The Design and Performance of a Real-Time CORBA Scheduling Service

Christopher D. Gill, David L. Levine, and Douglas C. Schmidt
{cdgill,levine,schmidt}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA*

October 3, 1999

A subset of this paper will appear in the *International Journal of Time-Critical Computing Systems*, special issue on Real-Time Middleware, guest editor Wei Zhao, 1999.

Abstract

There is increasing demand to extend CORBA middleware to support applications with stringent quality of service (QoS) requirements. However, conventional CORBA middleware does not define standard features to dynamically schedule operations for applications that possess deterministic real-time requirements. This paper presents three contributions to the study of real-time CORBA operation scheduling strategies.

First, we document our evolution from static to dynamic scheduling for applications with deterministic real-time requirements. Second, we describe the flexible scheduling service framework in our real-time CORBA implementation, TAO, which supports core scheduling strategies efficiently. Third, we present results from simulations and empirical benchmarks that quantify the behavior of these scheduling strategies and assess the overhead of dynamic scheduling in TAO.

Our simulation results show how hybrid static/dynamic strategies that consider operation criticality are capable of preserving scheduling guarantees for critical operations under an overloaded schedule. In addition, our empirical results using TAO show that dynamic scheduling of CORBA operations can be deterministic and can achieve acceptable latency for operations, even with moderate levels of queueing.

Keywords: Middleware and APIs, Quality of Service Issues, Mission Critical/Safety Critical Systems, Dynamic Scheduling Algorithms and Analysis, Distributed Systems.

*This work was supported in part by Boeing, DARPA contract 9701516, Lucent, Motorola, Nortel, NSF grant NCR-9628218, Siemens, and Sprint.

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 deterministic real-time requirements, such as process control and avionics mission computing systems [1], impose severe constraints on the design and implementation of real-time OO middleware. For example, avionics mission computing applications typically manage sensors and operator displays, navigate the aircraft's course, and control on-board equipment. Middleware for such applications must support deterministic real-time QoS requirements.

1.2 Design and Implementation Challenges

Figure 1 illustrates the architecture of a representative real-time application – an OO avionics mission computing platform [2] – developed and deployed using OO middleware components and services based on CORBA [3]. CORBA Ob-

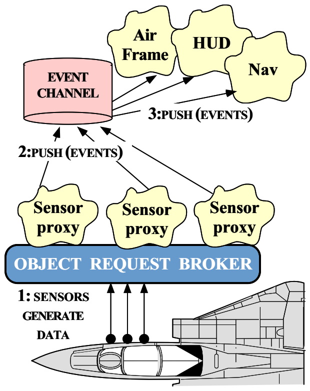

Figure 1: Example Avionics Mission Computing Application

ject Request Brokers (ORB)s allow clients to invoke operations on target object implementations without concern for where the object resides, what language the object implementations are written in, the OS/hardware platform, or the types of communication protocols, networks, and buses used to interconnect distributed objects [4]. However, achieving these benefits for deterministic real-time applications requires the resolution of the following design and implementation challenges:

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

Severe resource limitations: Many real-time applications must minimize processing due to strict resource constraints, such as cost, weight, and power consumption restrictions. A consequence of using off-line scheduling analysis is that worst-case processing requirements drive the schedule. 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 [5] may have to assume that such processing will take some maximum amount of time, when often it may take much less.

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 [6]. A large percentage of software maintenance involves testing. Current scheduling approaches are validated by extensive testing, which is tedious and non-comprehensive. Therefore, analytical assurance is essential to help reduce validation 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 Deterministic Real-Time Applications

Our experience using CORBA on telecommunication [7] and medical imaging projects [5] illustrates 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 [4].

However, conventional CORBA ORBs are not yet suited for demanding real-time applications because they do not provide features or optimizations to schedule operations that require deterministic real-time QoS [8]. To meet these requirements, we have developed a real-time CORBA Object Request Broker (ORB) called TAO [9]. TAO is an open source, implementation of standard CORBA whose ORB and services support efficient and predictable real-time, distributed object computing.

Our prior work on TAO has explored many dimensions of high-performance and real-time ORB design and performance, including event processing [2], request demultiplexing [10], I/O subsystem integration [11], concurrency and connection architectures [12], and IDL compiler stub/skeleton optimizations [13]. This paper extends our previous work on a real-time CORBA static scheduling service [9] by incorporating a *strategized scheduling service framework* into TAO. This framework allows the configuration and empirical evaluation of multiple static, dynamic, and hybrid static/dynamic scheduling strategies, such as Rate Monotonic Scheduling (RMS) [14], Earliest Deadline First (EDF) [14], Minimum Laxity First (MLF) [15], and Maximum Urgency First (MUF) [15].

To maintain scheduling guarantees and to simplify testing for demanding real-world real-time applications, we have extended our prior work on TAO incrementally. In particular, our approach focuses on applications with the following characteristics:

- *Bounded executions* – operations stay within the limits of their specified execution times.
- *Bounded rates* – dispatch requests arrive within the specified period.
- *Known operations* – all operations are known to the scheduler before run-time, or are reflected entirely within the execution times of other specified operations.

These types of applications are historically configured and scheduled *statically*, which enables TAO to minimize run-time overhead that would otherwise stem from mechanisms that enforce operation execution time limits [2] or perform dynamic admission control.

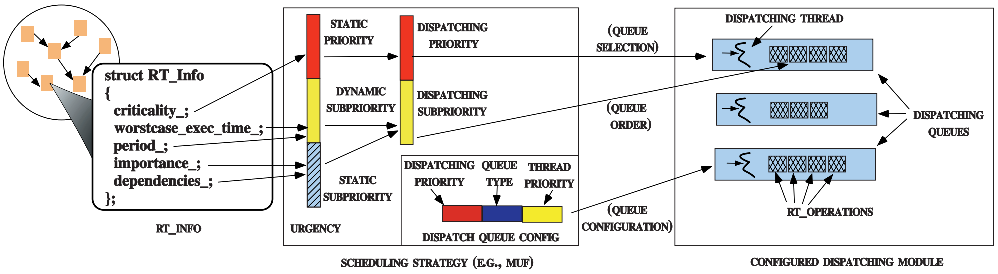

Figure 2: Relationships Between Operation, Scheduling, and Dispatching Terminology

Within these constraints, the work on TAO's strategized scheduling service framework described in this paper allows applications to specify custom static and/or dynamic scheduling strategies. This framework increases adaptability across application families and operating systems, while preserving the rigorous scheduling guarantees and testability offered by our previous work on statically scheduled CORBA operations.

