with cancellation performed in region 6, missing HRT deadlines in a significant number of the samples. Finally, in regions 9-11, RMS both with and without cancellation missed HRT deadlines in every data sample, and thus are not suitable strategies under those load conditions.

In addition to the missed HRT deadlines for RMS with and without cancellation, 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, and it occurred in the same region at the same sampling point for both strategies. We now examine the possible causes of this phenomenon.

Figure 8.23 shows the same function f shown in Equation 8.6 and plotted in Figures 8.20, 8.21, and 8.22, but over each sample $\mathcal{S}_{(j,9)}$ for MUF without cancellation and RMS+MLF with cancellation in region 9. As Section 8.2.1 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

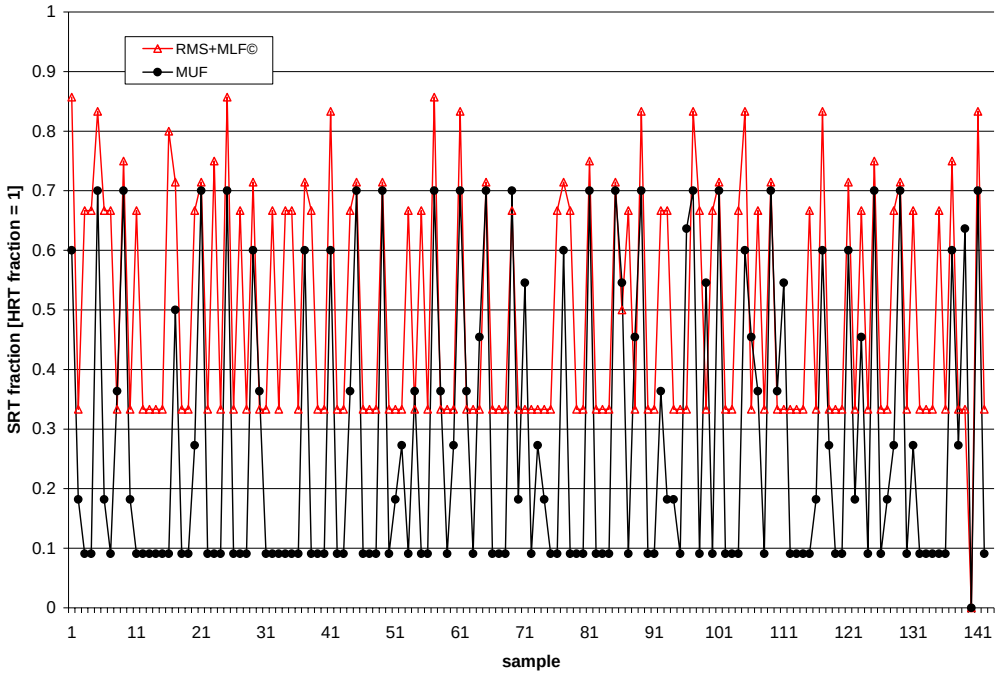

Figure 8.23: Region 9: missed HRT deadlines in MUF and RMS+MLF With Cancellation

operation used to induce randomized jitter to various operating regions, as described in Section 8.2.1.

The range of jitter in this operation for region 9, shown in Table 8.1, 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. Furthermore, 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.

Therefore, it 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 particular anomaly. In particular, if delays from preemption by spurious VxWorks network task interrupts contributed to this effect, it appears likely that they did so in aggregation of several small intervals of preemption, rather than in a single longer interval.

8.3.4 Dispatching Efficiency and Effectiveness

A key distinction in assessing the relative performance of the scheduling strategies in these studies is between the *number* and the *percentage* of deadlines made. In particular, strategies that successfully employed cancellation reduced the number of requests overall, but some percentage of those cancelled would have missed their respective deadlines had they been dispatched. We begin by comparing the efficiency and effectiveness of MUF and RMS+MLF both with and without cancellation in three representative operating regions.

As Figures 8.20, 8.21, and 8.22 show, RMS both with and without cancellation performed in an *all-or-nothing* manner in these experiments, either missing at least one HRT deadline or making all deadlines in each data sample. Therefore, the effectiveness and efficiency of these strategies are indistinguishable, and we thus confine our attention to MUF with and without cancellation and RMS+MLF with and without cancellation. We conclude by comparing the effectiveness of the strategies that performed best in at least on operating region (*i.e.*, MUF, RMS, and RMS+MLF all without cancellation) across operating regions.

To examine both the *efficiency* and the *effectiveness* of each strategy, we first define a weighted *effectiveness* function g over a sample, that is zero if any HRT deadline is missed in the data sample and otherwise is the number of SRT deadlines made in that sample:

$$g(\mathcal{S}_{(j,i)}) = \begin{cases} 0 & : \ \textbf{if } \exists \mathcal{O}_{(k,j,i)} \in \mathcal{S}_{(j,i)} \left| \begin{array}{l} critical_{(k,j,i)} \wedge \\ ((missed_{(k,j,i)} \neq 0) \\ \vee (cancelled_{(k,j,i)} \neq 0)) \end{array} \right. \\ srt\_made(\mathcal{S}_{(j,i)}) & : \ \textbf{otherwise} \end{cases} \tag{8.7}$$

We note that the effectiveness function g is very similar to the efficiency function f shown in Equation 8.6, except that it sums using the srt_made helper function instead of the $srt\_fraction$ helper function.

The plots shown in Figures 8.23, 8.20, 8.21, and 8.22 are useful for describing the fine-grained behavior of the heuristics, but are also very noisy, with the efficiency function f alternating frequently between two or more values. To assess overall efficiency and effectiveness of heuristics, we therefore seek a measure that

1. is more stable (is less noisy),

2. can capture the aggregate behavior of the heuristics, and

3. is still sufficiently fine-grained to reveal different behaviors in different parts of an operating region.

Without loss of generality, we can extend the definition of a sample to include the result of concatenation of samples under an operator $\oplus$, and extend the definitions of the efficiency function f shown in Equation 8.6 and the effectiveness function g shown in Equation 8.7, to apply to concatenations of samples as well as single samples.

We first adopt the following renaming convention when concatenating samples across operating region boundaries:

- sample $\mathcal{S}_{(|\mathcal{R}_i|,i)}$ is by definition the last sample in operating region $\mathcal{R}_i$
- sample $\mathcal{S}_{(1,i+1)}$ is the first in operating region $\mathcal{R}_{i+1}$
- we can without loss of generality rename any sample $\mathcal{S}_{(j,i)}$ in operating region $\mathcal{R}_i$ as though it were part of operating region $\mathcal{R}_{i+1}$, as long as we do not use the same values for index j as another sample already in $\mathcal{R}_{i+1}$
- in particular we can rename as follows:

$$\begin{array}{lcl} \mathcal{S}_{(|\mathcal{R}_i|,i)} & \Rightarrow & \mathcal{S}_{(0,i+1)} \\ \mathcal{S}_{(|\mathcal{R}_i|-c,i)} & \Rightarrow & \mathcal{S}_{(-c,i+1)} \\ \neg\exists_j \left| (\mathcal{S}_{(j,i)} \in \mathcal{R}_i \right. & \Rightarrow & \lambda \text{ (the } \textit{empty sequence}\text{)} \end{array}$$

- furthermore, renaming a sample can be extended trivially to renaming each of its constituent operation statistics:

$$\begin{array}{lll} \forall_k & (\mathcal{O}_{(k,|\mathcal{R}_i|,i)} & \Rightarrow \mathcal{O}_{(k,0,i+1))} \\ \forall_k & (\mathcal{O}_{(k,|\mathcal{R}_i|-c,i)} & \Rightarrow \mathcal{O}_{(k,-c,i+1))} \\ \neg\exists_{k,j} & \left| (\mathcal{O}_{(k,j,i)} \in \mathcal{S}_{j,i)} \in \mathcal{R}_i) \right. & \Rightarrow \lambda \end{array}$$

Finally, we define the result of a *concatenation window function* c, parameterized with an ending sample $\mathcal{S}_{(j,i)}$ and a window size w over a sequence of samples, to be the sequence of operation statistics produced by concatenation (denoted by operator $\oplus$) of the operation statistics in the samples in the resulting window, as follows:

$$
\begin{aligned}
c(w, \mathcal{S}_{(j,i)}) &= \bigoplus_{x=w-1}^{0} \mathcal{S}_{(j-x,i)} \\
&= \mathcal{O}_{(1,j-w+1,i)}, \ldots, \mathcal{O}_{(|\mathcal{S}_{(j-w+1,i)}|,j-w+1,i)}, \\
&\quad \ldots, \\
&\quad \mathcal{O}_{(1,j,i)}, \ldots, \mathcal{O}_{(|\mathcal{S}_{(j,i)}|,j,i)}
\end{aligned} \tag{8.8}
$$

For the comparisons in the remainder of this chapter, we selected a window size of 20 as most effective to simultaneously

1. minimize irrelevant noise in the efficiency and effectiveness functions, and
2. reveal meaningful variations in the performance of the heuristics both within and between operating regions.

We denote the partially bound concatenation window function of size 20 by c_{20}, and compose the efficiency function f shown in Equation 8.6 or effectiveness function g shown in Equation 8.7 with c_{20} to obtain $f \circ c_{20}$ or $g \circ c_{20}$, respectively.

The combinations of effectiveness and efficiency we observed can be shown effectively by examining three canonical operating regions, region 7 region 8, and region 10. As Figure 8.3 illustrates, region 7 had moderate jitter, region 8 had very low jitter, and region 10 had very high jitter. Figures 8.24 and 8.25 show respectively the efficiency and effectiveness functions f and g composed with the concatenation function of window size of 20 c_{20}, for the MUF and RMS+MLF scheduling strategies, both with and without cancellation, in operating region 7. Figure 8.24 shows the composed efficiency function $f \circ c_{20}$, and Figure 8.25 shows the composed effectiveness function $g \circ c_{20}$.

In region 7, RMS+MLF with cancellation was the most efficient among those not missing HRT deadlines, making a little over 50% of the SRT deadlines of all SRT requests made. MUF with cancellation made slightly less than 50% of the SRT deadlines, MUF without cancellation varied between 25% and 45% of SRT operation deadlines made, and RMS+MLF without cancellation was fairly steady at a little under 30% of SRT deadlines made.

Looking at effectiveness, the relative ordering of RMS+MLF with cancellation and MUF with cancellation seen for efficiency was preserved, as was the relative ordering of MUF without cancellation and RMS+MLF without cancellation. However, the strategies with cancellation though more efficient, were less effective in the total number of SRT deadlines made than were the strategies without cancellation. Although

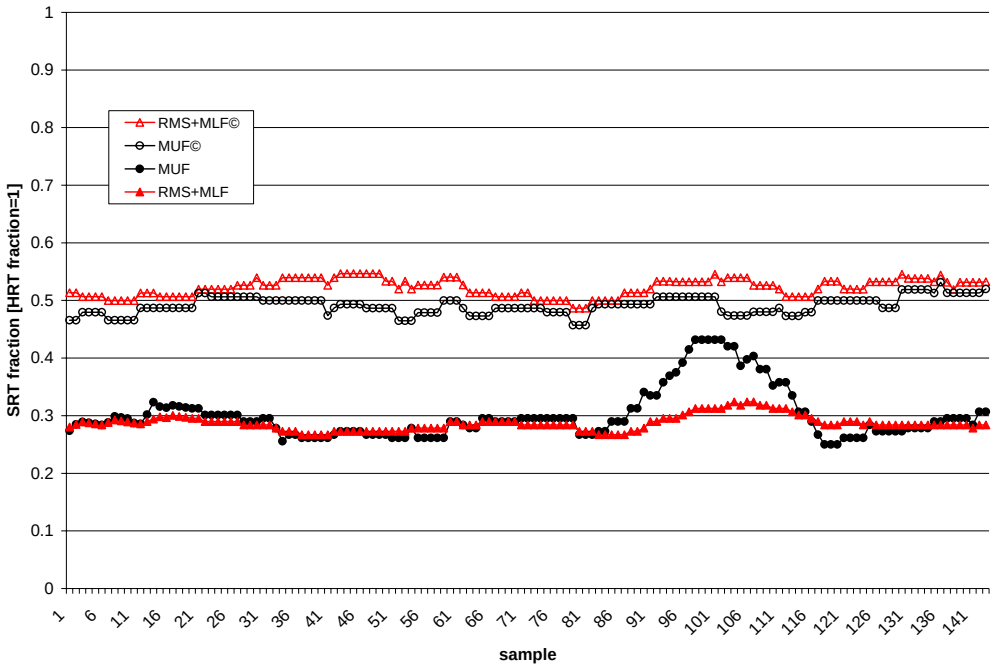

Figure 8.24: Region 7 SRT Deadlines Made: Efficiency

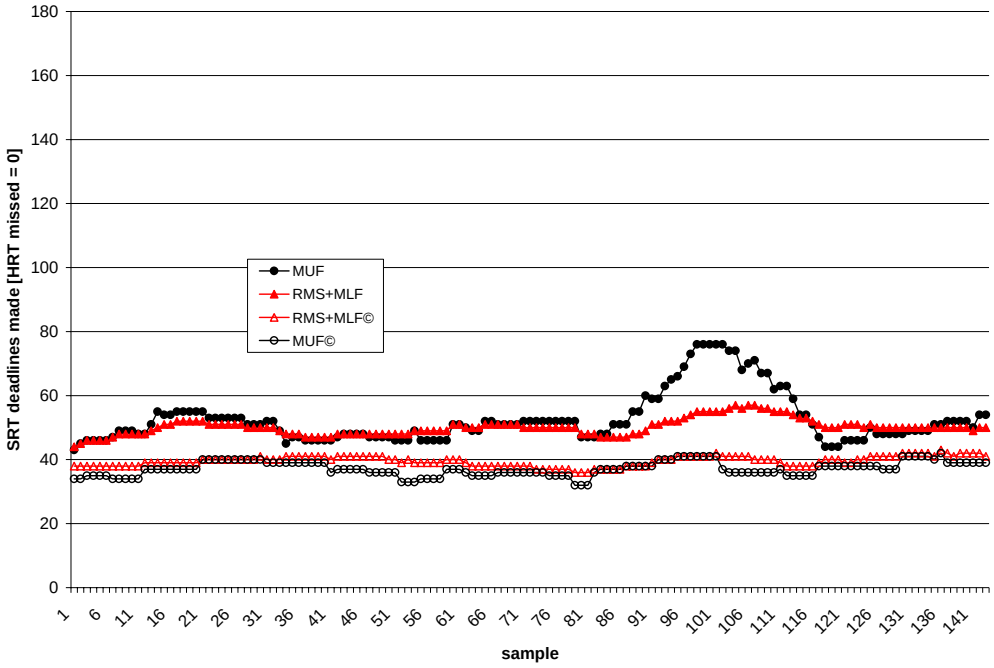

Figure 8.25: Region 7 SRT Deadlines Made: Effectiveness

the effectiveness improvement of MUF without cancellation over RMS+MLF without cancellation was moderate overall, we note that in the experimental application, any improvement in made deadlines can translate to a tangible improvement in the system, *e.g.*, by refreshing an additional computed route leg more frequently on the pilot's display.

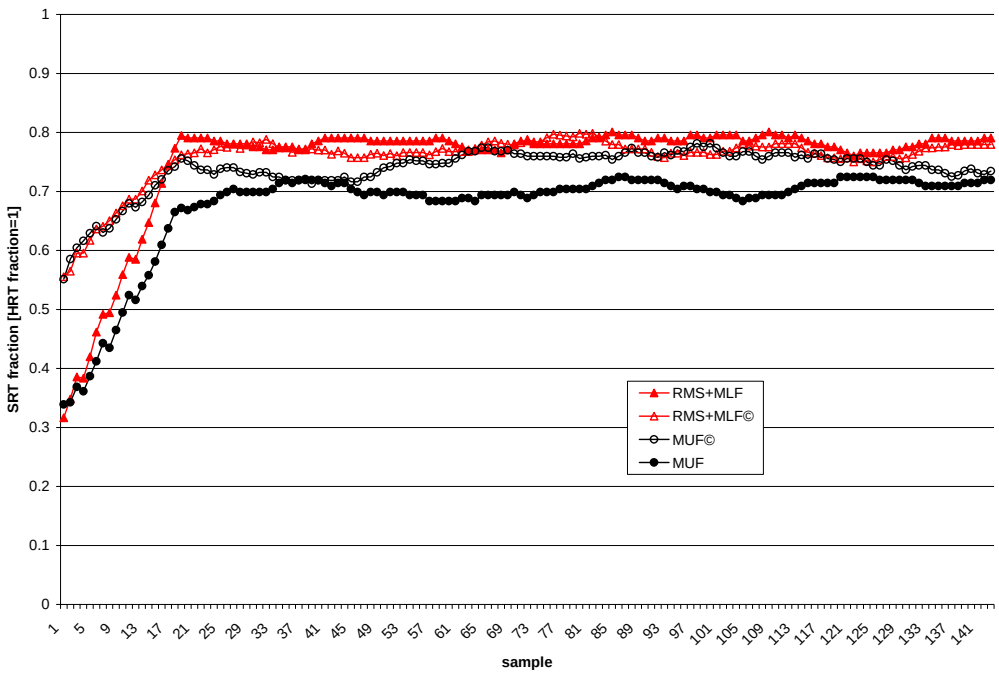

Figure 8.26: Region 8 SRT Deadlines Made: Efficiency

Figures 8.26 and 8.27 illustrate respectively the levels of efficiency and effectiveness of the scheduling strategies in an operating with very low jitter, region 8. Figure 8.26 shows the composed efficiency function $f \circ c_{20}$, and Figure 8.27 shows the effectiveness function $g \circ c_{20}$. In region 8, RMS+MLF without cancellation was slightly more efficient than RMS+MLF with cancellation, which was more efficient than MUF with cancellation, which was in turn more efficient than MUF without cancellation.

