Middleware Scheduling Optimization Techniques for Distributed Real-Time and Embedded Systems [*]

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

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

Abstract

Developers of mission-critical distributed real-time and embedded (DRE) systems face a fundamental tension between (1) the performance gains achievable with hand-crafted optimizations to systems built largely from scratch and (2) the development cost and cycle-time reductions offered by common off-the-shelf (COTS) middleware. This paper describes how the Kokyu portable middleware scheduling framework, which is built using standards-based COTS middleware and OS primitives, can be used both (1) to maintain the flexibility and re-use offered by COTS middleware and (2) to provide opportunities for domain-specific optimizations to meet stringent real-time performance requirements.

Keywords: Real-Time Middleware, Quality of Service Issues, Dynamic Scheduling Algorithms and Analysis, Adaptive Resource Management, Distributed Systems.

1 Introduction

Next-generation mission-critical distributed real-time and embedded (DRE) systems, such as integrated avionics mission computing systems [1], teams of collaborating emergency rescue robots [2], and distributed real-time automobile management systems [3], 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; (3) perform adaptive resource reallocations within firmly bounded time-scales.

While solutions built using standards-based COTS middleware promise greater re-use 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 [4]. Emerging COTS middleware approaches, such as Dynamic Scheduling Real-Time CORBA [5] and the RTSJ [6], 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) re-use and flexibility across families of systems and (2) optimized real-time performance in DRE systems, this paper describes the following enhancements to current real-time middleware scheduling approaches: (1) 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; (2) support for variable period tasks, to exploit degrees of freedom in performance of individual elements to achieve system-wide real-time properties; (3) 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 remainder of this paper is structured as follows: Section 2 gives an overview of (1) the target system for our optimizations: a research DRE avionics mission computing plat-

[*] This work was funded in part by Boeing and DARPA ITO.

form and (2) the *Kokyu*[1] scheduling framework; Section 3 describes optimizations for DRE target system performance under steady-state and adaptive conditions, and outlines extensions to our framework to support those optimizations; Section 4 presents qualitative and quantitative indications of the benefits of our approach, and describes experiments currently underway to quantify the benefits and associated costs of these optimizations; Section 5 surveys related work and describes how our work extends the state-of-the-art in middleware scheduling; and Section 6 offers concluding remarks and describes planned future work on scheduling middleware for DRE systems.

2 Overview of Target Platform and Kokyu Framework Infrastructure

This section describes key features of the platform upon which our work is based and the Kokyu scheduling and dispatching infrastructure within which we perform optimizations to that platform. Section 2.1 identifies the expected number and characteristics of schedulable tasks within the target platform itself. Section 2.2 describes primitive elements of Kokyu scheduling framework, focusing on how target platform tasks are mapped to the dispatching elements of this framework.

2.1 An Overview of the Target Platform

Figure 1 illustrates the architecture of the OO avionics mission computing platform [7] targeted by the scheduling optimizations we present in this paper. This platform was developed and deployed using OO middleware components and services based on CORBA [8]. 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.

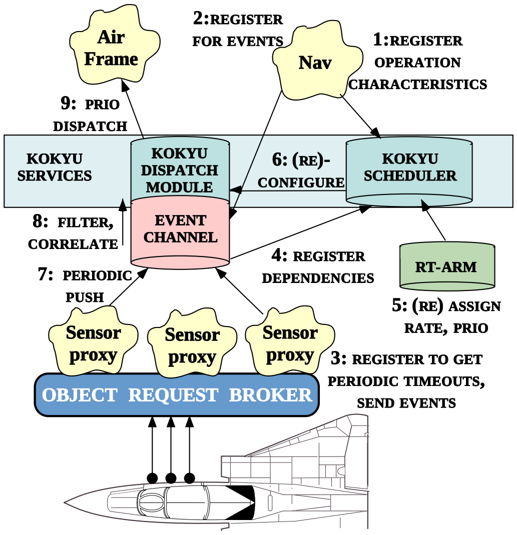

Figure 1: Avionics Example

Operations and Tasks: 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*.

The model that underlies our target platform differs somewhat from the model in [9]. 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.