1.4 Paper Organization

The remainder of this paper is organized as follows: Section 2 reviews the drawbacks of off-line, static scheduling and introduces the dynamic and hybrid static/dynamic scheduling strategies our research is evaluating. Section 3 discusses the design and implementation of TAO's scheduling service framework, which supports a range of static, dynamic, or hybrid static/dynamic real-time scheduling strategies. Section 4 demonstrates how TAO's scheduling service can be used to visualize scheduler behavior for different scheduling strategies at the critical instant. Section 5 presents results from benchmarks that evaluate the dynamic scheduling strategies empirically to compare the run-time dispatching overhead of static and dynamic scheduling strategies. Section 6 discusses related work and Section 7 presents concluding remarks.

2 Overview of Scheduling Strategies

This section describes the limitations of purely static scheduling and outlines the potential benefits of applying dynamic scheduling. In addition, we evaluate the limitations of purely dynamic scheduling strategies. This evaluation motivates the hybrid static/dynamic strategy used by TAO to schedule real-time CORBA operations, which is described in Section 3.

2.1 Synopsis of Scheduling Terminology

Precise terminology is necessary to describe and evaluate static, dynamic, and hybrid scheduling strategies. Figure 2 shows the relationships between the key terms defined below.

RT_Operation and RT_Info: In TAO, an `RT_Operation` is a scheduled CORBA operation [9]. In this paper, we use *operation* interchangeably with `RT_Operation`. An `RT_Info struct` is associated with each operation and contains its QoS parameters. Section 3.4.1 describes these concepts in more detail. The `RT_Info` structure contains the following operation characteristics shown in Figure 2 and 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. These concepts are discussed in more detail in Sections 3.4.3 and 3.4.4. The key elements of this transformation are shown in Figure 2 and defined as follows:

• **Urgency:** Urgency [15] 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 3 illustrates these relationships.

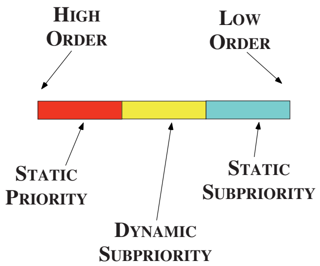

Figure 3: Relationships in the Urgency Tuple

• **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:

• **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.

TAO's 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 [16] and FIFO-r [17], 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

[1] An operation's laxity is the time until its deadline minus its remaining execution time.

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 [11], ORB Core [12], and/or the Event Service [2]. TAO's dispatching modules are discussed in Section 3.4.6.

2.2 Limitations of Static Scheduling

Many hard real-time systems, such as those for avionics mission computing and manufacturing process controllers, have traditionally been scheduled statically using RMS [18]. Static scheduling provides schedulability assurance prior to run-time and can be implemented with low run-time overhead [9]. 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[2] 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 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 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 $n(2^{1/n} - 1)$ [14] 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 [20]. 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[3] 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 [21, 22] 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.

2.3 Overcoming Static Scheduling Limitations with Dynamic Scheduling

Other scheduling algorithms provide some relief from the limitations of RMS. For instance, 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 described in Section 2.2. In particular, dynamic scheduling strategies offer optimal utilization capabilities [14] 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.

[2] A sporadic server [19] reserves a portion of the schedule to allocate to aperiodic events when they arrive.

[3] Priorities can be changed via *mode changes* [9], but that is too coarse to capture invocation-to-invocation variations in the resource requirements of complex applications.

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 paper to scheduling approaches that do not perform schedulability analysis at run-time. In particular, we do not consider strategies that require run-time admission control for dynamic scheduling. 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 maximal 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.

The remainder of this section reviews several strategies for dynamic and hybrid static/dynamic scheduling, using the terminology defined in Section 2.1. These scheduling strategies include purely dynamic techniques, such as EDF and MLF, as well as the hybrid MUF strategy.

2.3.1 Purely Dynamic Scheduling Strategies

This section briefly reviews two well known purely dynamic scheduling strategies, EDF [14, 18], and MLF [15]. These strategies are illustrated in Figure 4 and discussed below. In addition, Figure 4 depicts the hybrid static/dynamic MUF [15] scheduling strategy discussed in Section 2.3.2.

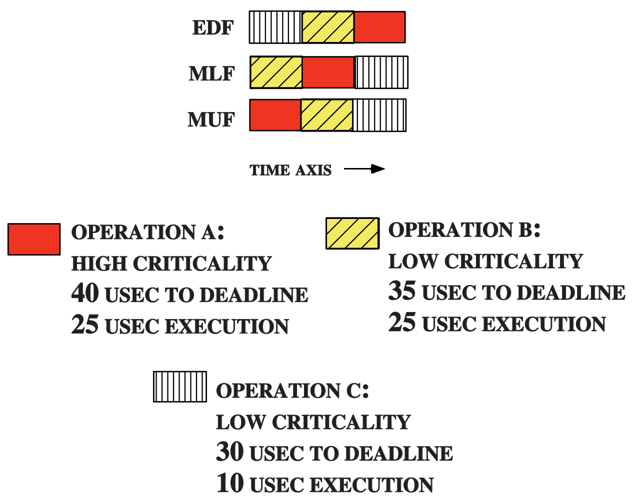

Figure 4: Dynamic Scheduling Strategies

Earliest Deadline First (EDF): EDF [14, 18] is a dynamic scheduling strategy that orders dispatches[4] of operations based on time-to-deadline, as shown in Figure 4. 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 3.4.4.

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 [15] refines the EDF strategy by taking into account operation execution time. It dispatches an operation whose *laxity* is least, as shown in Figure 4. Laxity is defined as the time-to-deadline minus the remaining execution time.

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 2.3.2.

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. In particular, the utilization phasing penalty described in Section 2.2 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 [14]. Finally, both EDF and MLF can dispatch operations within a single static priority level [14, 15], which is useful for non-preemptive single-threaded environments.

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

• **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 increases for every operation.

2.3.2 Maximum Urgency First (MUF)

The MUF [15] 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 [23]. TAO supports a variant of MUF in its strategized CORBA scheduling service framework, which is presented in Section 3. 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 described below:

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. 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 a rigid mapping 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 3.