This suggests that in very low levels of jitter the ability of cancellation to reduce futile dispatches is at least balanced and possibly exceeded by the additional overhead of doing cancellation, and that the overhead associated with the additional dynamic queue management in MUF is similarly undesirable with low jitter. With the highest efficiency, low overhead, and no possibility of incorrect cancellation, RMS+MLF without cancellation was also most effective in region 8, achieving between 10 and 20 more SRT

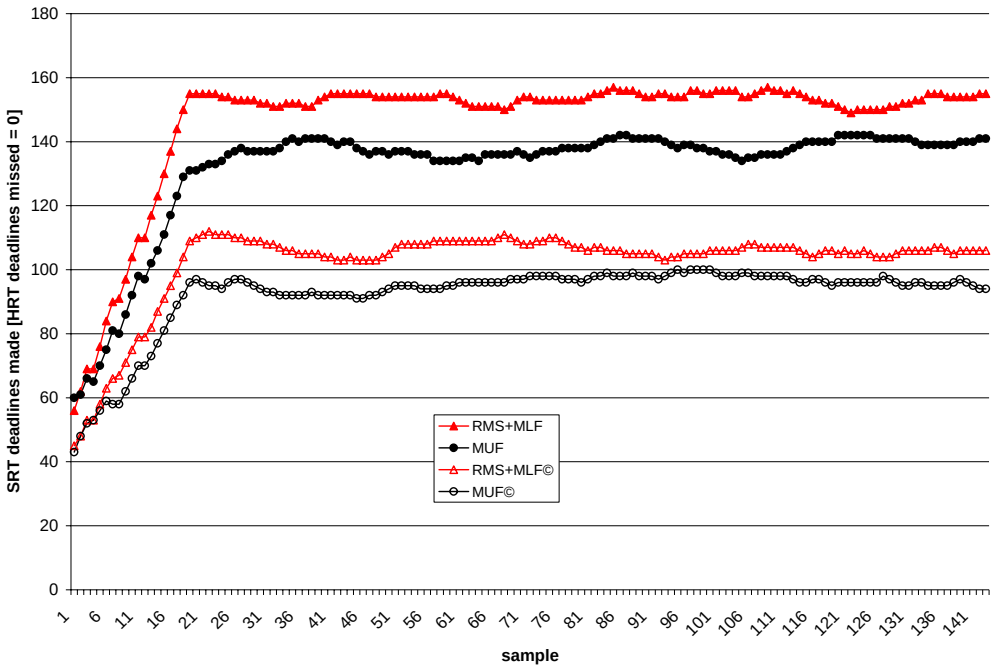

Figure 8.27: Region 8 SRT Deadlines Made: Effectiveness

deadlines over the window than any other strategy. MUF without cancellation was next, followed by RMS+MLF with cancellation, and finally RMS+MLF with cancellation.

Figures 8.28 and 8.29 illustrate respectively the levels of efficiency and effectiveness of the scheduling strategies in an operating with very high jitter, region 10. Figure 8.28 shows the composed efficiency function $f \circ c_{20}$, and Figure 8.29 shows the composed effectiveness function $g \circ c_{20}$.

At the left of each region 10 diagram there is a residual effect from the window size of 20 and the single missed deadline in region 9, for MUF without cancellation, and RMS+MLF with cancellation. Beyond that, RMS+MLF with cancellation was the most efficient, followed by MUF with cancellation, then RMS+MLF without cancellation, and finally MUF without cancellation. Similar to the case in region 7 with moderate jitter, the strategies with cancellation in region 10 are more efficient but are also less effective than the strategies without cancellation. RMS+MLF without cancellation is the most effective, followed by MUF without cancellation, then RMS+MLF with cancellation, and finally MUF with cancellation.

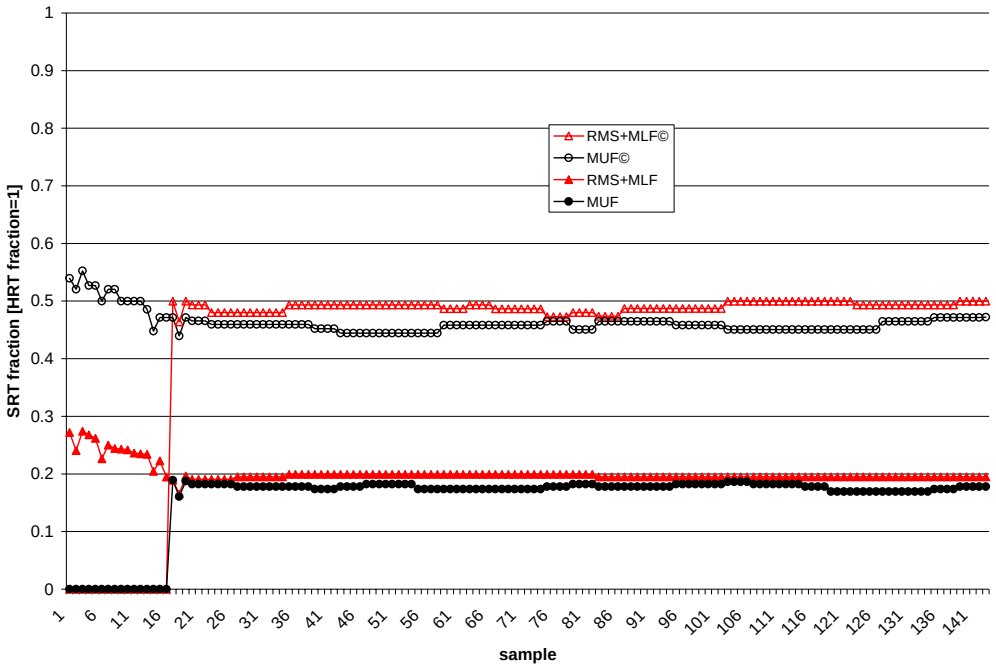

Figure 8.28: Region 10 SRT Deadlines Made: Efficiency

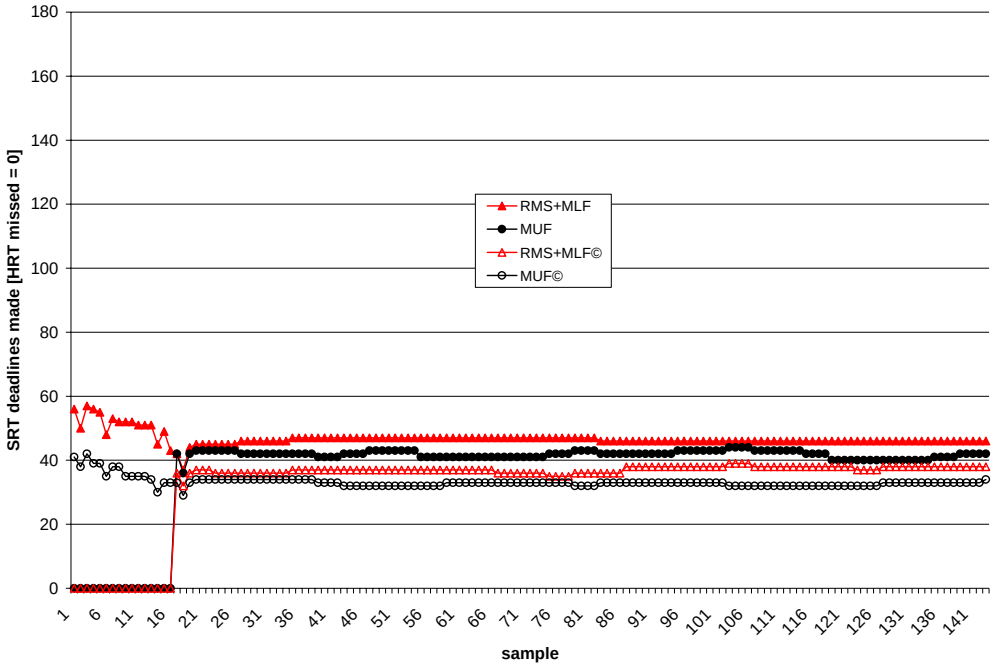

Figure 8.29: Region 10 SRT Deadlines Made: Effectiveness

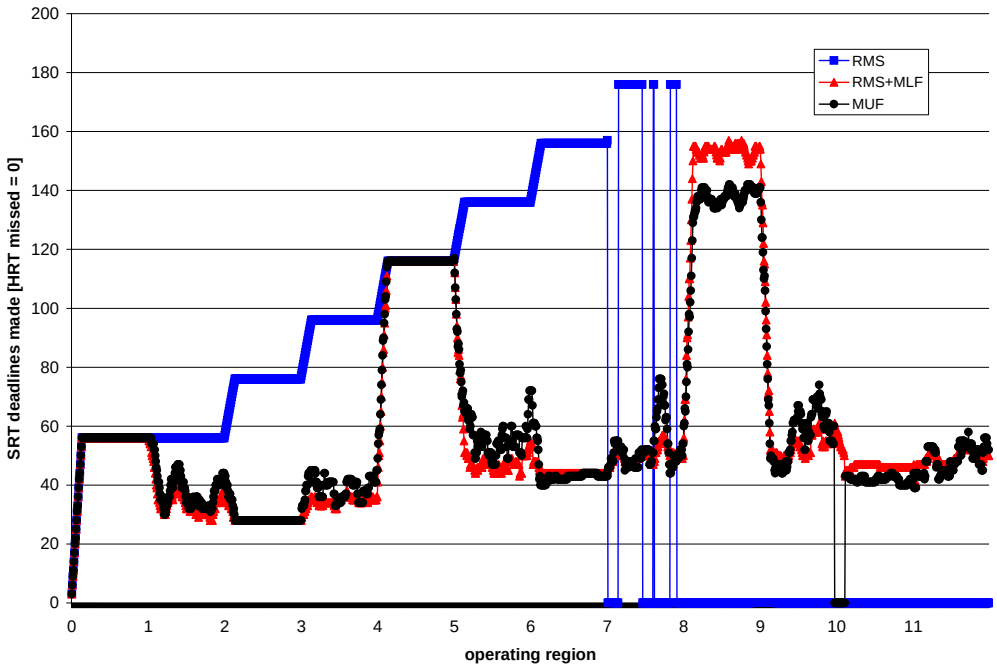

Figure 8.30: Effectiveness of the Dominant Strategies

Finally, we identify RMS, MUF, and RMS+MLF as *dominant* strategies, each having been the most effective strategy in at least one operating region. We then compare the effectiveness of each of the dominant strategies across operating regions. Figure 8.30 shows the composed effectiveness function $g \circ c_{20}$ in each operating region for each of the three strategies without cancellation, each of which was consistently more effective than its counterpart with cancellation. RMS without cancellation performed optimally among those surveyed under the load conditions seen in regions 0-6, but was not suitable due to missed HRT deadlines in regions 7-11. RMS with cancellation was as effective as RMS without cancellation in regions 0-5, but was unsuitable due to missed HRT deadlines in regions 6-11.

Interestingly, RMS+MLF without cancellation dispatched almost as many SRT deadlines in its most effective region, region 9, as RMS without cancellation did in its most effective region, region 6. Therefore, RMS+MLF without cancellation appears to a highly effective alternative to RMS under conditions of overload but where execution jitter is low. We also note that in even-numbered operating regions RMS+MLF without cancellation outperformed MUF without cancellation, and in odd-numbered operating

regions the reverse is true. We correlate this finding with the induced execution jitter described in Table 8.1, and note that in the even numbered operating regions the range of randomized additional execution jitter was either zero or 5-10 msec, while in the odd numbered operating regions it was either 0-5 msec or 0-10 msec. We consider correlation of several forms of runtime-observable information to this effect in Section 8.4, as such correlation may be of use in adaptively migrating among scheduling heuristics to optimize real-time behavior across varying load and load jitter conditions.

8.3.5 Summary of Observed Results

Section 8.3.1 examined the measured dispatching load and latency, and presents an analysis of deduced overhead. We observed the suppression effect of chain cancellation (described in Section 8.2.1) in MUF and RMS+MLF on each of the following: total load, total enqueue latency, and total dequeue latency. Cancellation did not have a noticeable effect on mean enqueue or dequeue latencies, except in MUF where it raised mean enqueue latency slightly.

We were able to isolate the overheads of static and dynamic queue management from jitter and preemption effects by studying the highest priority queue latencies of the MUF and RMS+MLF strategies. Using this approach, we are thus able to calibrate precisely the overhead of static and dynamic scheduling in the experimental application itself, rather than relying on the more general results discussed in Section dispimpl:selected:primitives. If a zero-jitter state can be found naturally in (or engineered into) an application, a similar precise evaluation can be made of the dispatching primitives employed for that application.

Section 8.3.2 presented measured operation deadline success, failure and cancellation data, and compared the behavior of the different strategies. Notably, while RMS+MLF with cancellation and MUF with cancellation shed SRT load by selectively cancelling SRT operations, RMS with cancellation failed to cancel even a single SRT operation. Using the simulation results presented in Section 4.6, we interpret this effect as a specific ineffectiveness of cancellation in RMS to shed high-rate SRT operations, in conditions of overload.

Section 8.3.3 made a closer examination of the canonical cases where HRT deadlines were missed. We defined an efficiency function f shown in Equation 8.6 that we used to examine the percentage of SRT operation deadlines made while still meeting all

HRT deadlines. In particular, operating regions 6 and 7 delimit the transition between a feasible and an infeasible schedule for RMS. In operating region 6, RMS without cancellation was still feasible, though the overhead of checking for cancellation pushed RMS with cancellation into overload, missing at least one HRT deadline in a significant number of samples. In operating region 7, RMS with cancellation missed HRT deadlines in all but a few of the samples and RMS without cancellation was just over the feasible utilization limit, missing at least one HRT deadline in each of five samples. Finally, we note the all-or-nothing efficiency of RMS both with and without cancellation. As each was assigned either a maximal (1) or minimal (0) score by efficiency function f, we forego examination of the *effectiveness* of RMS, except in comparison to the effectiveness of MUF and RMS+MLF.

Section 8.3.4 compared dispatching efficiency and effectiveness, particularly for strategies where cancellation had an effect, and compared effectiveness of all strategies across all operating regions. We defined an effectiveness function g shown in Equation 8.7. We then composed g with a concatenation window function c shown in Equation 8.8. We selected a concatenation window size of 20 as most effective in removing noise while retaining necessary precision. In states with moderate to high jitter, cancellation improved efficiency in MUF and RMS+MLF but decreased effectiveness. In states with no jitter, cancellation did not improve efficiency significantly.

Finally, we identified the most effective strategy for each operating region and compared the set of dominant strategies across operating regions. Figure 8.31 shows the

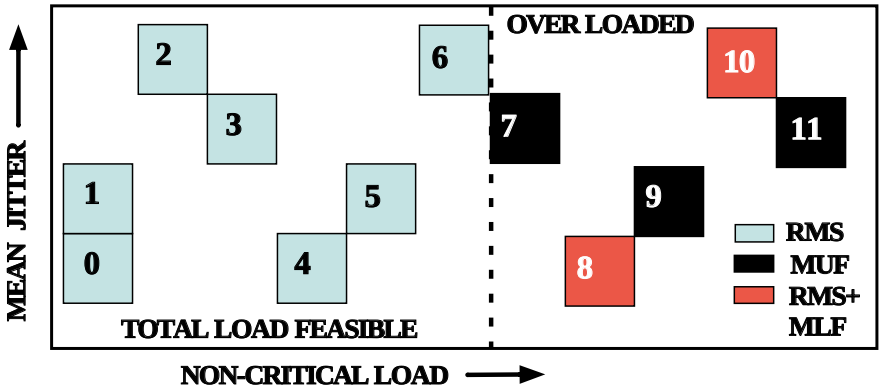

Figure 8.31: Most Effective Heuristic in each ASFD Operating Region

most effective heuristic in each of the operating regions. In Figure 8.30 we recolor each

of the operating regions originally portrayed in Figure 8.3 to indicate the scheduling heuristic that performed best under the composed function $f \circ c_{20}$ shown in Figure 8.30.