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* (RT-ARM) [10]. RT-ARM is a middleware service developed by Honeywell that adapts the rates of tasks according to changing environmental conditions [11].

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 and dependencies may be enabled or disabled at run-time by the application or a middleware resource manager, such as the RT-ARM.

2.2 An Overview of Kokyu

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 2, Kokyu currently provides real-time scheduling and dispatching services for TAO's real-time CORBA Event Service [7] that mediates supplier-consumer relationships between application operations. Figure 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. In addition to the features described here, Kokyu will also be used to implement the standard Real-Time CORBA 1.0 [4] Scheduling Service specification, using the same underlying mechanisms.

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

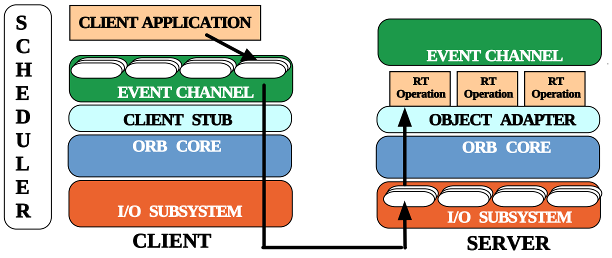

Figure 2: Kokyu Services used by TAO

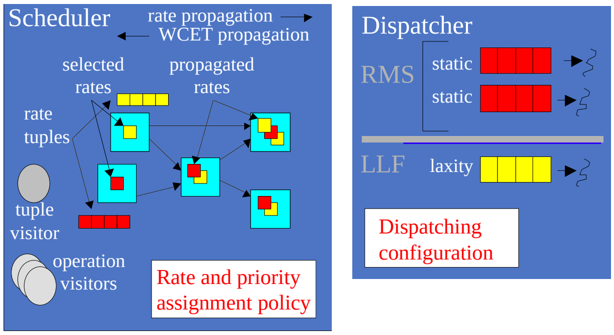

Figure 3: Kokyu Scheduling and Dispatching Infrastructure

Kokyu consists of two cooperating infrastructure segments, illustrated in Figure 3: (1) a pluggable scheduling infrastructure with efficient support for adaptive execution of diverse scheduling heuristics; and (2) flexible dispatching infrastructure that allows composition of primitive operating system and middleware mechanisms to enforce arbitrary scheduling heuristics. The combined framework provides 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, as we describe in Section 3.

2.2.1 Kokyu's Scheduling Infrastructure

Our earlier work on Kokyu's scheduling infrastructure [14] (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, Kokyu's scheduling infrastructure has 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.

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 3.3 considers these issues in detail.

2.2.2 Kokyu's Dispatching Infrastructure

The right side of Figure 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:

Dispatching queues: Each task is assigned by our strategized scheduling service [14] 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 with increasing queue number, so that the 0^{th} 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 3.2, 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.

This paper discusses three disciplines: static, deadline, and laxity. Static tasks are ordered by a static subpriority value, resulting in a 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 [15] (EDF) scheduling strategy. Finally, 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 [16] (MLF) scheduling strategy; laxity queues at different priority levels can be used to implement the maximum urgency first [16] (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 [9].

3 Scheduling Optimizations

Careful optimization of middleware is needed to meet the goals of mission-critical DRE systems described in Section 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 2.1.

3.1 Overview of System Modes

A *mode* is a Boolean function on the states of a system's constituent configuration items. For example, "the aircraft is engaged with ground threats" is a mode, and "all sensors are in their operational states" 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 paper, 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 described in Section 2. As illustrated in Figure 4, the steady-state mode partition contains any steady behavioral state, with a particular rate and priority assigned to each operation while in that state.