Dynamic Subpriority: 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.[5] 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 [15]. 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.

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 used in TAO's strategized scheduling service enforces a total dispatch ordering by providing an `importance` field in the TAO `RT_Info` CORBA operation QoS descriptor [9], which is described in Section 2.1. TAO's scheduling service 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 [15] uses the terms *dynamic priority* and *user priority*, whereas we use the term *dynamic subpriority* and *static subpriority* for TAO's scheduling service, respectively. We selected different terminology to indicate the subordination to static priority. These terms are interchangeable when referring to the MUF strategy, however.

[5]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.

3 The Design of TAO's Strategized Scheduling Service

TAO's scheduling service provides real-time CORBA applications the flexibility to specify and use different scheduling strategies, according to their specific QoS requirements and available OS features. This flexibility allows CORBA applications to extend the set of available scheduling strategies *without* impacting strategies used by other applications. Moreover, it shields application developers from unnecessary details of their scheduling strategies.

This section outlines the design goals and architecture of TAO's strategized scheduling service framework. After briefly describing TAO in Section 3.1, Section 3.2 discusses the design goals of TAO's strategized scheduling service. Section 3.3 offers an overview of its architecture and operation. Finally, Section 3.4 discusses the resulting architecture in detail.

3.1 Overview of TAO

TAO is a high-performance, real-time ORB endsystem targeted for applications with deterministic and statistical QoS requirements, as well as "best-effort" requirements. The TAO ORB endsystem contains the network interface, OS, communication protocol, and CORBA-compliant middleware components and features shown in Figure 5. TAO supports the standard OMG CORBA reference model [3], with the following enhancements designed to overcome the shortcomings of conventional ORBs [12] for high-performance and real-time applications:

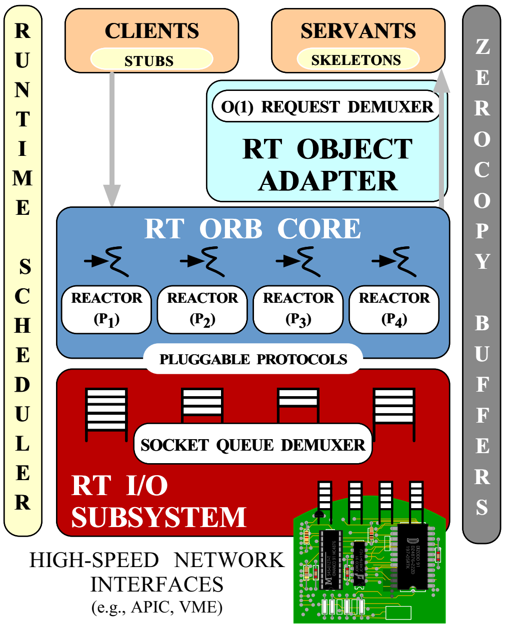

Figure 5: Components in the TAO Real-time ORB Endsystem

Optimized IDL Stubs and Skeletons: IDL stubs and skeletons perform marshaling and demarshaling of application operation parameters, respectively. TAO's IDL compiler generates stubs/skeletons that can selectively use highly optimized compiled and/or interpretive (de)marshaling [13]. This flexibility allows application developers to selectively trade off time and space, which is crucial for high-performance, real-time, and/or embedded distributed systems.

Real-time Object Adapter: An Object Adapter associates servants with the ORB and demultiplexes incoming requests to servants. TAO's real-time Object Adapter [10] uses perfect hashing [24] and active demultiplexing [10] optimizations to dispatch servant operations in constant $O(1)$ time, regardless of the number of active connections, servants, and operations defined in IDL interfaces.

Run-time Scheduler: A real-time scheduler [8] maps application QoS requirements, such as bounding end-to-end latency and meeting periodic scheduling deadlines, to ORB endsystem/network resources, such as CPU, memory, network connections, and storage devices. TAO's run-time scheduler supports both static [9] and dynamic [25] real-time scheduling strategies.

Real-time ORB Core: An ORB Core delivers client requests to the Object Adapter and returns responses (if any) to clients. TAO's real-time ORB Core [12] uses a multi-threaded, preemptive, priority-based connection and concurrency architecture [13] to provide an efficient and predictable CORBA protocol engine. TAO's ORB Core allows customized protocols to be plugged into the ORB without affecting the standard CORBA application programming model.

Real-time I/O subsystem: TAO's real-time I/O (RIO) subsystem [11] extends support for CORBA into the OS. RIO assigns priorities to real-time I/O threads so that the schedulability of application components and ORB endsystem resources can be enforced. When integrated with advanced hardware, such as the high-speed network interfaces described below, RIO can (1) perform early demultiplexing of I/O events onto prioritized kernel threads to avoid thread-based priority inversion and (2) maintain distinct priority streams to avoid packet-based priority inversion. TAO also runs efficiently and relatively predictably on conventional I/O subsystems that lack advanced QoS features.

High-speed network interface: At the core of TAO's I/O subsystem is a "daisy-chained" network interface consisting of one or more ATM Port Interconnect Controller (APIC) chips [26]. The APIC is designed to sustain an aggregate bi-directional data rate of 2.4 Gbps using zero-copy buffering optimization to avoid data copying across endsystem layers. In addition, TAO runs on conventional real-time interconnects, such as VME backplanes and multi-processor shared memory environments, as well as Internet protocols like TCP/IP.

TAO is developed using lower-level middleware called ACE [27], which implements core concurrency and distribution patterns [28] for communication software. ACE provides reusable C++ wrapper facades and framework components that support the QoS requirements of high-performance, real-time applications and higher-level middleware like TAO. ACE and TAO run on a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems like Sun/Chorus ClassiX, LynxOS, and VxWorks.