8.4 Correlation of Performance to Observable Characteristics

Because which scheduling heuristic is the most effective among those studied differs according to load and load jitter conditions, an important question is whether it would be possible to predict with reasonable accuracy that a heuristic will perform better than another using observable characteristics of the system. That is, although under the controlled conditions of these experiments we *know* the added load and load jitter, in general we would need to rely on the ability to *measure* those factors with reasonable fidelity in a running system, to predict which heuristic is most effective during an epoch of system operation. As a step in that direction, we consider the question of whether sources of information can be found that

1. can be directly observed or derived at run-time, and
2. correlate well with the actual performance of the heuristics.

This section is structured as follows: Section 8.4.1 examines the relationship between performance and information based on missed and made deadlines; Section 8.4.2 examines the relationship between performance and information based on measured operation latency.

8.4.1 Information Based on Deadlines

As a first approach simply monitoring an effectiveness function, such as $g \circ c_{20}$ shown in Figure 8.30, represents a possible source of run-time observable information that could correlate directly with performance of each heuristic at run-time, across a broader set of operating regions. For example, with sufficient test coverage of all possible operating regions in advance, this approach could be useful to at least partially constrain the run-time adaptation problem, by providing useful static information. First, this approach could serve to narrow the set of candidate heuristics for adaptation, by removing strategies that never outperformed the others. This could be done, *e.g.*, in parallel trials with

identical pseudo-random sequences, as was done for the experiments described in this chapter. Second, this approach might be used to determine key static parameters for adaptive control, *e.g.*, the feasible limit on the number of deadlines met without missing HRT deadline, using the *actual* system.

Unfortunately, this technique is unsatisfying for direct use in run-time adaptive control, as it unfortunately would need to be performed in several processors at once. If the narrowed set of candidate heuristics were small relative to the number of processors with *homogeneous* load and load jitter characteristics, adaptation using, *e.g.*, effectiveness function $g \circ c_{20}$ could be done at a cost of one CPU per heuristic. In this case a voting scheme could be used to determine which heuristic the controlled processors should be running at any given time.

However, this approach is inefficient and only applicable to a limited class of systems. Furthermore, it may be cumbersome to engineer, *e.g.*, identical loads would need to be offered to each of the CPUs with appropriate real-time characteristics. Therefore,

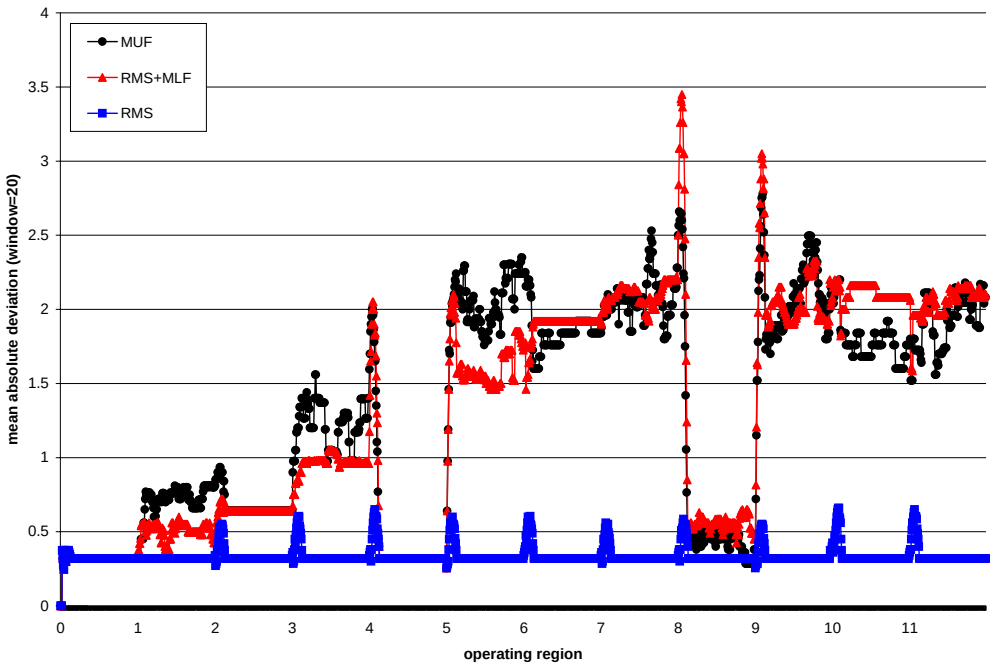

Figure 8.32: MAD of Effectiveness Function over Window Size 20

we would instead prefer an approach that could apply across heuristics and varying

levels of load and load jitter. We thus seek other run-time observable properties that correlate with the operating region, but do not depend on which specific heuristic is currently in use. Figure 8.32 illustrates one such alternative, which derives information from the composed effectiveness function $g \circ c_{20}$ plotted over all operating regions in Figure 8.30. Here, we consider the Mean Absolute Deviation (MAD) over a window of 20 samples of $g \circ c_{20}$. In general this MAD function tracks the load and load jitter conditions fairly well for MUF without cancellation and RMS+MLF without cancellation, with a more level MAD in even numbered operating regions, and a more variable MAD in odd numbered operating regions.

However, the MAD function is sensitive to the load level as well as the load jitter, so that rather than being able to use the value of the MAD function directly, derived values such as the variability of the MAD function must be computed in this approach. Furthermore, computing the variance in a measure such as the MAD function as in Section 8.4.1 is potentially computationally expensive, and we would naturally prefer an approach such as the latency measure described in Section 8.4.2 that offers lower overhead.

8.4.2 Information Based on Latency

As an alternative to deriving information from an effectiveness function as considered in Section 8.4.1, we might instead consider other run-time observable factors, such as operation dispatch latency. As latency is a fundamental factor in determining whether or not an operation misses its deadline, observing the aggregate latencies of a number of operations can be expected to offer reasonable correlation with the aggregate number of operations that miss or make their deadlines.

Figures 8.33 and 8.34 show the measured latencies of operation dispatches in each strategy in each operating region. Figure 8.33 shows the latencies of HRT operations, and Figure 8.34 shows the latencies of SRT operations.

Figures 8.35 and 8.36 show the measured latencies of operation dispatches in each strategy in each operating region for MUF. Figure 8.35 shows the latencies of HRT operations, and Figure 8.36 shows the latencies of SRT operations.

Figures 8.37 and 8.38 show the measured latencies of operation dispatches in each strategy in each operating region for RMS+MLF. Figure 8.37 shows the latencies of HRT operations, and Figure 8.38 shows the latencies of SRT operations.