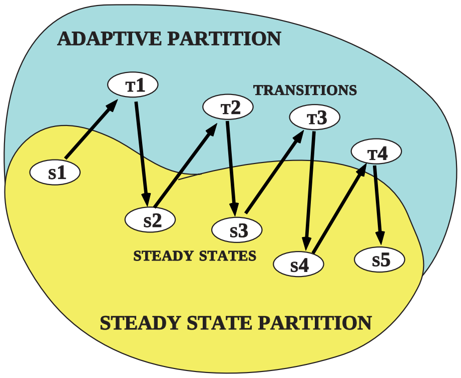

Figure 4: System Mode Partitions

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 3.2 describes optimizations to the steady-state mode partition, and Section 3.3 describes optimizations to the adaptive mode partition. In our current research, the RT-ARM described in Section 2 is invoked from the steady state mode partition, but may transition the system into the adaptive mode partition during its execution.

3.2 Steady-State Optimizations

Existing research [9, 17] 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 [9, 16]. Clearly, a variety of scheduling approaches and hybrid combinations of approaches 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 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. Since the RT-ARM described in Sections 2 and 3.3 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 in Section 1, we describe four types of performance optimizations for this scenario, illustrated in Figure 5:

A. Dynamic scheduling: If we cannot feasibly schedule all of the RT-ARM operations with the mandatory operations, or the combination produces a barely feasible schedule and we lack confidence in the precision of the advertised execution

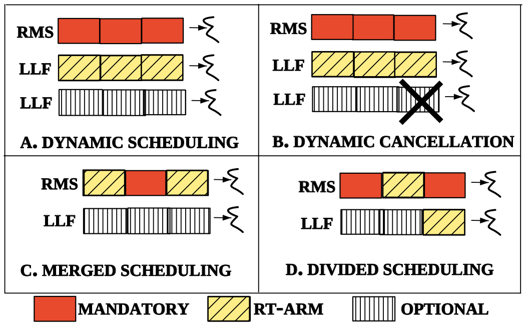

Figure 5: Steady State Optimizations

times, we might trade some measure of overhead for stricter partitioning between the mandatory and RT-ARM operations, and schedule the RT-ARM operations in an intermediate priority queue using a deadline- or laxity-based discipline. This optimization allows the target system some 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 RT-ARM or optional operations.

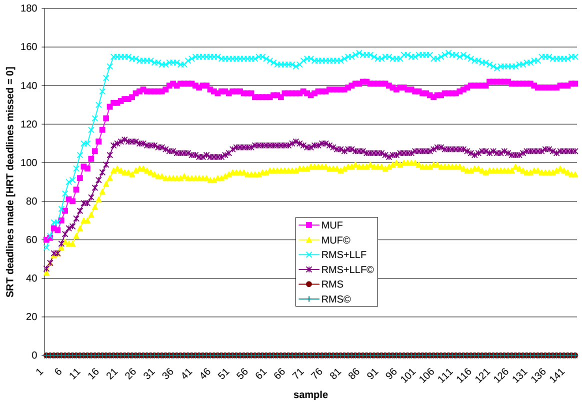

Figure 6: Effects of Pessimistic Cancellation

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 shown in Figure 6, our initial measurements of this technique using a rather pessimistic cancellation strategy actually *reduced* the number of optional operations that made their deadlines. 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 RT-ARM operations and mandatory operations together using RMS, then merging the RT-ARM 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 RT-ARM 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 RT-ARM operations themselves into mandatory and optional segments (*e.g.*, to consider different ranges of available rates) and the RT-ARM 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 RT-ARM.

By ensuring that the critical status assessment portion of the RT-ARM is feasibly scheduled, and thus avoiding consistency recovery costs, this optimization can help meet our goal of performing adaptive resource reallocations within firmly bounded time-scales. 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.

3.3 Adaptive Optimizations

In our prior adaptive scheduling research [11], a previous-generation real-time adaptive resource manager (RT-ARM) [10] interacted with a previous-generation instance of our scheduler via its *sensitivity interface*. This interface allowed the RT-ARM 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 RT-ARM performed these steps whenever a transition between steady states was needed.