3.2 Design Goals of TAO's Scheduling Service

To alleviate the limitations with existing scheduling strategies described in Section 2, 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 (when the set of critical operations can be identified), and (3) adapt flexibly to different application and platform characteristics. These goals are illustrated in Figure 6 and summarized below:

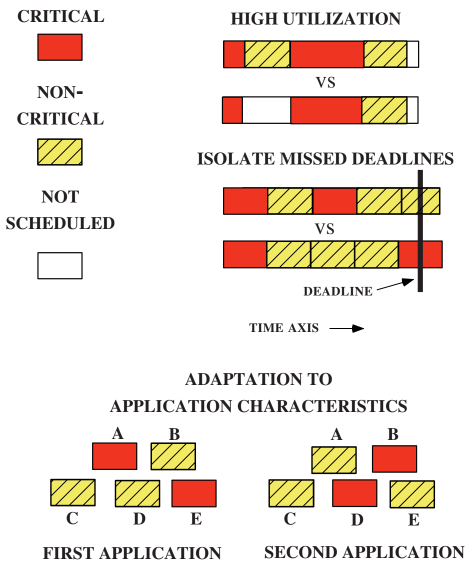

Figure 6: Design Goals of TAO's Dynamic Scheduling Service

Goal 1 – Higher utilization: The upper pair of timelines in Figure 6 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.

Goal 2 – Preserving scheduling guarantees: The lower pair of timelines in Figure 6 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 at the bottom of Figure 6 demonstrate our third research goal: *providing applications with the flexibility to adapt to varying application requirements and platform features*. In this example, the first and second applications use the same five operations. However, the first application considers operations A and E critical, whereas the second application considers operations B and D critical. By allowing applications 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 TAO's strategized scheduling service framework, described in Section 3.3. For the real-time systems [2, 9, 11, 12, 29] 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 TAO's scheduling service 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.

3.3 TAO's Strategized Scheduling Service Framework

TAO's scheduling service framework is designed to support a variety of scheduling strategies, including RMS, EDF, MLF, and MUF. In addition, this framework provides a common environment to systematically compare both existing and new scheduling strategies. This flexibility is achieved in TAO via the *Strategy* pattern [28], which allows parts of the sequence

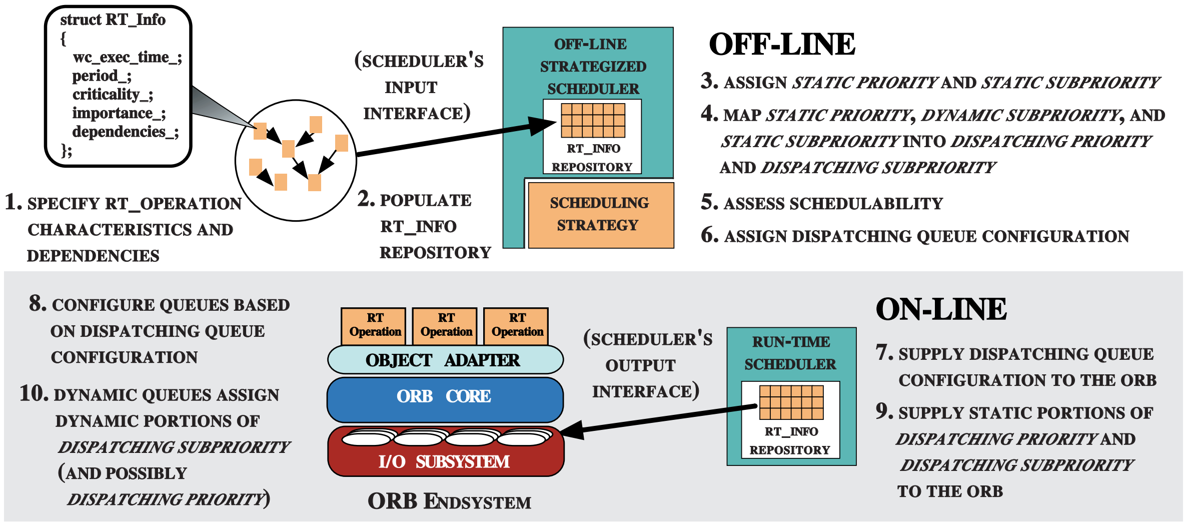

Figure 7: Processing Steps in TAO's Dynamic Scheduling Service Architecture

of steps in an algorithm to be replaced, thus providing interchangeable variations within a consistent algorithmic framework. TAO's scheduling service 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.

The architecture and behavior of TAO's strategized scheduling service is illustrated in Figure 7. This architecture evolved from our earlier work on a CORBA scheduling service [9] that supported purely static RMS for avionics mission computing applications [2, 1, 29]. Based on this work, as well as our experience prototyping dynamic scheduling strategies, we have identified the following set of common steps 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 TAO's scheduling service, which is implemented as a CORBA object, *i.e.*, it implements an IDL interface. The use of CORBA IDL allows applications to specify a sets of values (*i.e.*, `RT_Infos`) that concisely capture the characteristics of each of its schedulable operations (*i.e.*, `RT_Operations`), 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 TAO's scheduling service via its *input interface*, which is described in Section 3.4.1. TAO's scheduling service stores the QoS information in its repository of `RT_Info` descriptors. TAO's scheduling service constructs operation dependency graphs based on `RT_Infos` registered with it by the application. The scheduling service then identifies threads of execution by examining the terminal nodes of these dependency graphs. 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.

Step 3: Next, TAO's scheduling service assigns static priorities and subpriorities to operations. These values are assigned according to the specific strategy used to configure the scheduling service. For example, when the TAO scheduling service 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, TAO's strategized scheduling service can minimize overhead and non-determinism at run-time.

Step 4: Based on the specific strategy used to configure it, TAO's scheduling service 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, TAO's scheduling service 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* [14]. 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 off-line scheduling service, the number and types of dispatching queues needed to dispatch the generated schedule are assigned. For example, when the scheduling service is configured with the MLF strategy, there is a single queue, which uses laxity-based prioritization. As in Step 3, this static information is cached in the `RT_Info` repository until it is needed at run-time.

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 scheduling service's run-time scheduler component, which is collocated within an ORB endsystem. The ORB uses this run-time scheduler to retrieve (1) the thread priority at which each queue dispatches operations and (2) the type of dispatching prioritization used by each queue. The scheduling service's run-time component provides this information to the ORB via its *output interface*, as described in Section 3.4.2. By encapsulating the thread priority and dispatching type information behind its output interface, TAO's strategized scheduling service decouples the *policies* for dispatching behavior from the *mechanisms* used to enforce those policies.