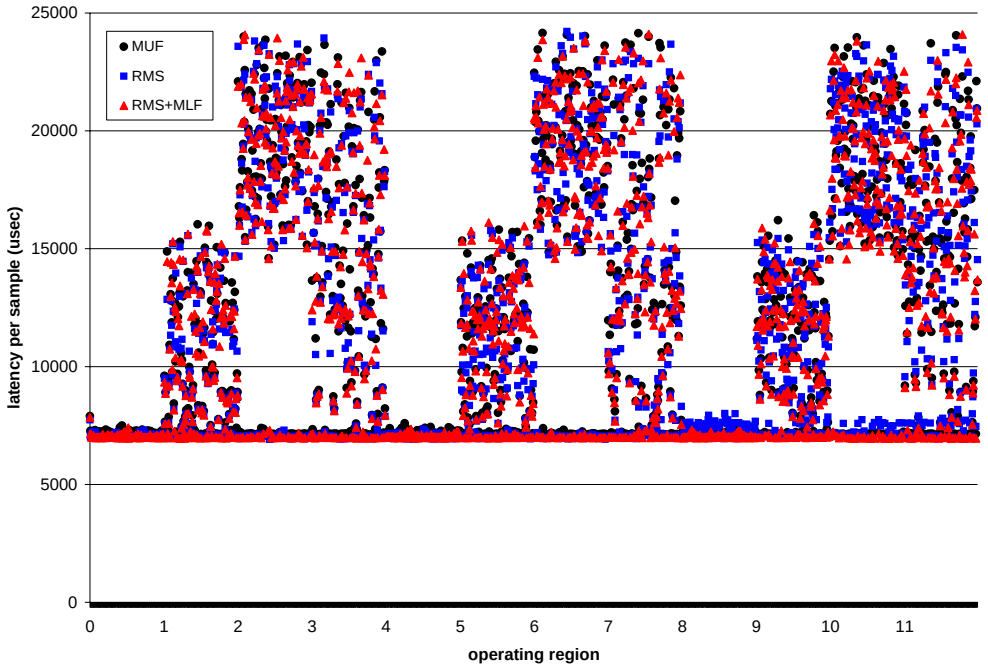

Figure 8.33: Measured Operation Latencies: HRT

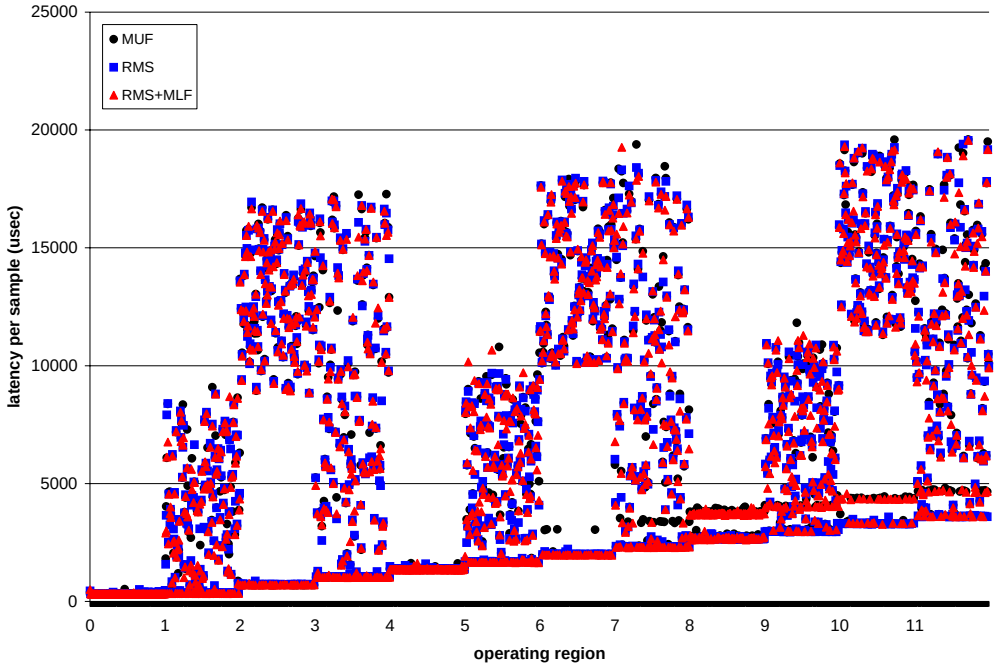

Figure 8.34: Measured Operation Latencies: SRT

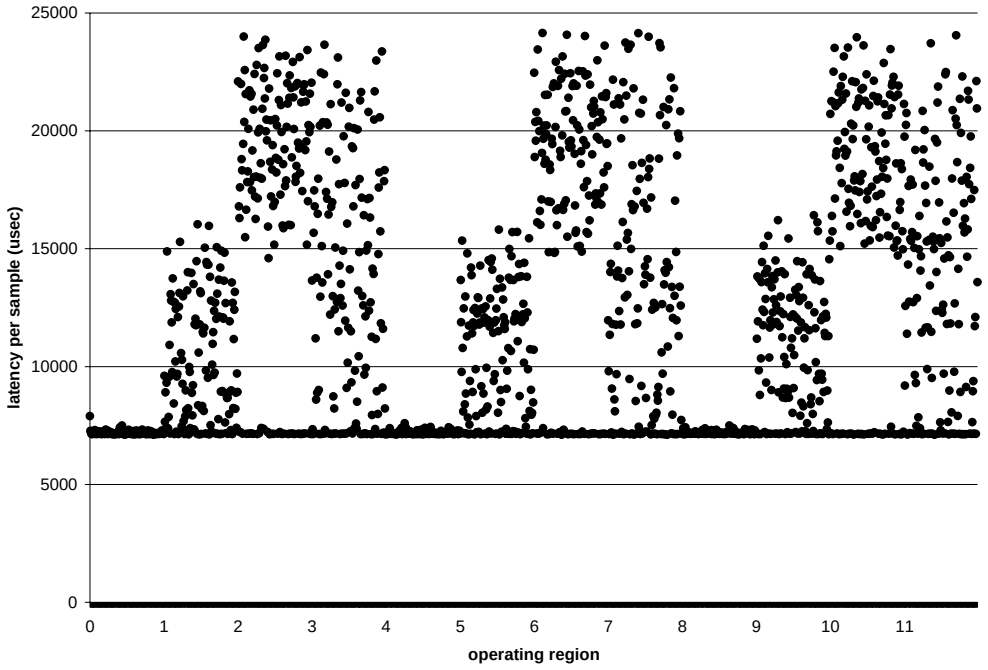

Figure 8.35: Operation Latencies in MUF: HRT

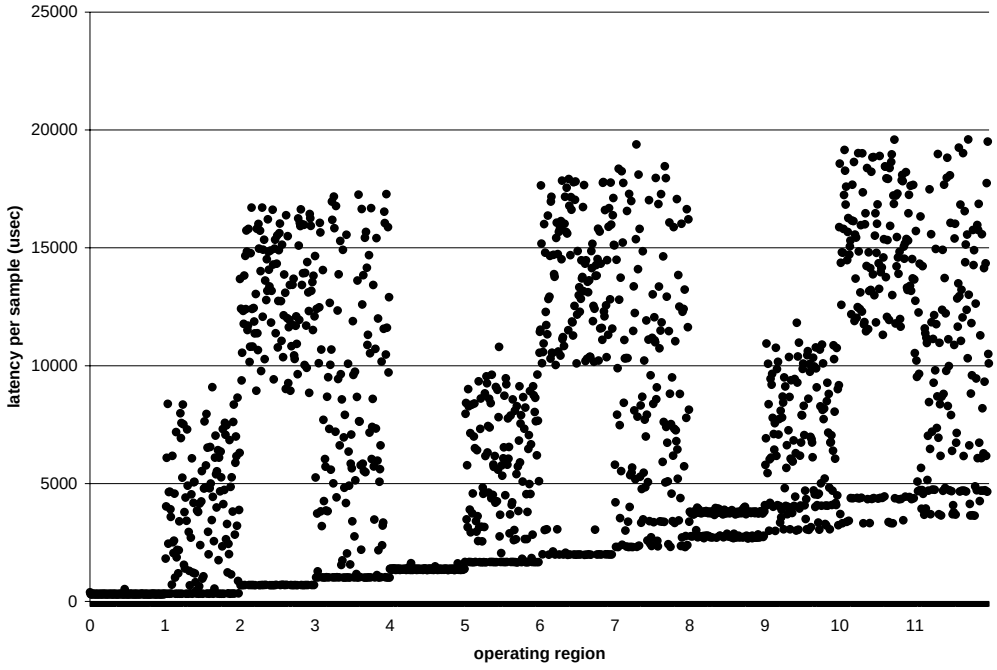

Figure 8.36: Operation Latencies in MUF: SRT

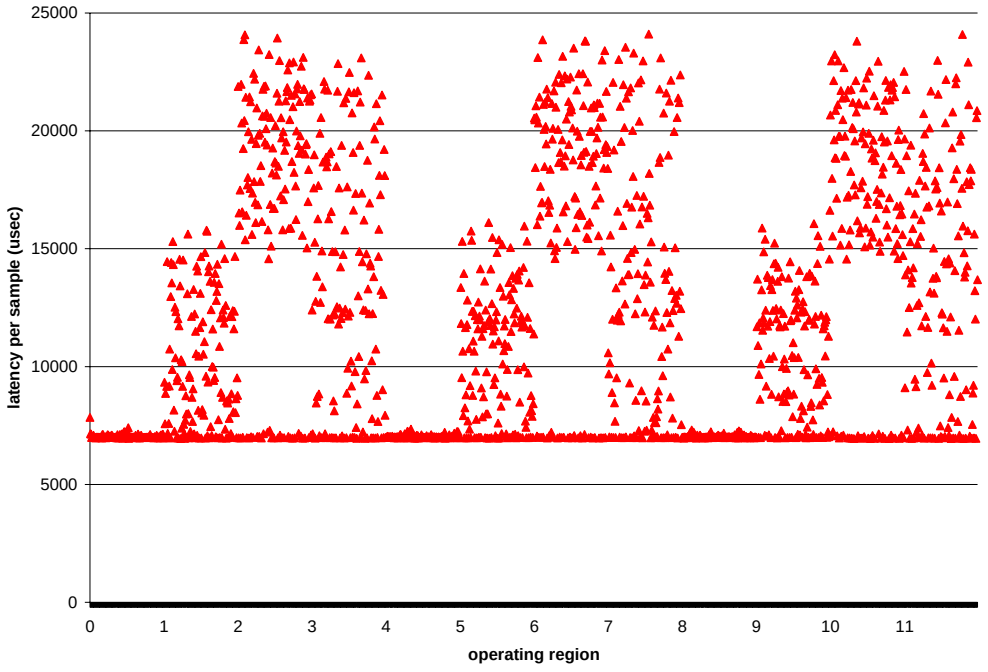

Figure 8.37: Operation Latencies in RMS+MLF: HRT

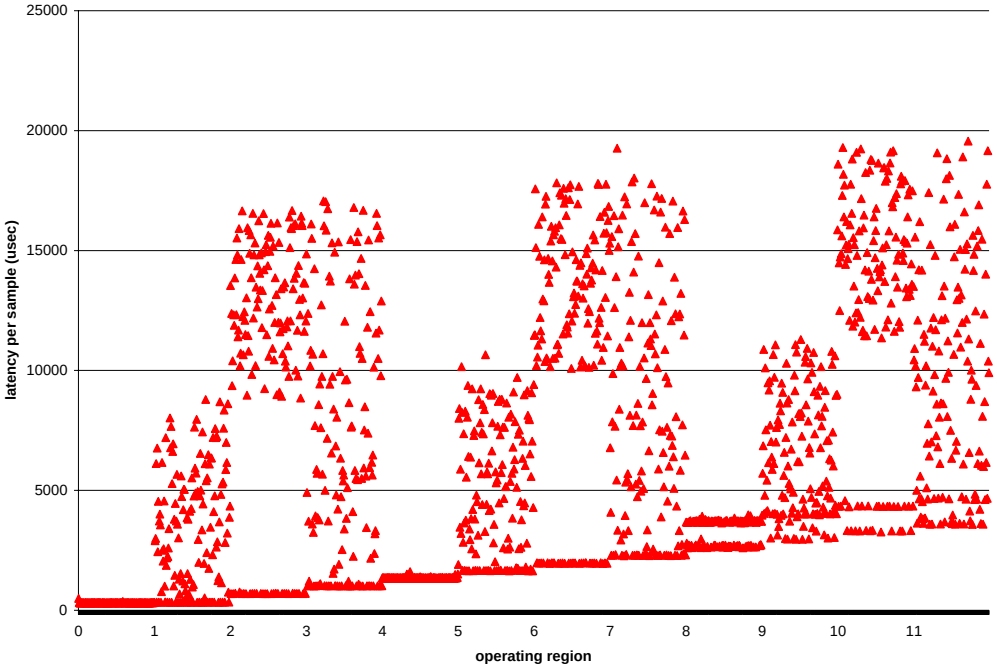

Figure 8.38: Operation Latencies in RMS+MLF: SRT

Although the latency of each operation includes preemption by other operations, the regular periodic nature of the application, combined with the harmonic framing of periods, results in a constant preemption delay over a small number of samples[2]. Therefore, we can for this OFP application consider operation latency a reliable indicator of load and load jitter. Because the experiments assumed an application where only SRT load was added (though load jitter was added to both HRT and SRT operations), and priority was used effectively to isolate HRT load from SRT load, the HRT operation latency was insensitive to the increased SRT load in each successive operating region, as shown in Figure 8.33. SRT operation latency on the other hand did show sensitivity to both load and load jitter, as shown in Figure 8.34.

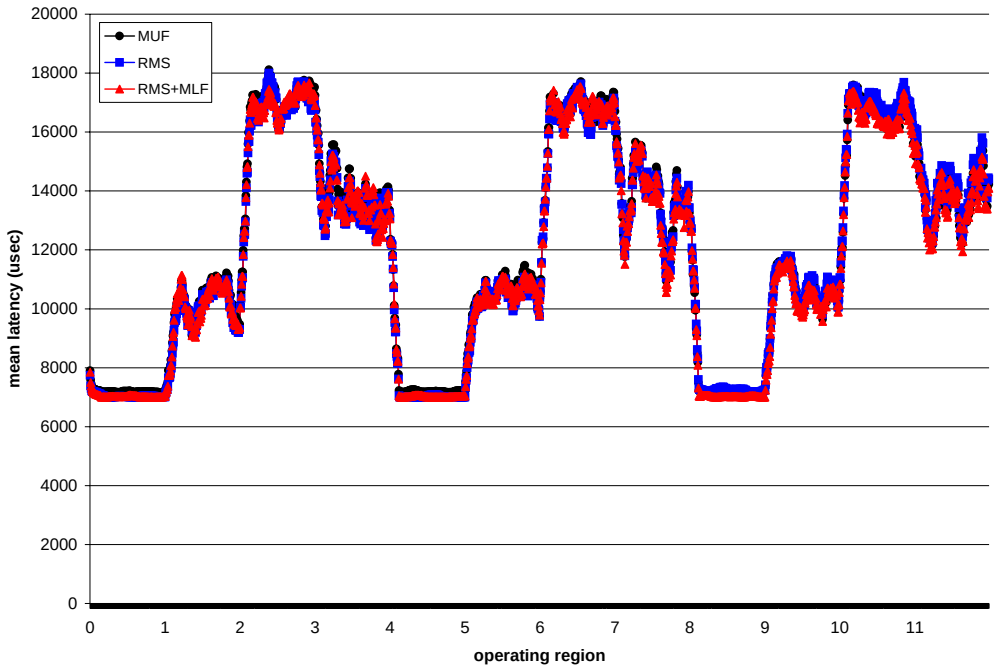

Figure 8.39: Mean Operation Latency Over 20 Samples: HRT

While operation latencies are highly correlated with the load and load jitter conditions of each operating region, it is only in aggregate that they are correlated with the operating regions, and thus with the performance of the heuristics themselves. Therefore, the final step towards a suitable run-time observable measure for potential use in

[2]In general, a similar argument applies to bounded blocking delays, though OFP operations are designed not to perform any blocking calls

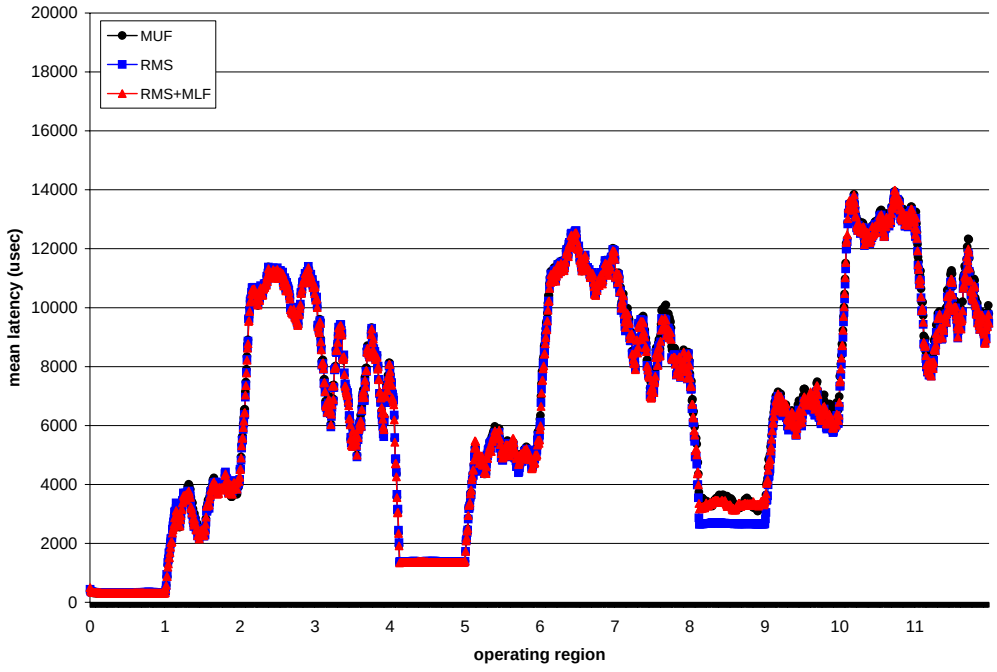

Figure 8.40: Mean Operation Latency Over 20 Samples: SRT

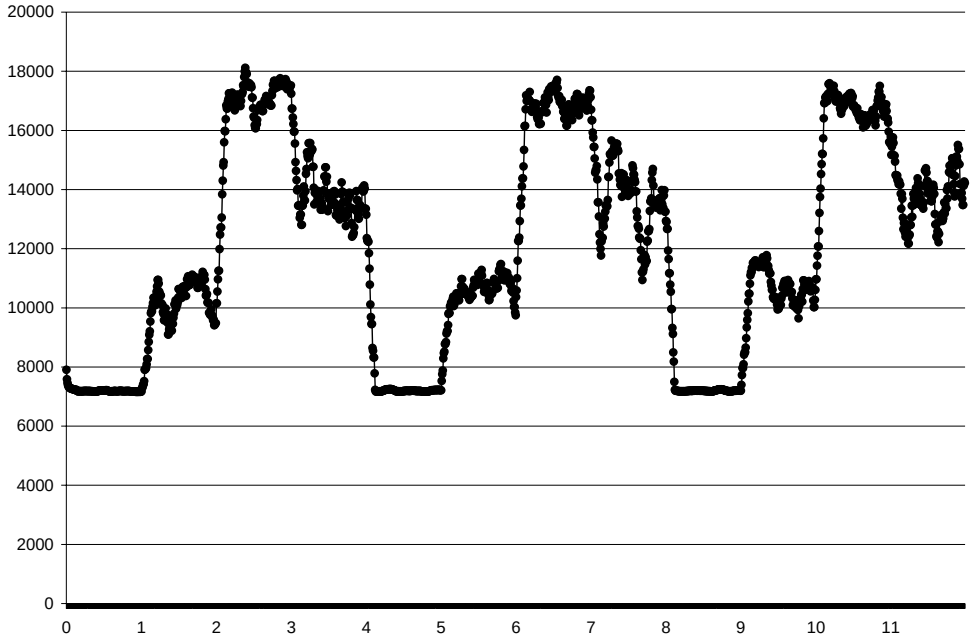

Figure 8.41: Mean Operation Latency Over 20 Samples in MUF: HRT

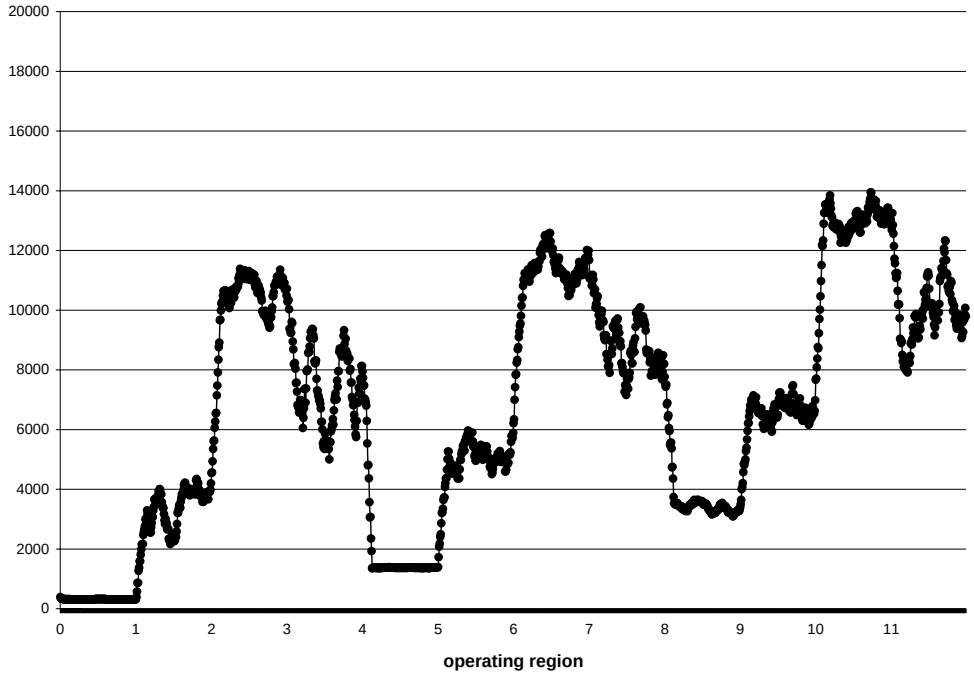

Figure 8.42: Mean Operation Latency Over 20 Samples in MUF: SRT

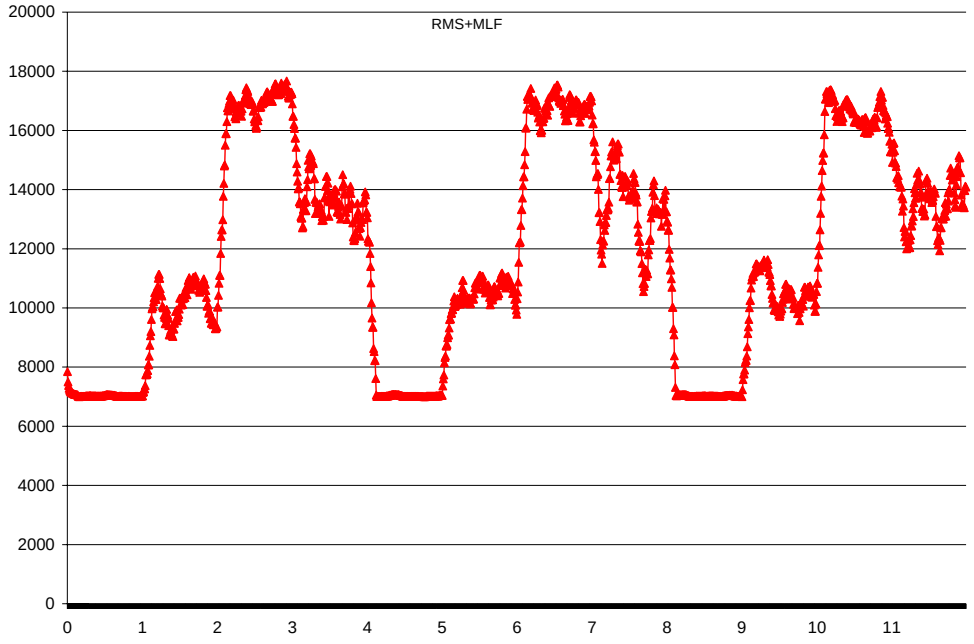

Figure 8.43: Mean Operation Latency Over 20 Samples in RMS+MLF: HRT

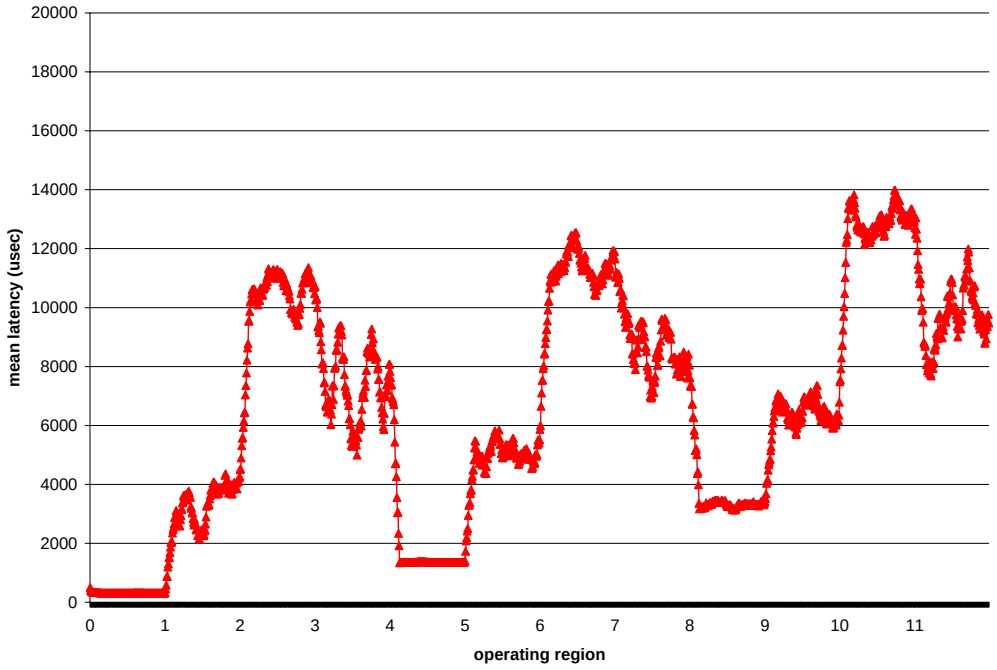

Figure 8.44: Mean Operation Latency Over 20 Samples in RMS+MLF: SRT

adaptive migration among scheduling strategies is to combine multiple individual latency measurements into a single value. Figures 8.39 and 8.40 show the mean operation dispatch latencies over a window of 20 samples. Figure 8.39 shows the plot for HRT operations, and Figure 8.40 shows the plot for SRT operations.

We note several features of these plots. First, the HRT mean latency is highly correlated with whether MUF without cancellation or RMS+MLF without cancellation performed better. In even numbered operating regions, the mean latency is below 8000 usec (8 msec) or above 1600 usec (16 msec), and in the odd numbered operating regions it lies between those values. This property holds across all scheduling strategies and operating region considered. Second, because of sensitivity to load, and the effects of SRT operation cancellation, this property does not hold for the mean SRT plot. Therefore, the mean HRT plot is selected as the preferred run-time observable measure.

8.5 Conclusions

This chapter has focused on the empirical study of several scheduling heuristics for mission computer OFPs. As mission computing software is being asked to execute

in more flexible ways, 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 of fundamental importance.