To perform its assignment algorithm, the RT-ARM 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 RT-ARM and the scheduler sensitivity implementation were reasonable, as shown in Figures 8 and 9 in Section 4. 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 RT-ARM 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 7:

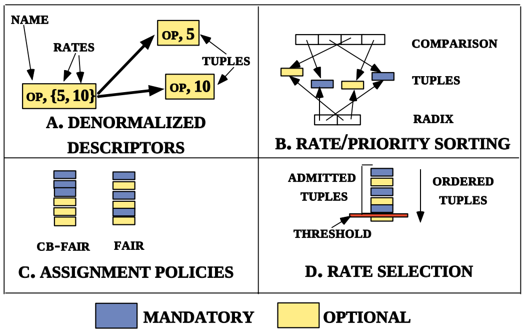

Figure 7: Adaptive Optimizations

A. De-normalized operation descriptors: We denormalize 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 mandatory, its available periods) are small, so the more efficient $O(n)$ radix sorts are applicable in many cases.

This optimization can help meet our system 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 $C_0 + 64C_1 + 4096C_2$. If we instead applied a comparison sorting strategy for combined rate and priority assignment, we would expect a tighter bound of $C_3 + 384C_4$. Finally, if we instead applied radix sorting for combined rate and priority assignment, we would expect a still tighter bound of $C_5 + C_6n = C_5 + 64C_6$.

As we discuss in Section 4.2, the constant overheads for the sensitivity, comparison sorting, and radix sorting approaches are expected to be similar. 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.

C. Assignment policies: We encapsulate specific sort ordering strategies as policies for rate assignment, much as we have done previously for scheduling policies [14]. We present two canonical strategies for rate selection, based on two different views of fairness:

- **FAIR Strategy:** In the first strategy, called *Fair Assignment by Indexed Rate* (FAIR), we emphasize fairness across all operations, ordering tuples by ascending rate index, then descending criticality, then mean rate, and finally by descriptor handle. This strategy selects the lowest rate for each operation, for mandatory first operations and then optional operations, then the next rate for each mandatory operation and the each optional operation, and so forth.

- **CB-FAIR Strategy:** In the second strategy, called *Criticality-Biased FAIR* (CB-FAIR), we emphasize criticality partitioning, and order tuples first by descending criticality, then by ascending rate index index, then mean rate, and finally descriptor handle. This optimization adds flexibility to meet our goal to improve real-time performance across heterogeneous criteria, *i.e.*, rate and criticality.

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 mandatory 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. This optimization can help meet our goals to trade performance of individual elements for overall real-time objectives, and to perform adaptive resource reallocations within firmly bounded time-scales.

4 Empirical Studies

4.1 Motivation for Empirical Studies

This section presents an empirical study consisting of the following experiments that quantify the effects of our optimizations and their overall benefit to the avionics mission computing platform we are targeting: (1) measure overhead factors for the original sensitivity-based adaptive rescheduling approach described in Section 3.3 and (2) quantify and demonstrate the trade-offs when choosing between two alternative scheduling strategies. The results of this study have the following benefits for designers and developers of DRE systems:

Original rescheduling profile: Characterizing the order and constant factors of the original sensitivity-based adaptive rescheduling approach gave insights leading to the comparison sorting and radix sorting optimizations. Furthermore, the constant factors for the original approach can be related to the constant factors for the optimized approaches, so that we can project the overhead effects of those optimizations based on the order reductions they provide.

Characteristics of primitive elements: Characterizing the overhead of primitive dispatching elements gives guidance on expected trade-offs between different adaptive rescheduling strategies, based on the configuration of dispatching elements each produces. These empirical results, along with analytical approaches such as exact [18] or polynomial-time [19] schedulability tests, serve to guide selection of scheduling strategies, based on performance and schedulability trade-offs.

Comparison of concrete alternatives: Examining two alternative scheduling strategies in detail reveals nuances of the expected trade-offs in a realistic environment for our target platform. We juxtapose these results with the appropriate schedulability considerations to show how the tension between empirical and analytical results guides optimization within the total space of scheduling heuristics.

Space of alternative strategies: Empirical measurements of a larger space of alternative scheduling strategies will (1) identify trade-offs for off-line selection of sets of appropriate scheduling strategies, (2) characterize the overhead factors across the space of relevant scheduling strategies, and (3) ultimately provide parameters for on-line adaptive selection of scheduling strategies to optimize both the effectiveness of scheduling for any steady application state and the time of adaptive transitions between steady states.