Step 8: In this step, the ORB configures its *dispatching modules*, *i.e.*, in its I/O subsystem, ORB Core, and/or Event Service, as described in Section 3.4.6. Dispatching modules use the information from the scheduling service's output interface to create the correct number and types of queues and associate them with the correct thread 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. To accomplish this, the dispatching module queries TAO's scheduling service's output interface, as described in Section 3.4.2. The run-time scheduler component of TAO's scheduling service (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 scheduling service 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. Steps 1-6 typically occur off-line during the schedule configuration process, while steps 7-10 typically occur on-line. Support for these static and dynamic steps underscore the hybrid nature of TAO's scheduling architecture.

The remainder of this section describes TAO's strategized scheduling service framework in detail. Section 3.4 motivates why TAO allows applications to vary their scheduling strategy and shows how TAO's framework design achieves this flexibility.

3.4 Enhancing Flexibility in TAO's Strategized Scheduling Framework

The QoS requirements of applications and the hardware/software features of platforms and networks on which they are hosted often vary significantly. For instance, a scheduling strategy that is ideal for telecommunication [7] or medical imaging [5] applications may be poorly suited for avionics mission computing [2]. Therefore, TAO's scheduling service framework allows applications to vary their scheduling strategies selectively. The flexibility of TAO's strategized scheduling service architecture is motivated by the following two design goals:

1. *Shield application developers from unnecessary implementation details of alternative scheduling strategies* – This improves the system's reliability and maintainability.
2. *Decouple the strategy for priority assignment from the dispatching model so the two can be varied independently* – This increases the system's flexibility to adapt to varying application requirements and platform features.

TAO supports this flexibility by decoupling the *fixed* portion of its scheduling framework from the *variable* portion, as follows:

Fixed CORBA IDL interfaces: The fixed portion of TAO's strategized scheduling service framework is defined by the following two CORBA IDL interfaces:

• **Input Interface:** As discussed in Section 3.4.1, the input interface consists of the three operations shown in Figure 9. Applications use these operations to manipulate QoS characteristics expressed with TAO's `RT_Info` descriptors [9] (steps 1 and 2 of Figure 7).

• **Output interface:** As discussed in Section 3.4.2, the output interface consists of the two operations shown in Figure 10. One operation returns the dispatching module configuration information (step 7 of Figure 7). The other returns the dispatching priority and dispatching subpriority components assigned to an operation (step 9 of Figure 7). Section 3.4.6 describes how TAO's dispatching modules use information from TAO's scheduling service's output interface to configure and manage dispatching queues, as well as dispatch operations according to the generated schedule.

Variable mappings: By decoupling (1) the strategy for urgency assignment from (2) the assignment of urgency to dispatching priority and dispatching subpriority, TAO allows the scheduling strategy and the underlying dispatching model to vary independently. In addition, this decoupling enables the systematic comparison of scheduling strategies over a range of dispatching models, from fully preemptive-by-urgency, through preemptive-by-priority-band, to entirely non-preemptive, which are discussed in Section 3.4.5. As illustrated in figure 8, The variable portion of TAO's schedul-

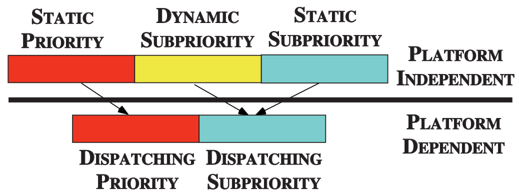

Figure 8: Distinct Mapping Levels Within Each Strategy

ing service framework is implemented by the following two distinct levels of mapping from input interface to output interface:

• **Input mapping:** The input mapping assigns *platform-independent* urgency components, *i.e.*, static priority, dynamic subpriority, and static subpriority, based on (1) the operation characteristics specified to the input interface and (2) the selected scheduling strategy, *e.g.*, MUF, MLF, EDF, or RMS. Section 3.4.3 describes how each of the strategies implemented in TAO maps from the input interface to urgency values.

• **Output mapping:** The output mapping assigns dispatching priority and dispatching subpriority according to based on the urgency components assigned in the first level, and the underlying dispatching model. Section 3.4.4 describes how the output mapping translates the assigned urgency values into the appropriate dispatching priority and dispatching subpriority values for the output interface. Section 3.4.5 describes alternatives to the output mapping used in TAO and discusses key design issues related to these alternatives.

The remainder of this section describes how TAO's strategized scheduling service framework supports these fixed CORBA IDL interfaces and variable mappings. For each aspect of TAO's scheduling framework, we outline the key design challenges and explain how our solution addresses these challenges.

3.4.1 TAO's Scheduling Service 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 3.4.1. 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 [15]. 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* CORBA 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 7, 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 the CORBA IDL interface operations shown in Figure 9 and described below:

```
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 9: TAO Scheduling Service Input IDL Interface

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 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.

3.4.2 TAO's Scheduling Service 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 TAO's strategized 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: TAO's scheduling service 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 7, the ORB uses TAO's scheduling service output interface to obtain QoS enforcement information and configure its dispatching modules accordingly. The output interface for TAO's scheduling service consists of the CORBA IDL interface operations shown in Figure 10 and described below:

dispatch_configuration(): This operation provides configuration information for queues in the dispatching modules used by the ORB endsystem (step 7 of Figure 7). 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. The run-time component of TAO's scheduling service retrieves these values from the `RT_Info` repository, where they were stored by TAO's off-line scheduling component (step 6 of Figure 7). 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 7). It takes an `RT_Info` descriptor handle as

```
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 10: TAO Scheduling Service Output IDL Interface

an input parameter and returns the assigned dispatching subpriority and dispatching priority as output parameters. The run-time component of TAO's scheduling service retrieves the dispatching priority and dispatching subpriority values stored in the `RT_Info` repository by its off-line component (step 4 of Figure 7). 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.