Furthermore, new increasingly non-deterministic kinds of processing are being targeted for transition to these systems [72]. The ability of the Kokyu framework to manage variations in execution load and load jitter through alternative scheduling heuristics increases the applicability of these techniques to OFP mission computing systems with next-generation software components. Increased openness of systems is thus enabled by the Kokyu framework.

Finally, by demonstrating the existence of run-time observable characteristics that correlate with performance of the scheduling heuristics, this work opens a larger possibility: performing truly adaptive scheduling using alternative heuristics at run-time, to accommodate variations in the systems operating environment and current mission objectives. There are several open questions to address, as Chapter 9 describes, before this kind of run-time adaptation will be applicable to avionics mission computing OFPs. However, these problems appear tractable, and planned future work will lead to a more complete solution.

We conclude this chapter with several key observations and recommendations. These observations and recommendations apply both to the particular experimental avionics mission computing application studied, and to a larger family of rate-based distributed real-time applications.

Overhead: 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 shown in Figures 8.10 and 8.11 was noticeable, but was within a reasonable scalar (approximately 1.5) of the more static queue management overhead. Furthermore, 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. Therefore, developers of rate-based real-time distributed applications should consider dynamic scheduling in middleware to be a reasonable and useful technique.

Overload: Hybrid static/dynamic scheduling strategies such as MUF and RMS+MLF appear to be effective in managing dynamic SRT load, and isolating the HRT that load. Furthermore, they did so under different levels and ranges of randomized jitter

in the execution times of certain HRT and SRT operations at different rates. Therefore, criticality-aware hybrid static/dynamic scheduling in middleware should be considered for systems that

1. have both critical and non-critical operations, and
2. may incur total load in excess of the feasible bound.

Cancellation: Operation cancellation was shown, under conditions of moderate to high execution jitter, to improve efficiency of the MUF and RMS+MLF strategies. Although the pessimistic form of cancellation we applied reduced the effectiveness of these strategies under those same conditions, several possible refinements to the cancellation algorithm show promise to *improve* effectiveness as we describe in Section 9.2. Of particular interest in the cancellation studies was:

1. the failure of RMS with cancellation to shed any SRT load, and
2. the efficiency improvement only in the presence of jitter for MUF and RMS+MLF.

We attribute the former effect to a particular limitation of RMS to shed high-rate SRT operations. We interpret the second effect to be a product of the stable and harmonic pattern of the schedule in operating regions with no additional jitter.

With a stable and harmonic schedule, and given preemption of lower priority operations by higher priority ones, the operations that cannot make their deadlines will be at the end of a particular phase of the schedule, and therefore will not impede operations that could have made their deadlines had those been cancelled. We note that jitter in execution times, much like noise on an electrical signal, both

1. changes the instantaneous (and possibly aggregate) utilization value, and
2. adds (likely non-harmonic) rates to the system [57].

We also note that with non-harmonic rates, the phasing of operations will vary which operations are vulnerable, leading to a similar need for active cancellation. Therefore, we recommend applying cancellation only in cases where:

1. the system is overloaded, so cancellation is beneficial,
2. jitter is present, or the set of rates is non-harmonic, and
3. the scheduling strategy in use can provide opportunities to cancel non-critical load by showing preference to critical operations.

Dominant Strategy: RMS without cancellation performed optimally when the execution load was below its feasible threshold. Above that limit RMS+MLF without cancellation performed best under conditions of high or low execution load jitter, and MUF without cancellation performed best under conditions of intermediate load jitter. Under the conditions studied, HRT operation latency was identified as an effective measure of execution jitter, and thus of the optimal strategy beyond the feasible RMS limit. Therefore we recommend using the following strategies in the following cases:

- RMS when the system is not overloaded,
- RMS+MLF or MUF when the system is overloaded, with the choice of which strategy to use depending on run-time observable characteristics that correlate with which strategy will perform best under those conditions.

While our preliminary results indicate a correlation between jitter levels (which we can best calculate using operation latencies) and the performance of RMS+MLF vs. MUF, we have only shown this for a single application. To generalize these results we believe it crucial to address the open questions in Section 9.4.

Chapter 9

Conclusions and Future Research Problems

This dissertation has focused on the problem of how to provide flexible and adaptive quality of service (QoS) management in middleware for realistic, rate-based real-time distributed systems. Taken together, the flexible scheduling and dispatching infrastructure described in Chapters 4 and 5 respectively, the flexible and efficient rate selection technique described in Chapter 6, and the performance monitoring and feedback infrastructure described in Chapter 7 constitute a robust and effective middleware environment for real-time QoS management.

The experiments and empirical results presented in Chapter 8 demonstrate the benefits and efficacy of applying the Kokyu middleware framework to both

1. avionics mission computing systems, and
2. a broader class of distributed rate-based real-time systems.

From a thorough examination of a particular avionics mission computing application, we distill the results of Section 8.3 to a concrete set of guidelines in Section 8.5, for general use by developers of rate-based distributed real-time systems. These guidelines and the results from which they are distilled form a core contribution we plan to expand to become a larger body of theory and practice on adaptive and reflective real-time QoS management and control for mission-critical distributed real-time and embedded systems.

The Kokyu framework has been to a significant and increasing degree closely integrated into the Bold Stroke research infrastructure, with potential for transition to

production avionics systems. Because the Kokyu framework supports, extends and most importantly *generalizes* the policies, mechanisms, interfaces, and programming abstractions of the previous-generation scheduling and dispatching software [117], integration with the Bold Stroke infrastructure has been straightforward.

Ultimately, the impact of this research will depend on the extent to which application requirements and corresponding resource constraints motivate its use. As distributed and real-time embedded applications

1. become more prevalent,
2. are asked to perform a greater variety of computation and communication tasks,
3. must still provide strict timeliness assurances,
4. rely on increasingly heterogeneous, distributed, and constrained resources, and
5. are subject to a variety of QoS requirements,

it is likely there will be *more* demand for flexible, reflective, and adaptive QoS management techniques, infrastructure, and analysis, rather than less.

However, to realize the full potential of this research, the open questions it raises must be addressed. First, Section 9.1 considers the wider applicability of the results obtained here, and suggests additional experiments and analysis to identify other contributors to the performance of scheduling strategies. Section 9.2 considers the question of whether the cancellation technique studied can be improved to be both effective and beneficial, and presents an approach to improving its accuracy and effectiveness using application-specific information and finer-grained profiles of operations' actual characteristics. Section 9.3 describes a set of related problems involving the ordering and re-ordering of operation dispatch requests in the face of

- adaptive transitions between strategies,
- multiple competing QoS policies, such as for real-time and fault-tolerance, and
- cooperative embedding of constraints, such as for intrusion detection and tolerance.

Finally, Section 9.4 considers the problem of adaptive rescheduling without modifying operation characteristics, outlines an approach to the sub-problem of performing adaptive control over scheduling strategies for improved QoS, and describes first steps in that direction.

9.1 Additional Studies

Additional studies are indicated to examine further the generality of the results described in Chapter 8, across a broader range of rate-based distributed real-time systems. In addition to performing similar experiments to those described in Chapter 8 both across a broader range of avionics mission computing applications, and applications outside that domain, we plan new kinds of experiments to address the following open issues:

Non-harmonic rates: whether applications with inherently non-harmonic rates exhibit similar correlation to performance of heuristics as seen with execution jitter. In particular, careful empirical study and comparison of the effects of jitter versus the effects of non-harmonic rates would serve to both verify and quantify the model of jitter as high-frequency additional load as suggested in Section 8.5.

Upper limits on jitter and load partitioning: *i.e.*, the range over which various strategies remain robust. The empirical studies in Chapter 8 demonstrate that the Rate Monotonic Scheduling (RMS)+Minimum Laxity First (MLF) and Maximum Urgency First (MUF) strategies can tolerate total load in excess of the feasible bound, while still meeting critical deadlines and achieving reasonable effectiveness in meeting non-critical deadlines. Furthermore, two distinct boundaries in the relative performance of RMS+MLF and MUF were observed, one between low and moderate jitter, and one between moderate and high jitter, over the range of jitter studied. Further studies are indicated to determine whether these boundaries are in fact disjoint or are both segments of a single larger phase transition boundary (possibly in additional dimensions), and also whether other such boundaries in jitter and load can be identified for these and other scheduling strategies. A final question of interest is the performance of strategies as the ratio of critical utilization to non-critical utilization is varied: in particular, at what point does adaptation become futile, and at what point is the feasibility of critical operations jeopardized even for strategies that attempt to isolate critical and non-critical processing.

Distributed dynamic scheduling: *i.e.*, whether coordinated multi-node dynamic and adaptive scheduling can demonstrate clear benefits end-to-end as well as locally. Of particular interest is whether variation in load at one location could be managed so its impact on other interdependent nodes could be reduced, and the operating conditions under which local versus global adaptation is indicated. Further empirical measurements are also needed to determine the impact of factors such as network latency on the end-to-end performance of dynamically scheduled distributed systems.

Available platform features: *i.e.*, what impact the availability or absence of various platform-specific features, such as preemptive multi-threading, would have on dispatching models and run-time scheduling behavior. This question is of particular interest for highly constrained embedded environments where memory footprint, power consumption, and timeliness considerations may dictate minimization and co-design of application, middleware, operating system, and even hardware features.

Application requirements: *i.e.*, what impact application specific requirements, such as policies for handling missed deadlines, *e.g.*,

1. retry versus cancellation,
2. early versus late cancellation, or
3. optimistic versus pessimistic cancellation,

might have on the effectiveness of strategies and overall application performance. Specifying and representing application requirements in a way visible to the scheduling and dispatching infrastructure appears beneficial to the levels of QoS that can be provided, as Section 9.2 examines in greater detail. In addition to *vertical* propagation of requirements, the effects of various end-to-end constraints such as bandwidth reservations or induced invocation rates are also of interest, particularly for multi-endsystem behaviors.

9.2 Improved Precision of Cancellation Decisions

With a more exact cancellation policy we might reduce the number of spurious cancellations and thus under particular conditions achieve an improvement in *effectiveness* as well as efficiency. Additional experiments are needed to assess the actual cost of cancellation in terms of *useful* work.

For example if missing a deadline for one of a chain of operations means the remaining operations are useless, then cancellation at that point may have a more beneficial effect on *total utility* than the are revealed by the simple assessment of deadlines made and missed, performed in the experiments described in Chapter 8. In general, experiments with a variety of application-specified utility functions are needed to assess cancellation mechanisms that are sensitive to application utility and can be tuned for appropriate levels of aggressiveness. These experiments will reveal the most effective ways to

1. preserve the resource for higher-value processing, while
2. minimizing overhead and unnecessary cancellation.

Clearly, a tension also exists between the overhead of more end-to-end and layer-to-layer sharing of information, and the impact that information can have on the effectiveness and efficiency of policies and mechanisms for key middleware QoS management services such as cancellation. As future work we will define and evaluate empirically models for managing the trade-offs and varying capabilities for *binding* information statically where possible, or *evaluating* information dynamically where necessary, in real-world systems.

9.3 Ordering, Transitions, and Multi-Dimensional QoS

A family of interesting problems relates to questions of the ordering of dispatching requests at run-time. These problems fall into three main categories: transition management, competitive constraint resolution, and cooperative constraint specification. In this section, we examine each of these categories in turn. Section 9.3.1 examines the problem of managing scheduling and dispatching invariants across transition boundaries. Section 9.3.2 considers the case where simultaneous requirements are in conflict or compete for a more basic property such as the ordering of dispatch requests, and must be resolved reasonably. Section 9.3.3 discusses the case where constraints may be applied cooperatively, to achieve a greater level of service in combination than separately.

9.3.1 Transition Management

An important question raised by this research is how dispatch requests can be managed effectively across adaptive transitions, whether in response to unanticipated events such as fault recovery requests, or to active adaptation to improve performance, as suggested in Section 9.4. The key problem is how to preserve invariants

1. in the pre-transition state,
2. in the post-transition state, and
3. across a consistent cut between states.

For example, consider the priority assignments under the *preemptive-by-priority-band* dispatching model described in Sections 5.2 and 5.3: in RMS priorities are assigned according to rate, and in MUF according to criticality. In addition, consider the queue ordering policy under the same dispatching model: in RMS dispatches within a priority level are ordered according to static subpriority, and in MUF according to laxity. Upon crossing from a feasible operating region to an overloaded operating with moderate jitter, such as between operating regions 6 and 7 in Figure 8.31 in Section 8.3.5, an adaptive transition from RMS to MUF might be performed.

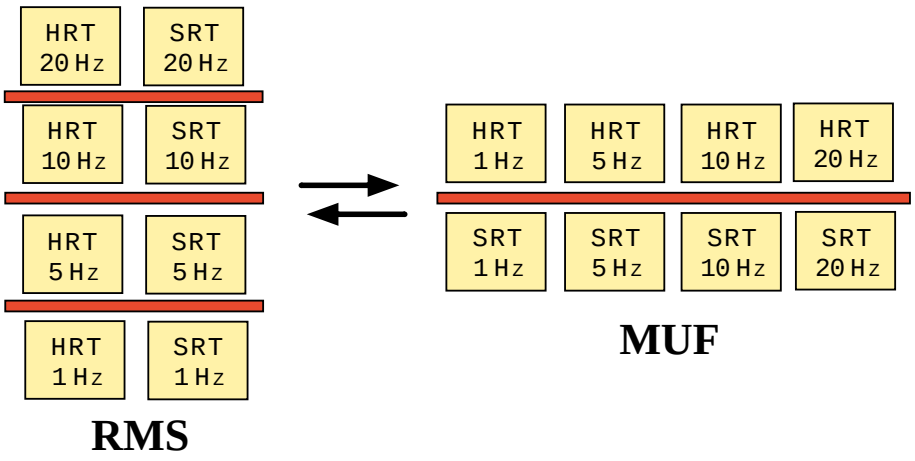

Figure 9.1: Adaptative Transition: RMS and MUF

Figure 9.1 illustrates the problem of reordering dispatch requests in a transition from RMS to MUF or vice versa. We show a pathological case where at each rate in RMS a soft real-time (SRT) dispatch request is enqueued ahead of a hard real-time

(HRT) dispatch request. We also note that the transitions between RMS and MUF are more challenging than those between RMS and RMS+MLF or between RMS+MLF and MUF, as follows:

- In transitions between RMS+MLF and MUF, a partial ordering of request priority is maintained across the transition: $\forall_{x \in HRT, y \in SRT}\ x > y$
- In transitions between RMS and RMS+MLF, only SRT operations need to change priority levels, fanning out from one laxity queue to multiple static queues, or vice versa.
- In transitions between RMS and MUF, both SRT and HRT operations need to change priority levels, and no partial order of priorities is maintained across the transition

This particular case poses several interesting problems, some theoretical, and some pragmatic. The main problems of theoretical interest are how to:

1. enforce the key *invariants* (*i.e.*, timeliness of HRT operations) of the previous operating region context before the transition,
2. enforce the key invariants of the next operating region context after the transition,
3. preserve key invariants that span the transition between the operating regions, as the context shifts from the previous operating region to the next one, and
4. to the extent possible maximize attainment of *goals* such as timeliness of SRT operations both in the previous and next operating regions, and across the transition.

We plan to investigate the question of how to manage partial orders to preserve invariants while also maximizing goals such as effectiveness in meeting SRT deadlines. For example, in an operating region where HRT feasibility can be assured, but not SRT, we would choose, *e.g.*, MUF over RMS and schedule HRT and SRT operations in separate ordered *criticality* partitions. If we then entered an operating region where both SRT and HRT operations are feasibly schedulable under RMS we might apply a different partial ordering, according to *rate*. Clearly, a transition involves switching between the criticality and rate partial orderings, but managing the intermediate context during the transition must be shown to still preserve the key invariants across the transition.

Pragmatic considerations include how to:

1. define and implement the necessary data structures and algorithms,
2. provide necessary parameters to system services (*e.g.*, setting thread priorities),
3. provide efficient additions to the infrastructure described in Chapters 4, 5, 6, and 7, and
4. define appropriate resource management policies

to extend the Kokyu scheduling and dispatching infrastructure so that it can:

1. maintain enforcement of changing partial orders of dispatch requests,
2. rapidly reassign partial orders to requests,
3. preserve invariants while reassigning partial orders, and
4. do all of this correctly while minimizing overhead and complexity.

9.3.2 Competitive Constraint Resolution

When two or more constraints are in conflict or compete for a more basic property such as the ordering of dispatch requests, we must determine some reasonable resolution. Such conflicts may arise either

- transiently – *e.g.*, when changing from one partial order to another as described in Section 9.3.1, or
- persistently – *e.g.*, when the ordering for fault-tolerance or transactional semantics is in conflict with the ordering for timeliness properties.

In the latter case, we plan to investigate whether it is possible to weaken slightly the invariants of one or more of the competing requirements, so that a solution can be obtained that still provides necessary properties. For both cases we will seek to identify common representations of the problem across different requirements. Rufus, *et al.*, describe an adaptive transition technique for local controllers in autonomous aerial vehicles [112] based on fuzzy neural models [113].