Concrete rescheduling characteristics: Quantifying the original sensitivity-based adaptive rescheduling approach, and the comparison sorting and radix sorting optimizations in the target platform environment will give hard data on the effectiveness of these optimizations for production avionics systems. Furthermore, precise measurement of the constant overhead factors for these approaches will allow reliable projection of the effects of different numbers of operations and the degree of information about their scheduling characteristics on actual scheduling behavior in those systems.

The first experiments were conducted on a single CPU (300MHz Pentium) machine, running the Windows NT Workstation 4.0 operating system. The second set of experiments was conducted on single-CPU (200MHz Power PC) single-board computers, running the VxWorks 5.3 operating system.

4.2 Empirical Results

In this section, we examine the results of the three completed sets of experiments in our empirical study. Section 4.2.1 profiles the original sensitivity-based approach to adaptive rescheduling of operations. Section 4.2.2 explores concrete alternative scheduling strategies in the target platform environment, describes their relative performance, and relates those results to the earlier more basic measurements.

4.2.1 Original Rescheduling Profile

Qualitative comparisons of the adaptive transition optimizations we propose for integrating predictable RT-ARM execution with mandatory and optional tasks were discussed in Section 3.3. We support these indications of the benefits of our approach with new results measuring the behavior of the previous generation RT-ARM and scheduler during adaptive transitions over a small number of operations.

Figures 8 and 9 show respectively the number and average duration of calls to our scheduler by the earlier-generation RT-ARM during an adaptive transition. In each figure, we plot data for the `operation_set_utilization_value` call, which returns the utilization level for a particular assignment of rates to tasks by the RT-ARM, and for the `operation_sensitivity` call, which returns a value indicating the sensitivity of the current utilization level to changes in the proposed rates.

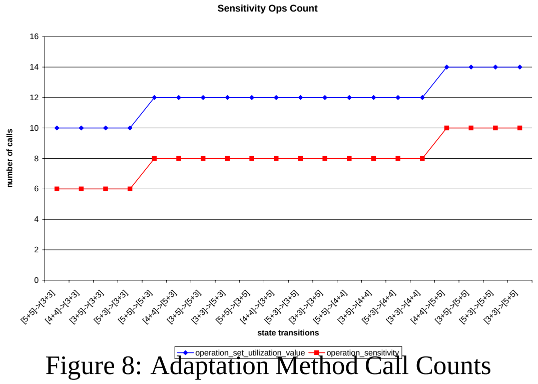

Figure 8: Adaptation Method Call Counts

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

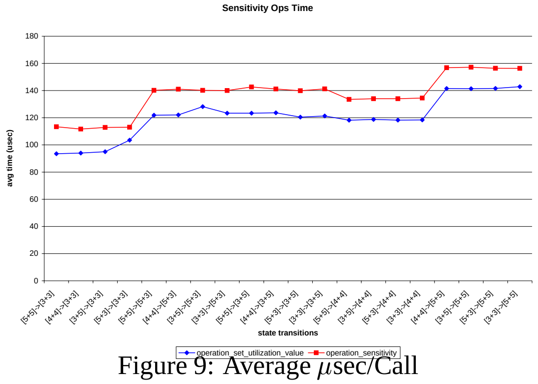

Figure 9: Average μsec/Call

These results motivate the comparison sort and radix sort optimizations to adaptive rescheduling described in Section 3.3. In particular, we expect that constants $C_0, \ldots, C_6$ in the order complexity equations 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.

4.2.2 Comparison of Concrete Alternatives

As shown in Figure 10, there was a measurable difference in the dispatching performance of three alternative scheduling strategies, RMS [15], MUF [16], and RMS+LLF [9] in the target platform environment. In each subsequent operating state we increased the non-critical load, and also cycled from state to state through low, medium-low, high, and medium high levels of execution time jitter. RMS performed optimally in these studies up to the point of overload, shown in state 7. While both MUF and RMS+LLF allowed critical operations to meet their deadlines in overload[2], RMS+LLF was consistently able to achieve more deadline successes in states with low or high jitter levels, while MUF made more deadlines in states with intermediate jitter levels.