3.4.3 Input Mappings Implemented in TAO's Scheduling Service

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 TAO's strategized 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: TAO decouples priority from the specific priority enforcement capabilities of the underlying OS platform by providing a separate, platform-independent, level of priority assignment. TAO's strategized scheduling service 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, TAO's strategized scheduling service 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 TAO's strategized scheduling service, 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 TAO'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: TAO provides a platform-independent mapping from operation characteristics onto urgency for MUF as shown in Figure 11. Static priority is assigned according to criticality in

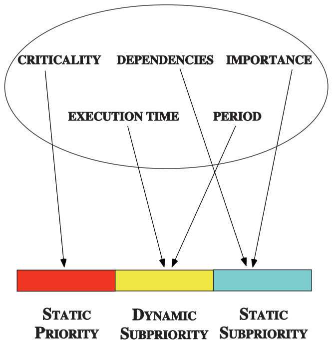

Figure 11: MUF Input Mapping

this mapping. 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: TAO provides a platform-independent mapping from operation characteristics onto urgency for MLF as shown in Figure 12. 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

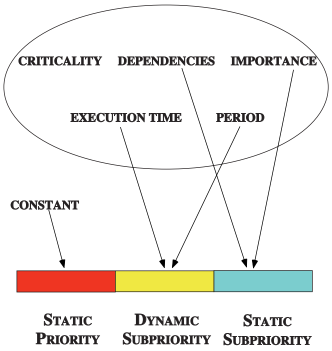

Figure 12: MLF Input Mapping

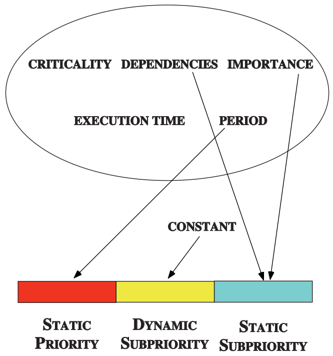

Figure 14: RMS Input Mapping

priority. The MLF strategy assigns the dynamic subpriority of each operation according to its laxity.

Defining a platform-independent mapping for EDF: TAO provides a platform-independent mapping from operation characteristics onto urgency for EDF as shown in Figure 13. Like the MLF mapping, the EDF mapping also assigns a zero value to the static priority of each operation. Moreover, the

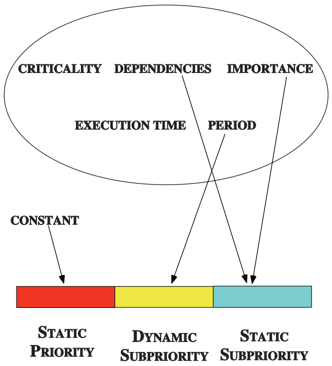

Figure 13: EDF Input Mapping

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: TAO provides a platform-independent mapping from operation characteristics onto urgency for RMS as shown in Figure 14. 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.

3.4.4 Output Mapping Implemented in TAO's Scheduling Service

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 TAO's strategized scheduling framework resolves them, are as follows:

Enforcing priority through platform-specific dispatching: The input mappings described in Section 3.4.3 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 TAO's dispatching modules: In each of TAO'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 15 illustrates the output mapping used by the scheduling strategies implemented in TAO. Each part of the mapping is described below.

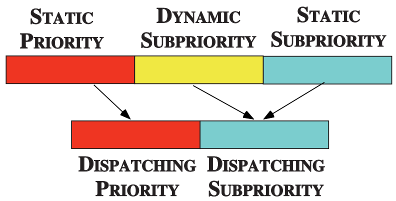

Figure 15: Output Mapping Implemented in TAO

- **Dispatching priority:** In this mapping, static priority maps directly to dispatching priority. This mapping corresponds to the priority band dispatching model described in

Section 3.4.2. Each unique static priority assigned by the input mapping results in a distinct thread priority in TAO's ORB request dispatching modules, which are described in Section 3.4.6. 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. TAO's strategized scheduling service performs this mapping efficiently at run-time by transforming both dynamic and static subpriorities into a binary representation. TAO'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.

3.4.5 Alternative Output Mappings

The scheduling strategies implemented in TAO 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 3.4.4 and to evaluate the uses and implications of alternative output mappings. Key design issues, and how TAO's strategized scheduling framework addresses them, are as follows:

Adapting to alternative OS dispatching models: TAO's strategized 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. 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 TAO's scheduling service. Below, we examine three canonical variations supported by TAO, which are illustrated in Figure 16 and described below:

• **Preemptive-by-urgency:** The output mapping currently implemented in TAO 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 [30]. This model incurs greater complexity in the dispatching module implementation, including locking during enqueue and dequeue operations, which in turn increases run-time overhead.

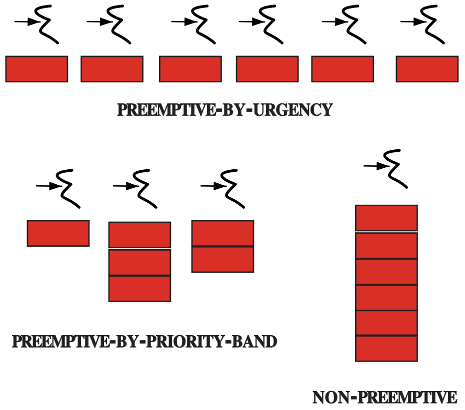

Figure 16: Dispatching Models supported by TAO

• **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 [31, 11]. 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 [32], 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 TAO's strategized scheduling service use a form of this model, as described in Section 3.4.4.

• **Non-preemptive:** This model uses a single priority queue and is non-preemptive. Unlike the previous models, this model can be used on platforms that lack preemptive multi-threading.

3.4.6 Integrating TAO's Scheduling Service with its Dispatching Modules

Applications can benefit from strategized scheduling at a variety of points along an end-to-end request-response path. TAO's dispatching modules can be integrated at a number of different points within TAO's 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 TAO's dispatching modules, and (4) discusses the issue of run-time control over dispatching priority within these dispatching modules.

Enforcing end-to-end QoS requirements: As noted in Section 2.1, one of our key research challenges is to implement dispatching modules that can enforce end-to-end QoS requirements. 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 TAO's 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 [2] use the scheduler output interface, as does its I/O subsystem [11]. Figure 17(A) illustrates a configuration where a dispatching module resides in TAO's real-time Event Service [2]. In Figure 17(A), the client application **(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.