To provide assurances in multiple QoS dimensions such as timeliness and message ordering, however, we would prefer a representation that may be analyzed symbolically. In particular, planned schedules offer a substrate capable of expressing both

timeliness and ordering, so manipulating invariants over planned schedules appears a preferable alternative to non-symbolic approaches. We believe, however, that our approach could be combined readily with fuzzy and neural approaches in the contexts to which they are being applied, notably an open control platform for unmanned aerial vehicles [138, 137, 50].

More generally, we plan to approach these problems by

1. determining the extent to which they can be reduced to well-known problems such as, *e.g.*, distributed snapshots [14, 86],

2. applying specific information (*e.g.*, that rates of execution are harmonically framed) to the common representation to further reduce the particular problem, and

3. applying an invariant-based analysis similar to that in Section 9.3.1 to the remaining problem.

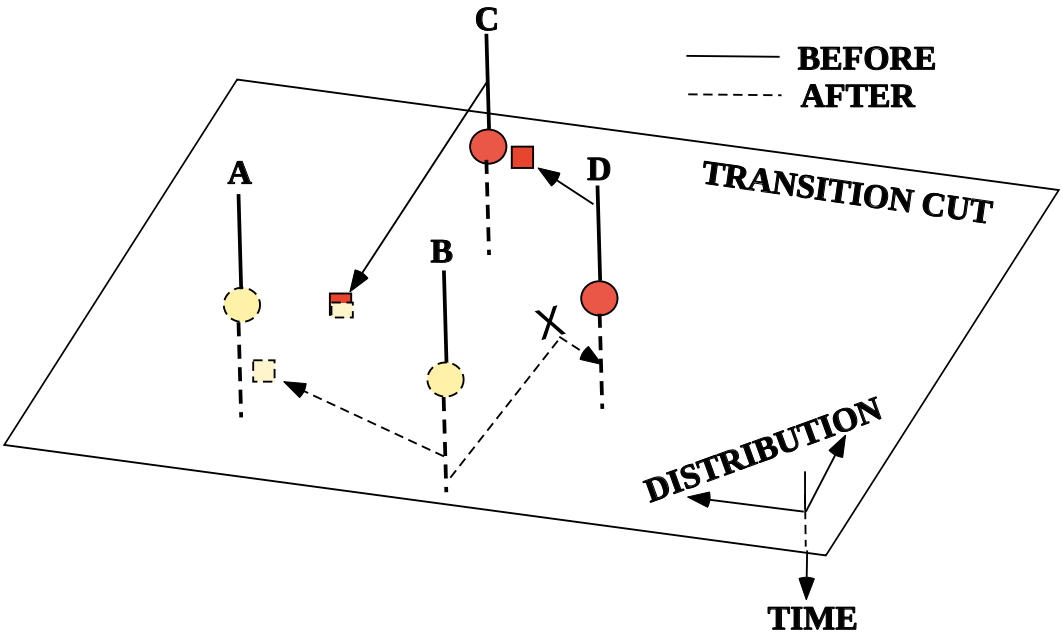

Figure 9.2: Distributed Transition Cut

Figure 9.2 illustrates the problem of maintaining consistent end-to-end timeliness properties across a distributed *cut* due to adaptation at multiple endsystems. Not only must each endsystem perform its own consistent transition locally as described in Section 9.3.1, it must also ensure that:

- properties of dispatches from an endsystem that *has not* made the transition are maintained upon receipt by an endsystem that *has not* made the transition
- properties of dispatches from an endsystem that *has* made the transition are maintained upon receipt by an endsystem that *has* made the transition
- properties of dispatches from an endsystem that *has not* made the transition are *transformed* [19] upon receipt by an endsystem that *has* made the transition
- properties of dispatches from an endsystem that *has* made the transition *cannot be applied* to another endsystem until it too has also made the transition

For example, endsystems A and B have made a transition in Figure 9.2, but endsystems C and D have not. C and D can exchange dispatch requests, which are processed under the invariants established before the transition. Similarly, A and B can exchange dispatch requests, but these are processed under the invariants established following the transition. C and D can send dispatch requests to A and B, in which case they pass through the distributed cut, and are transformed upon arrival as appropriate to the transition context. C and D cannot, however, handle dispatch requests from A or B until they have completed their own transitions, as requests cannot travel from the future to the past of a consistent cut. To ensure consistency in standard asynchronous message passing approaches we can either

1. advance the receiver immediately to the future of the cut [62, 86], *i.e.*, forcing it to make the transition upon receipt of the first dispatch from another endsystem that has made the transition, or
2. delay a request from an endsystem in the *future* of the cut until the receiver is also in the future of the cut [135, 86].

Unfortunately, with real-time constraints, delaying a request may amount to cancellation, as it may not then be able to make its associated deadline. Furthermore, forcing an endsystem across a transition boundary may not be immediately achievable without some form of distributed coordination protocol, *e.g.*, in the case of dependencies across endsystems. Hybrid approaches appear promising, *i.e.*, by delaying SRT requests and forcing a transition only over HRT requests we might be able in practice to improve the predictability and shorten the interval of convergence of the transition.

Finding solutions to these problems will allow adaptive scheduling in middleware to be applied across endsystems, presumably under the control of a (likely also distributed) higher level transactional commit protocol to coordinate transitions at each endsystem. Furthermore, it offers a general approach to reconciling other potentially competing but desirable properties such as timeliness and fault-tolerance.

9.3.3 Cooperative Constraint Specification

While some properties, such as timeliness and fault-tolerance, may introduce conflicting requirements, another question is whether constraints might be added or integrated cooperatively to achieve new system properties. Because real-time systems are sensitive to variations in latency, denying service to a real-time system can be simply a matter of *delaying* a request, rather than interdicting it entirely. However, many of the same policies and mechanisms used to provide real-time QoS assurances can be used to make a system more resilient in the face of an attack, thus ameliorating this vulnerability to some extent.

Webber, *et al.*, term systems with increased resistance to malicious attack even in the face of an untrustworthy environment, *defense-enabled* [134, 101]. They further distinguish between *protection* in which attempts are made to prevent an attacker from gaining access to the system, and *defense*, which includes protection but also seeks to delay or divert an attack if protection fails. Cuckier, *et al.*, state the goal of defense enabled systems as increasing either the *probability* or the *length* of survival of an attack [22], even when the attacker gains access to parts of the trusted computing base (TCB) [134].

The key insight in applying flexible and adaptive scheduling techniques to defense-enabled systems is that QoS monitoring and control models and infrastructure form a basis for defense [100, 73]. In real-time systems, there is a significant similarity to the effects of faults, overload, and attack. Furthermore, adaptation of the kinds enabled by the Kokyu framework may be employed to relieve these effects. All the approaches suggested in this section seek to:

1. identify the extent to which QoS management policies and mechanisms constitute vulnerabilities,

2. identify the capabilities those same policies and mechanisms provide to enable defense,

3. minimize accessibility of sensitive information by untrusted observers, and

4. exploit internal degrees of freedom within the structure of externally visible constraints, to reduce vulnerability to information that is accessible outside the TCB.

For example, if two end-systems exchange dispatch requests with advertised timing requirements, neither may in general assume the other uses a particular mechanism to meet that externally visible policy. One endsystem might assign priorities to preserve sufficient preemptive access to resources to ensure timely completion, while the other might use a planned schedule to achieve the same assurance.

Applying a modeling discipline such as I/O Automata [86] to these considerations appears a useful area of future work, to distinguish more formally between *internal* and *external* actions and whether particular *executions* of a model meet both internal and external constraints. Furthermore, support for security in a real-time environment may require simultaneous resolution of constraints as in Section 9.3.2, *e.g.*, for timeliness and message ordering for group membership [22].

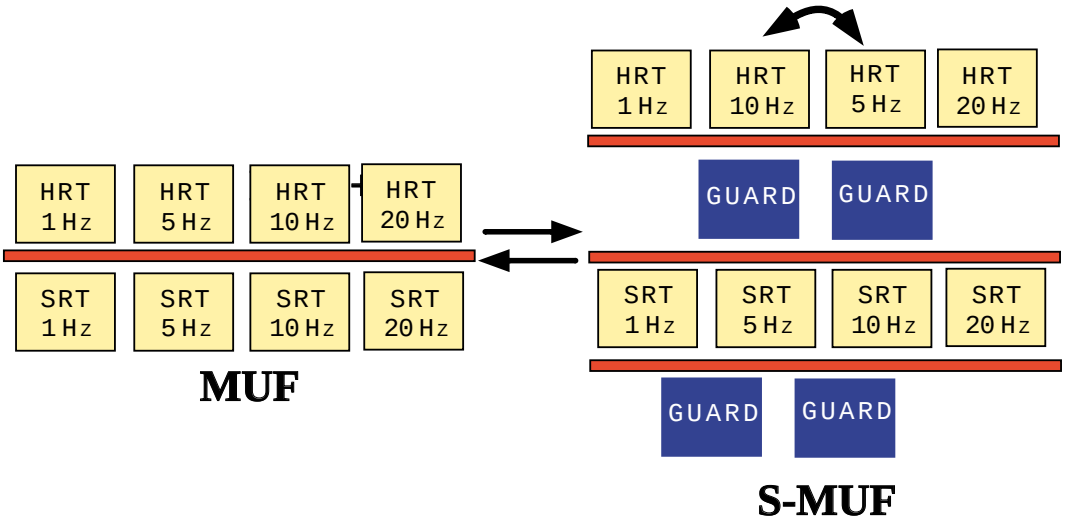

Figure 9.3: Cooperative Embedding of Constraints in Heuristics

We categorize the kinds of defensive responses to which flexible and adaptive scheduling may be employed as follows:

1. intrusion detection
2. adaptive resiliance to damage
3. resistance to monitoring and steering

Figure 9.3 shows an example of how a single scheduling heuristic, MUF might be transformed for defensive purposes into a notional strategy called Secure-MUF (S-MUF). We conclude this section by considering how each of the above categories of defensive responses might be enabled by this kinds of transformation.

Intrusion Detection: An analogy to security firewalls [11, 21, 63] is useful when considering intrusion detection. A number of security protection features are already available to middleware based on standards and patterns, such as the Common Object Request Broker Architecture (CORBA) Security Service [93], or the ability to plug in alternative communication protocols [58] for secure connections between trusted endsystems. However, defense-enabled systems must go beyond these protections, and respond at every level to the possibility that some of these may be compromised.

Assuming an attacker can intercept a GIOP message [34] and perform a form of stateful packet inspection (SPI) [21] on it, request priorities [91] or dynamic scheduling information [94] may be completely visible outside the trusted endsystems. A first line of defense is to detect an attacker that tries to exploit this information. The level of sophistication of the attack is important: if a source can be identified, a filter could be placed to block further attack from that source [11, 21]. However, a more patient attacker might probe for vulnerabilities in less readily identifiable ways, and detecting any change in the QoS context is beneficial to avoid a larger intrusion [21].

Decoys of several kinds are useful both for detection [21] and for wasting an attacker's effort [134]. For example, *guard* operations that are never invoked by a trusted endsystem can be run at various priority levels as shown in Figure 9.3, and their invocation would signal an attack. At higher priority levels, a guard could limit its execution to simply setting a flag to avoid exacerbating a denial of service attack. At lowest priority levels a guard could execute for an arbitrary duration to log more information and delay the progress of the attacker.

Detection accuracy may improve with multiple forms of detection [73]. Another form of detection would be to place guard operations with known execution durations at vulnerable points in the schedule. For example in an operating region with very little

jitter, we could pad the end of a feasible schedule with another kind of guard operation request so that if any of these missed its deadline or was cancelled we would suspect the onset of a change in the QoS context, which could trigger adaptation whether the change was due to a change of operating region, a fault, or even an attack.

Adaptive Resiliance to Damage: A particularly difficult defense is that against attacks indistinguishable from QoS failures, *e.g.*, changing a sensor rate to give bad data [101] or cause overuse of resources. For example, an attacker gaining unrestricted access to one endsystem might modify the `RT_Info` data structures described in Chapter 3 to use the scheduling and dispatching infrastructure on that endsystem to attack another endsystem. It would be difficult for the target endsystem to detect whether, *e.g.*, a suddenly greater rate of arrival of requests for an operation were due to an attack, a fault, or simply a change in operating region.

Adaptation to restore system properties is beneficial whatever the source of change in QoS context, though for defense-enabled systems adaptation must be combined with additional features. Trusted endsystems would likely have agreed on the rates of requests sent between them, so simply changing the rate of a request, or even adding jitter to the time at which a request is sent might be detected. Voting schemes might be applied across trusted endsystems to increase the probability of detection if one or more of the endsystems became faulty or was compromised. Unfortunately, agreement is not possible in the general case for distributed asynchronous invocations in the face of arbitrary faults [86]. Randomized algorithms could be used to offer probabilistic assurances of agreement, though at a cost of additional overhead and system complexity.

Resistance to Monitoring and Steering: Randomization is also useful to make it more difficult for an attacker to monitor and predict or even *steer* adaptation. For example, in Figure 9.3, the 10 Hz and 5 Hz HRT operations are shown in reverse order in the highest priority queue in S-MUF compared to MUF. In general, an endsystem may randomly

1. permute orders of requests that are not otherwise constrained,
2. choose another heuristic that still meets all constraints during adaptation,
3. add or remove execution jitter within known ranges of tolerance.

Hysteresis control for adaptation is also important, as it increases the difficulty of an attacker's attempts to induce or control adaptation, making the target endsystem

"balky" to an attacker. Finally, we might wish to reduce both external monitoring and steering risks by

1. marking trusted operation requests with digital signatures, *e.g.*, based on an agreed upon one-time-pad, then

2. virtualizing access to resources, and dispatching untrusted requests at random in slots that do not compete for resources with operations that matter to the system.

9.4 Towards Control Automata for Adaptation

Sections 8.5 and 9.3 motivate the use of flexible and adaptive scheduling and dispatching to address a variety of QoS considerations in distributed real-time systems. Applications of these techniques include:

1. adaptive management of transitions between system operating regions as in Section 9.3.1,

2. co-scheduling applications and resource managers as in Section 1.4.5,

3. resolving QoS requirements, such as for fault-tolerance and timeliness as in Section 9.3.2, and

4. participating in defense against attacks as in Section 9.3.3.

To achieve these goals, however, a significant level of fidelity to the actual constraints of the system must be achieved and demonstrated empirically. Fundamentally, what is needed is forms of adaptive *control* [124] of scheduling and dispatching.

Given the correlation between the performance of heuristics and measured mean HRT operation latencies described in Section 8.4, simple laws might be constructed using HRT execution latencies and HRT and SRT operation deadline statistics to control adaptive transitions among the scheduling strategies. Combining the measured feasible SRT threshold value described in Section 8.4.1 with the mean HRT latency value described in Section 8.4.2, we obtain a simple adaptation control law over the strategies without cancellation, as follows:

- if we are below the feasible threshold (by definition not missing deadlines) we should be using RMS; otherwise

- if the mean HRT latency over a window of size 20 is between 8 msec and 16 msec we should be using MUF; otherwise
- we should be using RMS+MLF.

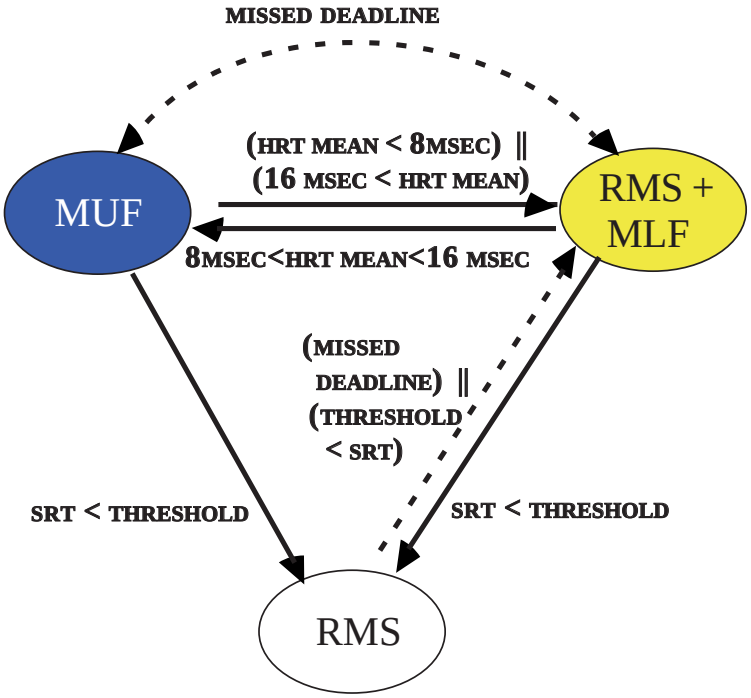

Figure 9.4: Adaptation Automaton over RMS, MUF, and RMS+MLF

A comparably simple automaton can be constructed for this control law, as figure 9.4 illustrates. The dashed lines indicate transitions that are taken immediately, once their associated condition is detected to be true. Solid lines indicate transitions that may require their associated condition to be true for some minimum number of samples, to avoid hysteresis. The transitions shown with dashed lines are designed to react quickly to failures or impending failures, *i.e.*, to avoid (or at least reduce the number of) HRT deadline failures. The solid line transitions are designed to improve performance, and allow the system to seek that improvement monotonically while avoiding the overhead of transitions that do not produce a meaningful improvement in performance.