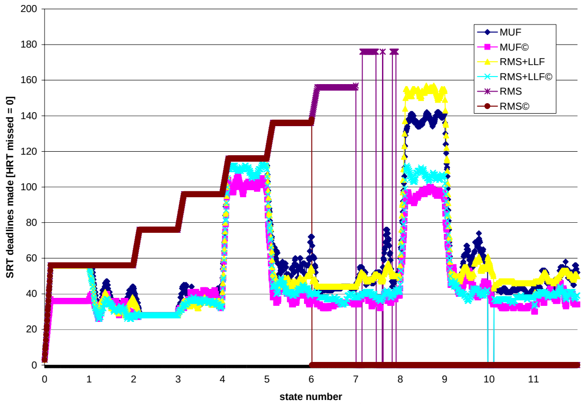

Figure 10: Comparison of Strategies

We attribute these differences to two factors. First, RMS+LLF applies a static queueing discipline for all critical operations and fans them out across multiple rate-based queues, both of which tend to reduce the overhead per operation at a given level of loading. MUF on the other hand manages all critical operations within a single queue ordered by laxity. Second is the ability to manage queue ordering and preemption effectively, and the known sensitivity of RMS to non-harmonic rates of execution requests. As the jitter in execution times may be considered a kind of additional sporadic load, it seems reasonable that at some levels of jitter a kind of non-harmonic behavior would result, *e.g.*, as in the intermediate levels observed here.

[2] a single critical deadline miss was observed late in state 9, for MUF without cancellation and for RMS+LLF with cancellation.

4.3 Interpretation of Empirical Results

The empirical results presented here support the following conclusions: (1) the original sensitivity-based approach leaves ample room for improvement, and the comparison sorting and radix sorting optimizations show promise for significant reductions in overhead and thus completion time for adaptive rescheduling of operations; (2) empirical measurement of overhead for primitive dispatching elements offers some guidance selection of scheduling strategies, and these results are born out in practice by the measured performance of MUF and RMS+LLF in the target platform environment.

5 Related Work

Traditional approaches to QoS enforcement have adopted existing solutions from the domain of real-time scheduling [15, 9, 17], fair queuing in network routers [20], or OS support for continuous media applications [21]. In addition, there have been efforts to implement new concurrency mechanisms for real-time processing, such as the real-time threads of Mach [22] and real-time CPU scheduling priorities of Solaris [23].

In contrast to research on network- and OS-level QoS, the programming model for developers of OO middleware focuses on invoking remote operations on distributed objects. Determining how to map the results from the network and OS layers to OO middleware is a major focus of our research. Our previous research has examined many dimensions of DRE middleware, including static [24] and dynamic [14] scheduling and real-time event services [7]. This earlier work provides the basis for our research on optimizing a flexible middleware scheduling framework described in this paper.

Feng, *et al.* [25] compare and contrast previous-generation CORBA scheduling approaches and offered suggestions for producing more open and scalable real-time CORBA middleware. Our approach follows and expands on several of their suggestions, notably offering flexible policies and mechanisms for configuring a variety of scheduling approaches, while preserving isolation of the application from low-level scheduling details.

Montez, *et al.* [26] present an approach based on hybridizing polymorphic invocation and (m,k)-firm scheduling assurances. This approach could prove beneficial for RT-ARM [10] scheduling in particular, and we plan to investigate this approach for implementation in our framework.

Standard COTS middleware approaches, such as the approved Real-Time CORBA 1.0 [4] specification, and emerging approaches, such as Dynamic Scheduling Real-Time CORBA [5] (DSRT CORBA) and the RTSJ [6], 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 DSRT CORBA under the C++ language binding. We intend to re-host Kokyu on a range of RTSJ-compliant environments, which would enable its use in implementing both the RTSJ schedulers and DSRT CORBA schedulers under the Java language binding.

6 Concluding Remarks

This paper presented a number of middleware-specific optimizations for a target application, using a flexible middleware scheduling framework. We describe a performance-oriented approach to designing and optimizing scheduling policies, and show qualitative and preliminary quantitative evidence of our approach's benefits. We believe these techniques are useful and appropriate for building mission-critical distributed real-time and embedded (DRE) applications using standards-based COTS middleware.