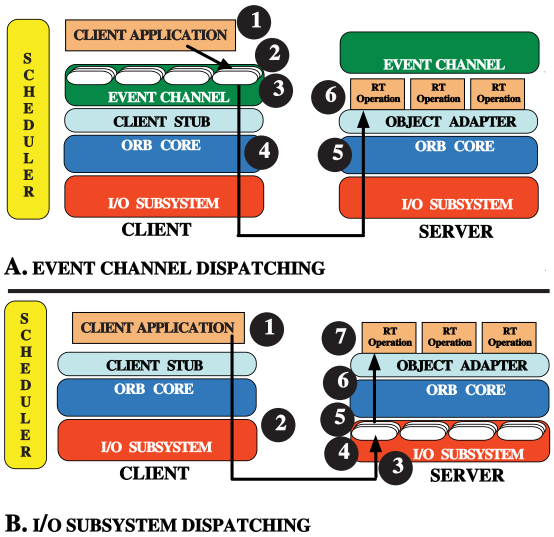

Figure 17: Alternative Placement of Dispatching Modules

Figure 17(B) illustrates an alternative configuration where a dispatching module resides in TAO's I/O subsystem [11]. 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 17 illustrate two alternatives for configuring a dispatching module within a TAO ORB endsystem. However, TAO 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 helps avoid priority inversions on server ORB endsystems via *early demultiplexing* [11] 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 TAO's dispatching modules: Figure 18 illustrates the general queueing mechanism used by the dispatching modules in TAO's ORB endsystem. In addition, this figure shows how the output information provided by TAO's scheduling service is used to configure and operate a dispatching module.

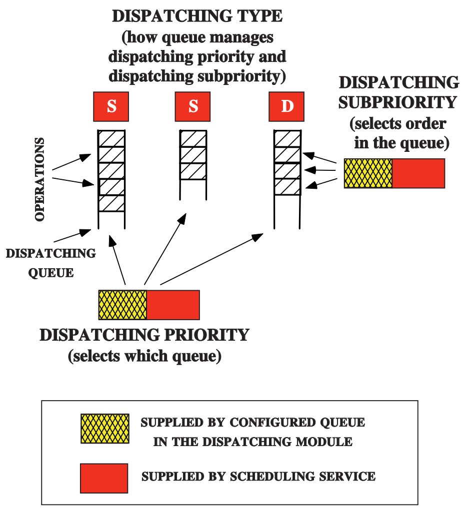

Figure 18: Example Queueing Mechanism in a TAO Dispatching Module

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, as described in Section 3.4.2. 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 [2]. 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- and laxity-based queues update operation dispatching subpriorities whenever an operation is enqueued or dequeued.

4 Simulating TAO's Critical Instant Behavior

As described in Section 3.2, two of our research goals are (1) to increase effective CPU utilization while (2) preserving scheduling guarantees for critical operations. This section presents the results of a simulation that visualizes the behavior of TAO's scheduling service under overload conditions. Our focus is on the *critical instant*, which occurs in a preemptive schedule when all operation requests arrive simultaneously [14].

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.[6] 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 different scheduling strategies in terms of operation latency, laxity, and missed deadlines, and (3) presents conclusions supported by the simulation results. The simulation results indicate the feasibility of achieving our research goals and motivate our empirical benchmarks described in Section 5.

4.1 Simulation Design

We instrumented TAO's scheduling service framework to generate timelines for the dispatching and preemption order of the operations after the critical instant. To measure 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.

To present a fair comparison of TAO's supported scheduling strategies, *i.e.*, MUF, MLF, EDF, and RMS, our simulation employs a *preemptive-by-urgency* dispatching model, as discussed in Section 3.4.5. This model always executes the highest priority operation that is ready to execute at a given time, preempting any lower priority operation when a higher priority 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 [14] specifies that it is dispatched in a fully preemptive manner.

In our simulation, 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. Table 1 summarizes the characteristics of each operation in the simulation.

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

Table 1: **Characteristics of Simulated Operations**

Each scheduling strategy emphasizes different static and

[6] This is called a *critical instant*, which is the worst case request pattern [14].

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.

4.2 Comparing Operation Latency in the Scheduling Strategies

Figure 19 depicts the average latency values for the operations using each of the scheduling strategies in the simulation. Only

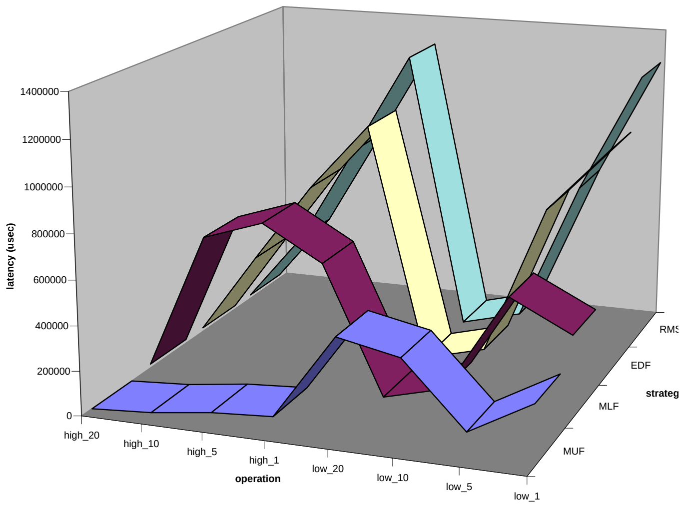

Figure 19: Latency of Operations for each Strategy

the MUF strategy minimizes the latency of critical operations, as shown in the left half of the figure. In addition, MUF detects which operations will fail to meet their deadlines. This results in an overall decrease in both latency and laxity of operations that can meet their deadlines in an overloaded system.

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

Upward spikes in the latency graph in Figure 19 show which operations incur high average latency under each strategy. Where MLF, EDF, and RMS show latency spikes for both critical and non-critical operations, MUF shows a latency spike only in the non-critical set. Maximum average laxity is lowest for MUF and MLF, which consider both the worst-case execution time and time-to-deadline. Maximum average laxity is higher for EDF, which only considers time-to-deadline. It is higher still for RMS, which does not consider any dynamic characteristics.

4.3 Comparing Operation Laxity in the Strategies

The laxity of an operation is defined as its time-to-deadline minus its remaining execution time. Figure 20 shows the average laxity values for the operations for each scheduling strategy. As with Figure 19, only the MUF strategy protects the set of

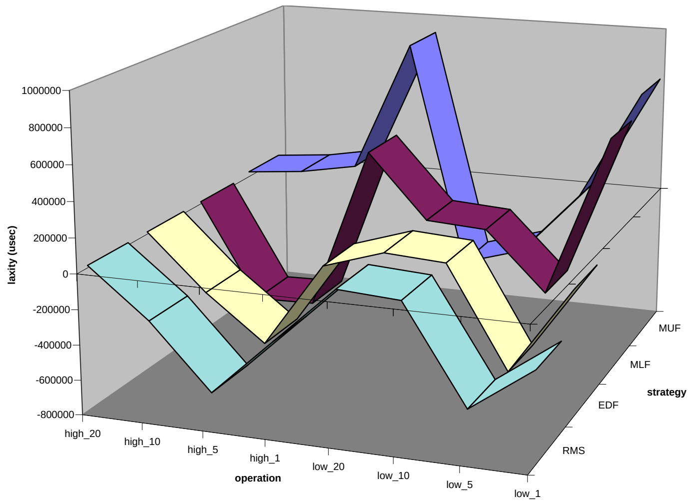

Figure 20: Laxity of Operations for each Strategy

critical operations. The other strategies have 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 positive laxity when they complete their execution have met their deadlines. Another way to visualize the operation behavior with respect to laxity is to graph the fraction of all dispatches of an operation that miss their respective deadline. Figure 21 depicts this graph for the simulated operations and strategies.