To demonstrate the utility of this approach, and to apply it to real-world applications, two remaining open questions must be answered:

- What control laws for adaptation between dominant strategies can be identified and verified empirically for various classes of rate-based distributed applications?

- Where control laws cannot be identified more generally, is it possible to learn laws for adaptive control for a particular application, and what learning techniques would be most useful?

We conclude by considering experiments to assess the validity of and possibly expand on the adaptive control automaton shown in Figure 9.4, and examining the question of whether adaptive control laws could be learned for a particular (previously unspecified) application.

Empirical Studies: We must first ask how predictive is the automaton shown in Figure 9.4 over a broader category of applications. We are interested in how well it can predict performance of scheduling heuristics over a range of different OFP applications. We are then interested in its applicability to rate-based distributed real-time systems in other application domains, using the experiments suggested in Section 9.1. As we learn more from these studies, we anticipate an evolution of our models and infrastructure for flexible and adaptive scheduling and dispatching, to support new kinds of adaptive control.

Learning Adaptive Control Laws: Where possible, we would like to find control laws that are general across families of applications. Where necessary, however, machine learning techniques might be applied to obtain the necessary control variables and laws to meet adaptive QoS requirements for a particular application. Key questions in this area include:

1. What kinds of learning are most applicable?
2. What models of adaptive control can be learned?
3. Could learning be used to mine basic attributes useful in formulating control laws, or must a more complete basis be provided *a priori*?
4. Could learning be used to categorize these attributes and define more generally applicable control laws?
5. What kinds of training data would be available and useful to increase speed of learning and effectiveness of the resulting control laws?

Appendix A

Real-Time Scheduling Terminology

Precise terminology is necessary to describe and evaluate static, dynamic, and hybrid scheduling strategies. In this appendix, we define a number of terms used throught this dissertation.

RT_Operation and RT_Info: In TAO, an `RT_Operation` is a scheduled CORBA operation [117]. In this dissertation, we use the term *operation* interchangeably with `RT_Operation`. An `RT_Info` structure is associated with each operation and contains its QoS parameters. The `RT_Info` structure contains the following operation characteristics described below:

• **Criticality:** Criticality is an application-supplied value that indicates the significance of a CORBA operation's completion prior to its deadline. Higher criticality should be assigned to operations that incur greater cost to an application if they fail to complete execution before their deadlines. Some scheduling strategies, such as MUF, give greater priority to more critical operations than to less critical ones.

• **Worst-case execution time:** This is the longest time required to execute a single dispatch of an operation. Worst case execution times may be determined through techniques like simulation, instruction counting, or benchmarking on the target platform.

• **Period:** Period is the interval between dispatches of an operation.

• **Importance:** Importance is a lesser indication of a CORBA operation's significance. Like its criticality, an operation's importance value is supplied by an application. Importance is used as a "tie-breaker" to assign a unique static subpriority for each operation.

• **Dependencies:** An operation may *depend* on data produced by another operation. An operation that depends on the data from another operation may execute only after the other operation has completed.

Scheduling strategy: A scheduling strategy transforms the information from an operation's `RT_Info` by (1) assigning an *urgency* to the operation based on its static priority, dynamic subpriority, and static subpriority values, (2) mapping urgency into dispatching priority and dispatching subpriority values for the operation, and (3) providing dispatching queue configuration information so that each operation can be dispatched according to its assigned dispatching priority and dispatching subpriority. The key elements of this transformation are defined as follows:

• **Urgency:** Urgency [126] is an ordered tuple consisting of (1) static priority, (2) dynamic subpriority, and (3) static subpriority. Static priority is the highest ranking priority component in the urgency tuple, followed by dynamic subpriority and then static subpriority, respectively. Figure A.1 illustrates these relationships.

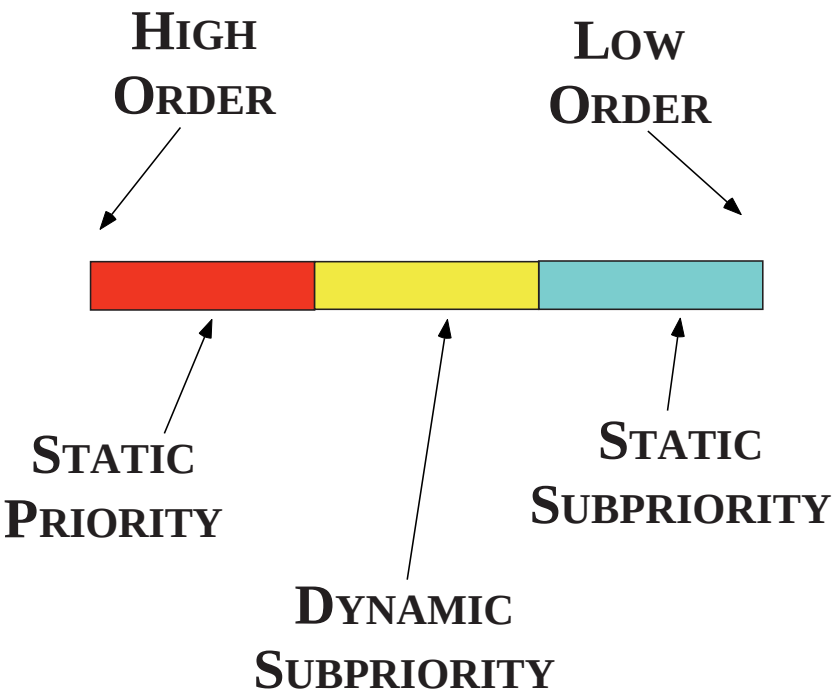

Figure A.1: Relationships in the Urgency Tuple

• **Static priority:** Static priority assignment establishes a fixed number of priority partitions into which all operations must fall. The number of static priority partitions is established off-line. An operation's static priority value is often determined off-line. However, the value assigned a particular dispatch of the operation could vary at run-time, depending on which scheduling strategy is employed.

• **Dynamic subpriority:** Dynamic subpriority is a value generated and used at run-time to order operations *within* a static priority level, according to the run-time and static characteristics of each operation. For example, a subpriority based on the operation with the "closest deadline" must be computed dynamically.

• **Static subpriority:** Static subpriority values are determined prior to run-time. Static subpriority acts as a tie-breaker when both static priority and dynamic subpriority are equal.

• **Dispatching priority:** An operation's dispatching priority corresponds to the real-time priority of the thread in which it will be dispatched. Operations with higher dispatching priorities are executed in threads with higher real-time priorities.

• **Dispatching subpriority:** Dispatching subpriority is used to order operations within a dispatching priority level. Operations with higher dispatching subpriority are executed ahead of operations with the same dispatching priority, but with lower dispatching subpriority.

• **Queue configuration:** A separate queue must be configured for each distinct dispatching priority. The scheduling strategy assigns each queue a dispatching type, *e.g.*, static, deadline, or laxity [1] ; a dispatching priority; and a thread priority.

Together, urgency and dispatching (sub)priority assignment specify requirements that certain operations will meet their deadlines. To support end-to-end QoS requirements, operations with higher dispatching priorities *should not* be delayed by operations with lower dispatching priorities. Two research challenges must be resolved to achieve this goal: (1) strategies must be identified to correctly specify end-to-end QoS requirements for different operations and (2) dispatching modules must enforce these end-to-end QoS specifications. The following two definitions are useful in addressing these challenges:

[1] An operation's laxity is the time until its deadline minus its remaining execution time.

• **Critical set:** The critical set consists of all operations whose completion prior to deadline is crucial to the integrity of the system. If all operations in the critical set can be assured of meeting their deadlines, a schedule that preserves the system's integrity can be constructed.

• **Minimum critical priority:** The minimum critical priority is the lowest dispatching priority level to which operations in the critical set are assigned. Depending on the scheduling strategy, the critical set may span multiple *dispatching priority* levels. To ensure that the critical set is schedulable, all operations at the minimum critical priority level must be schedulable.

In Kokyu, scheduling strategies rely primarily on priority- and subpriority-based dispatching, which can be enforced efficiently either by mechanisms available in the OS kernel (*e.g.*, preemptive thread priorities) or can be implemented efficiently in middleware (*e.g.*, dynamic subpriorities). Other scheduling strategies, such as Time-based Scheduling [133] and FIFO-r [130], use additional characteristics to order the dispatches of operations. These characteristics include:

• **Resource share:** Resource share is a measure of an operation's appropriate share of a resource (*e.g.*, CPU time), and is used to ensure fairness among operations that are not otherwise prioritized. For example, a share-based scheduling strategy might maintain information about each operation's past execution time. This information could be used to dispatch operations so that within every priority level each operation consumes CPU time proportional to its fair share.

• **Timing constraints:** Timing constraints capture explicit requirements for operation dispatch and completion times. For example, a timing constraint might specify that an operation must be dispatched within T time units after another operation completes.

Dispatching module: A dispatching module is responsible for (1) constructing the appropriate type of queue for each dispatching priority and (2) assigning each dispatching thread's priority to the value provided by the scheduling strategy. A TAO ORB endsystem can be configured with dispatching modules at several layers, including the I/O subsystem [59], ORB Core [118], and/or the Event Service [41].

References

[1] T. Abdelzaher, S. Dawson, W.-C.Feng, F.Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. Shin, Z. Wang, and H. Zou. ARMADA Middleware Suite. In *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, San Francisco, CA, December 1997. IEEE.

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

[3] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, and Michael Kircher. Applying C++, Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks. *C++ Report*, 12(3), March 2000.

[4] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, Michael Kircher, and Jeff Parsons. The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. In *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, April 2000.

[5] Alia Atlas and Azer Bestavros. Statistical Rate Monotonic Scheduling. In *The 19th IEEE Real-Time Systems Symposium (RTSS '98)*, Madrid Spain, December 1998.

[6] Alia Atlas and Azer Bestavros. Design and Implementation of Statistical Rate Monotonic Scheduling in KURT Linux. In *The 20th IEEE Real-Time Systems Symposium (RTSS '99)*, Phoenix AZ, December 1999.

[7] Neil Audsley and Andy Wellings. Analysing APEX Applications. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 39–44, December 1996.

[8] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, Reading, Massachusetts, 1999.

[9] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Optimal Reward-Based Scheduling for Periodic Real-Time Tasks. *IEEE Transactions on Computers*, 50(2):111–129, February 2001.

[10] A. Bavier, L. Peterson, and D. Mosberger. BERT: A Scheduler for Best Effort and Realtime Tasks. Technical Report TR-602-99, Princeton University, 1999.

[11] Uyless Black. *Internet Security Protocols*. Prentice Hall, New Jersey, 2000.

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

[13] Center for Distributed Object Computing. TAO: A High-performance, Real-time Object Request Broker (ORB). www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[14] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[15] J.-Y. Chung, J. W.-S. Liu, and K.-J. Lin. Scheduling Periodic Jobs that Allow Imprecise Results. *IEEE Transactions on Computers*, 39(9):1156–1174, September 1990.

[16] The Boeing Company. Open Systems Core Avionics Requirement (OSCAR). http://www.acq.osd.mil/osjtf/pdf/oscar.pdf.

[17] G. Cooper, L. Cingiser DiPippo, L. Esibov, R. Ginis, R. Johnston, P. Kortman, P. Krupp, J. Mauer, M. Squadrito, B. Thuraisingham, S. Wohlever, and V. Fay Wolfe. Real-Time CORBA Development at MITRE, NRaD, Tri-Pacific and URI. In *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, San Francisco, CA, December 1997. IEEE.

[18] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT, 1990.

[19] Angelo Corsaro, Douglas C. Schmidt, Ron K. Cytron, and Chris Gill. Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Disciplines in Open Distributed Real-Time Systems. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications.*, pages 289–299, Rome, Italy, September 2001. OMG.

[20] Joseph K. Cross and Douglas C. Schmidt. Meta-Programming Techniques for Distributed Real-time and Embedded Systems. In *Proceedings of the* 7th *Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 2002. IEEE.

[21] Jeff Crume. *Inside Internet Security*. Addison-Wesley, Harlow, England, 2000.

[22] Michel Cukier, James Lyons, Prashant Pandey, HariGovind V. Ramasamy, William H. Sanders, Partha P. Pal, Franklin Webber, Richard E. Schantz, Joseph Loyall, Ronald Watro, Michael Atighetchi, and Jeanna Gossett. Intrusion Tolerance Approaches in ITUA. In *Fast Abstract in Supplement of the 2001 International Conference on Dependable Systems and Networks*, July 2001.

[23] Z. Deng and J. W.-S. Liu. Scheduling Real-Time Applications in an Open Environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1997.

[24] Bryan S. Doerr and David C. Sharp. Freeing Product Line Architectures from Execution Dependencies. In *Proceedings of the 11th Annual Software Technology Conference*, April 1999.

[25] Bryan S. Doerr, Thomas Venturella, Rakesh Jha, Christopher D. Gill, and Douglas C. Schmidt. Adaptive Scheduling for Real-time, Embedded Information Systems. In *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 1999.

[26] Victor Fay-Wolfe, John K. Black, Bhavanai Thuraisingham, and Peter Krupp. Real-time Method Invocations in Distributed Environments. Technical Report 95-244, University of Rhode Island, Department of Computer Science and Statistics, 1995.

[27] W. Feng, U. Syyid, and J.W.-S. Liu. Providing for an Open, Real-Time CORBA. In *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, San Francisco, CA, December 1997. IEEE.

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[29] Christopher D. Gill, Ron Cytron, and Douglas C. Schmidt. Middleware Scheduling Optimization Techniques for Distributed Real-Time and Embedded Systems. In *Proceedings of the 7th Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 2002. IEEE.

[30] Christopher D. Gill, Joseph W. Hoffert, David C. Sharp, and Patrick 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)*, October 2001.

[31] Christopher D. Gill, David L. Levine, Carlos O'Ryan, and Douglas C. Schmidt. Distributed Object Visualization for Sensor-Driven Systems. In *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 1999.

[32] Christopher D. Gill, David L. Levine, and Douglas 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*, 20(2), March 2001.

[33] Aniruddha Gokhale and Douglas C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In *Proceedings of SIGCOMM '96*, pages 306–317, Stanford, CA, August 1996. ACM.

[34] Aniruddha Gokhale and Douglas C. Schmidt. Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance. In *Hawaiian International Conference on System Sciences*, January 1998.

[35] Aniruddha Gokhale and Douglas 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, 17(9), September 1999.

[36] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics, A Foundation for Computer Science*. Addison Wesley, Reading, Massachusetts, 1990.

[37] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computers*, 44:1443–1451, December 1995.

[38] Ching-Chih Jason Han. A Better Polynomial-Time Schedulability Test for Real-Time Multiframe Tasks. In *IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998. IEEE.

[39] Ching-Chih Jason Han and H. Ying Tyan. A Better Polynomial-Time Schedulability Test for Real-Time Fixed-Priority Scheduling Algorithms. In *IEEE Real-Time Systems Symposium*, San Francisco, CA, December 1997. IEEE.

[40] Hansson, Lawson, Bridal, Eriksson, Larsson, Lon, and Stromberg. BASEMENT: An Architecture and Methodology for Distributed Automotive Real-Time Systems. *IEEE Transactions on Computers*, 46(9):1016–1027, SEPTEMBER 1997.

[41] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997. ACM.

[42] Timothy H. Harrison, Carlos O'Ryan, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. *submitted to the Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 1998.

[43] Kenneth E. Iverson. *A Programming Language*. Wiley, 1962.

[44] J. Huang and R. Jha and W. Heimerdinger and M. Muhammad and S. Lauzac and B. Kannikeswaran and K. Schwan and W. Zhao and R. Bettati. RT-ARM: A Real-Time Adaptive Resource Management System for Distributed Mission-Critical Applications. In *Workshop on Middleware for Distributed Real-Time Systems, RTSS-97*, San Francisco, California, 1997. IEEE.

[45] Kevin Jeffay. The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, 1993.

[46] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser. Dynamic Scheduling for Soft Real-Time Distributed Object Systems. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Newport Beach, CA, March 2000. IEEE/IFIP.

[47] 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, November 2000. IEEE.

[48] 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, April 2001. IEEE.

[49] V. Kalogeraki, P.M. Melliar-Smith, and L.E. Moser. Soft Real-Time Resource Management in CORBA Distributed Systems. In *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, San Francisco, CA, December 1997. IEEE.

[50] S. Kannan, C. Restrepo, I. Yavrucuk, L. Wills, D. Schrage, and J.V.R. Prasad. Control Algorithm and Flight Simulation Integration Using the Open Control Platform for Unmanned Aerial Vehicles. In *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 2000.

[51] Khanna, S., *et al.* Realtime Scheduling in SunOS 5.0. In *Proceedings of the USENIX Winter Conference*, pages 375–390. USENIX Association, 1992.

[52] Gregor Kiczales. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[53] K. H. (Kane) Kim. Object Structures for Real-Time Systems and Simulators. *IEEE Computer*, pages 62–70, August 1997.