Lessons learned during the Kokyu research project include the following: (1) empirical results serve to validate an adaptive and hybrid scheduling approach; (2) quantifying the costs/benefits of discrete alternatives can be powerful when combined with feasibility analysis; (3) composable dispatching modules based on primitive elements enables domain-specific and even environment-specific optimizations; (4) design decisions are aided by empirical data; (5) experiments currently underway are needed to offer a quantitative blueprint for co-scheduling middleware services such as the RT-ARM with applications; (6) these experiments will allow us to demonstrate a general co-scheduling technique where feasibility analysis and empirical studies meet.

The optimizations and framework extensions described in this paper have been integrated first into the TAO Event Service [7], and the *Kokyu* source code will be available as a distinct framework provided with the ACE [27] and TAO distributions. Our continuing work is focusing on (1) a more thorough analysis of the space of scheduling heuristics enabled by this approach, combining *a priori* observations and empirical measurements to offer specific patterns and overall design guidance to developers of DRE systems and (2) further work on measuring and optimizing real-time interactions with other higher-level resource managers and schedulers in adaptive DRE middleware.

7 Acknowledgments

This work was funded in part by Boeing. We gratefully acknowledge the support and direction of Boeing Principal Investigators David Corman and David Sharp. In addition, we would like to thank Boeing Engineers Brian Mendel and Jeanna Gossett for their contributions to this research.

References

[1] D. L. Levine, C. D. Gill, and D. C. Schmidt, "Dynamic Scheduling Strategies for Avionics Mission Computing," in *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Nov. 1998.

[2] E. R. 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.

[3] H. Hansson and H. Lawson and O. Bridal and C. Eriksson and S. Larsson and H. Lon and M. Stromberg, "BASEMENT: An Architecture and Methodology for Distributed Automotive Real-Time Systems," *IEEE Transactions on Computers*, vol. 46, pp. 1016–1027, SEPTEMBER 1997.

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

[5] Object Management Group, *Dynamic Scheduling Real-Time CORBA Joint Revised Submission*, OMG Document orbos/2000-08-12 ed., August 2000.

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

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

[8] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

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

[10] 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), IEEE, 1997.

[11] B. S. Doerr, T. Venturella, R. Jha, C. D. Gill, and D. C. Schmidt, "Adaptive Scheduling for Real-time, Embedded Information Systems," in *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Oct. 1999.

[12] Center for Distributed Object Computing, "TAO: A High-performance, Real-time Object Request Broker (ORB)." www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

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

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

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

[16] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Optimal Reward-Based Scheduling for Periodic Real-Time Tasks," *IEEE Transactions on Computers*, vol. 50, pp. 111–129, February 2001.

[17] 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*, pp. 166–171, IEEE Computer Society Press, 1989.

[18] C.-C. J. Han and H. Y. Tyan, "A Better Polynomial-Time Schedulability Test for Real-Time Fixed-Priority Scheduling Algorithms," in *IEEE Real-Time Systems Symposium*, (San Francisco, CA), IEEE, Dec. 1997.

[19] L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switched Networks," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 19–29, ACM, Sept. 1990.

[20] G. Coulson, G. Blair, J.-B. Stefani, F. Horn, and L. Hazard, "Supporting the Real-time Requirements of Continuous Media in Open Distributed Processing," *Computer Networks and ISDN Systems*, pp. 1231–1246, 1995.

[21] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards Predictable Real-time Systems," in *USENIX Mach Workshop*, USENIX, October 1990.

[22] Khanna, S., *et al.*, "Realtime Scheduling in SunOS 5.0," in *Proceedings of the USENIX Winter Conference*, pp. 375–390, USENIX Association, 1992.

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

[24] W. Feng, U. Syyid, and J.-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), IEEE, December 1997.

[25] 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.

[26] D. C. Schmidt, "Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software," in *Handbook of Programming Languages* (P. Salus, ed.), MacMillan Computer Publishing, 1997.