[54] Kane Kim and Eltefaat Shokri. Two CORBA Services Enabling TMO Network Programming. In *Fourth International Workshop on Object-Oriented, Real-Time Dependable Systems*. IEEE, January 1999.

[55] Michael Kircher and Douglas C. Schmidt. DOVE: A Distributed Object Visualization Environment. *C++ Report*, 11(2), March 1999.

[56] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, Norwell, Massachusetts, 1993.

[57] Fred Kuhns. Personal communication, November 2001.

[58] Fred Kuhns, Carlos O'Ryan, Douglas C. Schmidt, and Jeff Parsons. The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware. In *Proceedings of the IFIP* 6th *International Workshop on Protocols For High-Speed Networks (PfHSN '99)*, Salem, MA, August 1999. IFIP.

[59] Fred Kuhns, Douglas C. Schmidt, and David L. Levine. The Design and Performance of a Real-time I/O Subsystem. In *Proceedings of the* 5th *IEEE Real-Time Technology and Applications Symposium*, pages 154–163, Vancouver, British Columbia, Canada, June 1999. IEEE.

[60] Fred Kuhns, Douglas C. Schmidt, Carlos O'Ryan, and David Levine. Supporting High-performance I/O in QoS-enabled ORB Middleware. *Cluster Computing: the Journal on Networks, Software, and Applications*, 3(3), 2000.

[61] Ralph Lachenmaier. Open Systems Architecture Puts Six Bombs on Target. `http://www.cs.wustl.edu/~schmidt/TAO-boeing.html`, December 1998.

[62] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 26(7):558–565, 1978.

[63] Eric Larson and Brian Stephens. *Web Servers, Security, and Maintenance*. Prentice Hall, New Jersey, 2000.

[64] R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In *Proceedings of the* 2nd *Annual Conference on the Pattern Languages of Programs*, pages 1–7, Monticello, Illinois, September 1995.

[65] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society Press, 1989.

[66] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced Aperiodic Scheduling in Hard Real-Time Environments. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1987.

[67] David L. Levine, Christopher D. Gill, and Douglas C. Schmidt. Dynamic Scheduling Strategies for Avionics Mission Computing. In *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, November 1998.

[68] Baochun Li and Klara Nahrstedt. A Control-based Middleware Framework for QoS Adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632–1650, September 1999.

[69] Baochun Li and Klara Nahrstedt. Impact of Control Theory on QoS Adaptation in Distributed Middleware Systems. In *Proceedings of the 2001 American Control Conference*, June 2001.

[70] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1):46–61, January 1973.

[71] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, New Jersey, 2000.

[72] 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)*, pages 625–634. IEEE, April 2001.

[73] Joseph P. Loyall, Partha P. Pal, Richard E. Schantz, and Franklin Webber. Building Adaptive and Agile Applications Using Intrusion Detection and Response. In *Proceedings of NDSS 2000, the Network and Distributed System Security Symposium*, February 2000.

[74] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance Specifications and Metrics for Adaptive Real-Time Systems. In *The*

21st IEEE Real-Time Systems Symposium (RTSS '00), Orlando FL, December 2000.

[75] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and Evaluation of a Feedback Control EDF Scheduling Algorithm. In *The 20th IEEE Real-Time Systems Symposium (RTSS '99)*, Phoenix AZ, December 1999.

[76] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms. *Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 2002, to appear.

[77] Chenyang Lu. *Feedback Control Real-Time Scheduling*. PhD thesis, University of Virginia, Charlottesville, VA, May 2001.

[78] Charlie McElhone. Soft Computations within Integrated Avionics Systems. In *Proceedings of the IEEE National Aerospace and Electronics Conference (NAECON 2000)*, October 2000.

[79] Ashish Mehra, Atri Indiresan, and Kang G. Shin. Structuring Communication Software for Quality-of-Service Guarantees. *IEEE Transactions on Software Engineering*, 23(10):616–634, October 1997.

[80] Aloysius K. Mok and Deji Chen. A Multiframe Model for Real-Time Tasks. *IEEE Transactions on Software Engineering*, 23(1):635–645, October 1997.

[81] L. Molesky, K. Ramamritham, C. Shen, J. Stankovic, and G. Zlokapa. Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel, extended abstract. In *IEEE Workshop on Real-Time Operating Systems and Software*, May 1990.

[82] C. Montez, J. Fraga, R. Oliveira, and J.-M. Farines. An Adaptive Scheduling Approach in Real-Time CORBA. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*. IEEE/IFIP, 1999.

[83] A.B. Montz, D. Mosberger, S. W. O'Malley, L.L. Peterson, and T.A. Proebsting. Scout: A Communications-Oriented Operating System. In *5th Workshop on Hot Topics in Operating System*. IEEE Computer Society Press, May 1995.

[84] David Mosberger. *Scout: A Path-Based Operating System*. PhD thesis, University of Arizona, 1997.

[85] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the* 1st *Symposium on Operating Systems Design and Implementation*. USENIX Association, October 1996.

[86] Nancy Ann Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[87] John R. Newport. *Avionics Systems Design*. CRC Press, Boca Raton, Florida, 1994.

[88] Douglas Niehaus. Improving Support for Multimedia System Experimentation and Deployment. In Nabil R. Adam and Bharat Bhargava, editors, *Workshop on Parallel and Distributed Real-Time Systems*. Lecture Notes in Computer Science 1586, Parallel and Distributed Processing, Springer Verlag, 1999.

[89] Object Management Group. *CORBA Messaging Specification*, OMG Document orbos/98-05-05 edition, May 1998.

[90] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.2 edition, February 1998.

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

[92] Object Management Group. *Dynamic Scheduling Real-Time CORBA Joint Revised Submission*, OMG Document orbos/2000-08-12 edition, August 2000.

[93] Object Management Group. *Security Service 1.8 Specification*, OMG Document security/00-11-03 edition, November 2000.

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

[95] Shui Oikawa and Raj Rajkumar. Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. In *Proceedings of the* 5th *IEEE Real-Time Technology and Applications Symposium*, Vancouver, British Columbia, June 1999. IEEE.

[96] Carlos O'Ryan, Fred Kuhns, Douglas C. Schmidt, Ossama Othman, and Jeff 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, April 2000.

[97] Carlos O'Ryan, Douglas C. Schmidt, Fred Kuhns, Marina Spivak, Jeff Parsons, Irfan Pyarali, and David Levine. Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA 3.0. In *Proceedings of the* 6th *IEEE Real-Time Technology and Applications Symposium*, Washington DC, May 2000. IEEE.

[98] Carlos O'Ryan, Douglas C. Schmidt, David Levine, and Russell Noseworthy. Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations. In *Proceedings of the* 5th *Workshop on Object-oriented Real-time Dependable Systems*, Montery, CA, November 1999. IEEE.

[99] Carlos O'Ryan, Douglas C. Schmidt, and J. Russell Noseworthy. Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations. *International Journal of Computer Systems Science and Engineering*, 2002.

[100] Partha P. Pal, Joseph P. Loyall, Richard E. Schantz, John A. Zinky, and Franklin Webber. Open Implementation Toolkit for Building Survivable Applications. In *Proceedings of DISCEX 2000, the DARPA Information Survivability Conference and Exposition*, January 2000.

[101] Partha P. Pal, Franklin Webber, Richard E. Schantz, Michael Atighetchi, and Joseph P. Loyall. Defense-Enabling Using Advanced Middleware - An Example. In *Proceedings of Milcom 01*, October 2001.

[102] Irfan Pyarali. *Patterns for Providing Real-Time Guarantees in DOC Middleware*. PhD thesis, Washington University, St. Louis, MO 63130, December 2001. Defense December 18th, 2001.

[103] Irfan Pyarali, Tim Harrison, Douglas C. Schmidt, and Thomas D. Jordan. Proactor – An Architectural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events. In *The* 4th *Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34)*, September 1997.

[104] Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt. Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging. *USENIX Computing Systems*, 9(4), November/December 1996.

[105] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale. Applying Optimization Patterns to the Design of Real-time ORBs. In *Proceedings of the* 5th *Conference on Object-Oriented Technologies and Systems*, San Diego, CA, May 1999. USENIX.

[106] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale. Using Principle Patterns to Optimize Real-time ORBs. *IEEE Concurrency Magazine*, 8(1), 2000.

[107] Steve Rago. *UNIX System V Network Programming*. Addison-Wesley, Reading, Massachusetts, 1993.

[108] Ragunathan Rajkumar, Mike Gagliardi, and Lui Sha. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *First IEEE Real-Time Technology and Applications Symposium*, May 1995.

[109] K. Ramamritham, C. Shen, O. Gonzáles, S. Sen, and S. Shirgurkar. Using Windows NT for Real-time Applications: Experimental Observations and Recommendations. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998. IEEE.

[110] Krithi Ramamritham, John A. Stankovic, and Perng-Fei Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.

[111] John Regehr and Jay 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.

[112] F. Rufus, S. Clements, S. Sander, B. Heck, L. Wills, and G. Vachtsevanos. Software-Enabled Control Technologies for Autonomous Aerial Vehicles. In *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 2000.

[113] F. Rufus, B. Heck, and G. Vachtsevanos. Software-Enabled Adaptive Mode Transition Control for Autonomous Manned Vehicles. In *Proceedings of the 19th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 2000.

[114] Douglas C. Schmidt. GPERF: A Perfect Hash Function Generator. In *Proceedings of the 2nd C++ Conference*, pages 87–102, San Francisco, California, April 1990. USENIX.

[115] Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 529–545. Addison-Wesley, Reading, Massachusetts, 1995.

[116] Douglas C. Schmidt. A Family of Design Patterns for Application-level Gateways. *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, 2(1), 1996.

[117] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.

[118] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha 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*, 21(2), 2001.

[119] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[120] David C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In *Proceedings of the 10th Annual Software Technology Conference*, April 1998.

[121] David C. Sharp. Avionics Product Line Software Architecture Flow Policies. In *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 1999.

[122] Balaji Srinivasan, Shyamalan Pather, Robert Hill, Furquan Ansari, and Douglas Niehaus. A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software. In *Proceedings of the* 4th *IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998. IEEE.

[123] J. A. Stankovic, T. He, T. F. Abdelzaher, M. Marley, G. Tao, S. H. Son, and C. Lu. Feedback Control Scheduling in Distributed Systems. In *The 22nd IEEE Real-Time Systems Symposium (RTSS '01)*, London UK, December 2001.

[124] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao. The Case for Feedback Control Real-Time Scheduling. In *11th EuroMicro Conference on Real-Time Systems*, York UK, June 1999.

[125] Alex Stepanov and Meng Lee. The Standard Template Library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[126] David B. Stewart and Pradeep K. Khosla. Real-Time Scheduling of Sensor-Based Control Systems. In W. Halang and K. Ramamritham, editors, *Real-Time Programming*. Pergamon Press, Tarrytown, NY, 1992.

[127] David B. Stewart, D. E. Schmitz, and Pradeep K. Khosla. Implementing Real-Time Robotic Systems using CHIMERA II. In *Proceedings of 1990 IEEE International Conference on Robotics and Automation*, Cincinnatti, OH, 1992.

[128] D.A. Stuart, M. Brockmeyer, A.K. Mok, and F. Jahanian. Simulation-Verification: Biting at the State Explosion Problem. *IEEE Transactions on Software Engineering*, 27(7):599–617, 2001.

[129] Bhavani Thuraisingham, Peter Krupp, Alice Schafer, and Victor Wolfe. On Real-Time Extensions to the Common Object Request Broker Architecture. In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) Workshop on Experiences with CORBA*. ACM, October 1994.

[130] Hung-Ying Tyan and Jennifer C. Hou. A rate-based message scheduling paradigm. In *Fourth International Workshop on Object-Oriented, Real-Time Dependable Systems*. IEEE, January 1999.

[131] Statement of Work for the Extend Sentry Program, CPFF Project, ECSP Replacement Phase II, February 1997. Submitted to OMG in response to RFI ORBOS/96-09-02.

[132] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1997.

[133] Yu-Chung Wang and Kwei-Jay Lin. Implementing A General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel. In *IEEE Real-Time Systems Symposium*, pages 246–255. IEEE, December 1999.

[134] Franklin Webber, Partha P. Pal, Richard E. Schantz, and Joseph P. Loyall. Defense-Enabled Applications. In *Proceedings of the second DARPA Information Survivability Conference and Exposition (DISCEX II)*, June 2000.

[135] Jennifer Lundelius Welch. Simulating Synchronous Processors. *Information and Computation*, 74(2):159–171, 1987.

[136] L. R. Welch, B. A. Shirazi, B. Ravindran, and C. Bruggeman. DeSiDeRaTa: QoS Management Technology for Dynamic, Scalable, Dependable Real-Time Systems. In *IFACs 15th Symposium on Distributed Computer Control Systems (DCCS98)*. IFAC, 1998.

[137] L. Wills, S. Kannan, B. Heck, G. Vachtsevanos, C. Restrepo, S. Sander, D. Schrage, and J.V.R. Prasad. An Open Software Infrastructure for Reconfigurable Control Systems. In *Proceedings of the 19th American Control Conference (ACC-2000)*, June 2000.

[138] Linda Wills, Sam Sander, Suresh Kannan, Aaron Kahn, J.V.R. Prasad, and Daniel Schrage. An Open Control Platform for Reconfigurable, Distributed, Hierarchical Control Systems. In *Proceedings of the 19th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 2000.

[139] Wind River Systems. VxWorks 5.3. www.wrs.com/products/html/vxworks.html.

[140] Don C. Winter. Modular, Reusable Flight Software for Production Strike Aircraft. In *Proceedings of the 15th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pages 401–406, 1996.

[141] Victor Fay Wolfe, Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Igor Zykh, and Russel Johnston. Real-Time CORBA. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, Montréal, Canada, June 1997.

[142] E. Riseman Z. Zhu, K. Rajasekar and A. Hanson. Panoramic Virtual Stereo Vision of Cooperative Mobile Robots for Localizing 3D Moving Objects. In *Proceedings of the IEEE Workshop on Omnidirectional Vision (OMNIVIS'00)*. IEEE, 2000.

[143] John A. Zinky, David E. Bakken, and Richard Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.

Vita

Christopher D. Gill

Date of Birth January 12, 1965

Place of Birth Madison, Wisconsin, United States of America

Degrees B.A. *cum laude*, English and Biology, May 1987, Washington University
M.S. Computer Science, May 1997, University of Missouri-Rolla

Professional Societies ACM, IEEE, IEEE Computer Society, USENIX, OMG

Publications

Gill, C. D., Levine, D. L., Schmidt, D. C., "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, guest editor Wei Zhao, March 2001, Vol. 20 No. 2, pp. 117-154.

Levine, D. L., Gill, C. D., Schmidt, D. C., "Object Lifetime Manager - A Complementary Pattern for Controlling Object Creation and Destruction", Design Patterns in Communications Software, Linda Rising, ed., Cambridge University Press, 2001, pp. 495-534.

Gill, C. D., Hoffert, J. W., Sharp, D. C., and Goertzen, P. H. "An Evolution of QoS Context Propagation in Event-Mediated Avionics Software Architectures", Proceedings of the 20th IEEE/AIAA Digital Avionics System Conference, Daytona Beach, Florida, October 14-18, 2001.

Corsaro, A., Schmidt, D. C., Gill, C., Cytron, R., “Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Disciplines in Open Distributed Real-Time Systems”, 3rd International Symposium on Distributed Objects and Applications (DOA ’01), September 8-10, 2001, Rome, Italy.

Loyall, J., Gossett, J., Gill, C., Schantz, R., Zinky, J., Pal, P., Shapiro, R., Rodrigues, C., Atighetchi, M., and Karr, D., “Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications”, Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), Phoenix, Arizona, USA, April 16-19, 2001, pp. 625-634.

Gill, C. D., Levine, D. L., “Quality of Service Management for Real-Time Embedded Information Systems”, Proceedings of the 19th IEEE/AIAA Digital Avionics System Conference, Philadelphia, Pennsylvania, 7-13 October 2000.

Judkins, T. V., Gill, C. D., “Synthesizer, A Pattern Language for Designing Digital Modular Synthesis Software”, Proceedings of the 6th Pattern Languages of Programming Conference, Allerton Park, Illinois, USA, 13 – 16 August 2000.

Doerr, B., Venturella, T., Jha, R., Gill, C., Schmidt, D., “Adaptive Scheduling for Real-time, Embedded Information Systems”, Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference, St. Louis, Missouri, October 24-29, 1999.

Gill, C. D., Levine, D. L., O’Ryan, C., Schmidt, D. C., “Distributed Object Visualization for Sensor-Driven Systems”, 18th IEEE/AIAA Digital Avionics System Conference, St. Louis, Missouri, Oct 24-29 1999.

Levine, D. L., Gill, C. D., Schmidt, D. C., “Dynamic Scheduling Strategies for Avionics Mission Computing”, Proceedings of the 17th IEEE/AIAA Digital Avionics System Conference, Seattle, Washington, 31 October - 6 November 1998.

May 2002