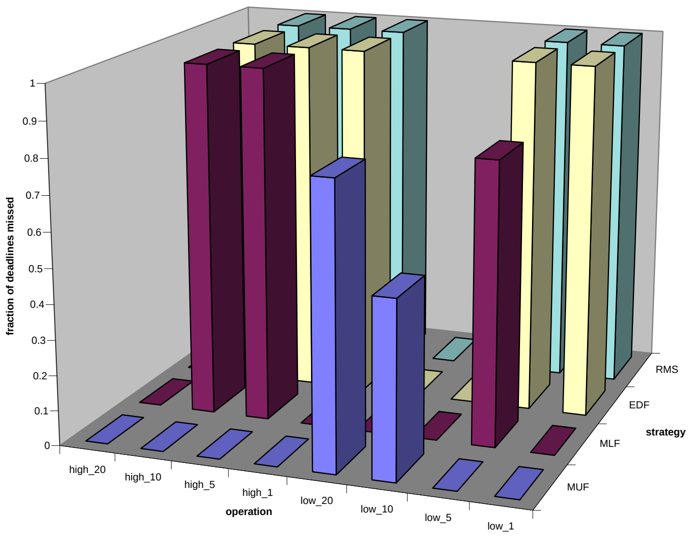

Figure 21: Fraction of Deadlines Missed for each Strategy

The MUF strategy prevents the critical operations from missing their deadlines. It does so at a cost of missed deadlines in the non-critical set. However, MUF minimizes 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 detect scheduling failures prior to deadline. They preempt operations with negative laxity in favor of operations with positive laxity, and thus allow more operations to meet their deadlines.

4.4 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 strategy considers operation criticality in assigning priority, so operations in the critical set make their deadlines in preference to non-critical operations in MUF. 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.

4.5 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 considers fewer operation characteristics and fails to meet at least one of these criteria.

5 The Performance of TAO's Strategized Scheduling Service

The conditions under which we ran the simulations in Section 4 were somewhat idealized. In particular, factors such as run-time overhead for dynamic scheduling mechanisms and OS dispatch latency [30] significantly affect the scheduling behavior of these strategies in actual systems. Therefore, empirical benchmarks are needed to validate the simulation results. To ensure that TAO's strategized scheduling service framework is efficient and predictable, we measured the dispatching overhead in TAO's strategized scheduling service. We 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.

We conducted two experiments. The first determined the run-time cost of dynamic dispatching for end-to-end performance. The second assessed the potential increase in dispatching overhead as varying loads were placed on the dispatching queues described in Section 3.4.6. These tests demonstrated that TAO's dispatching modules can enforce dynamic end-to-end QoS requirements within acceptable levels of overhead.

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.

5.1 Measuring Dynamic Scheduling Overhead in TAO's Real-Time Event Service

The first experiment quantified the dynamic scheduling overhead in TAO's Event Service [33], shown in Figure 22. 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.

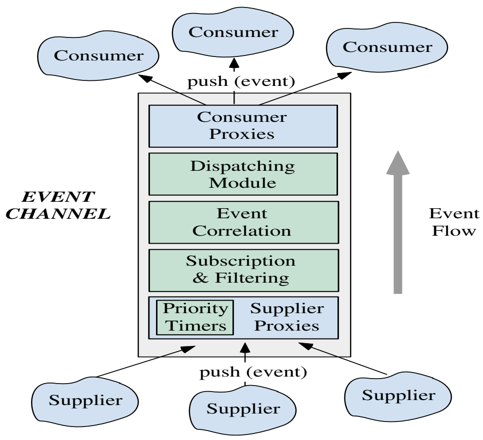

Figure 22: TAO's Event Service Architecture

We measured the latency in event delivery between the high-priority supplier and consumer. This latency included (1) the time required for the TAO run-time scheduler to satisfy the Event Service dispatch module scheduling request plus (2) 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 [11].

TAO's strategized scheduling service was configured with an off-line RMS strategy and an $O(1)$ table lookup 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 remove 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 23. This figure illustrates 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.

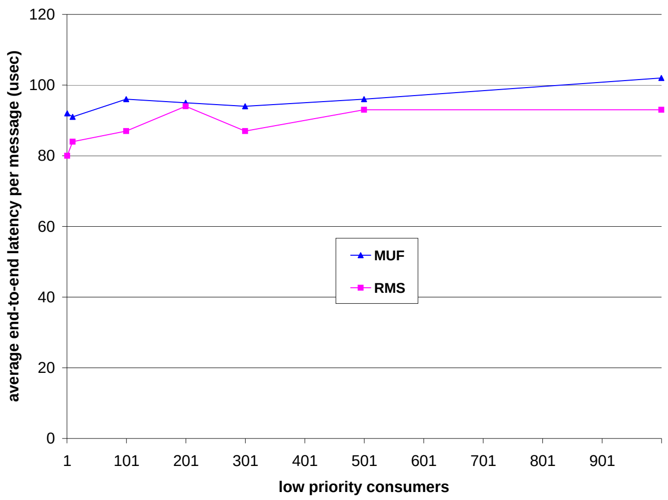

Figure 23: End-to-end Run-time Overhead of Dynamic Scheduling

5.2 Measuring Dynamic Scheduling Overhead in TAO's Dispatching Modules

The experiment described in Section 5.1 established the minimum relative end-to-end overhead for dynamic scheduling in TAO. 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-, and laxity-based queues. The static queue, which was used by the RMS scheduling strategy, 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 and 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 24, there was 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.

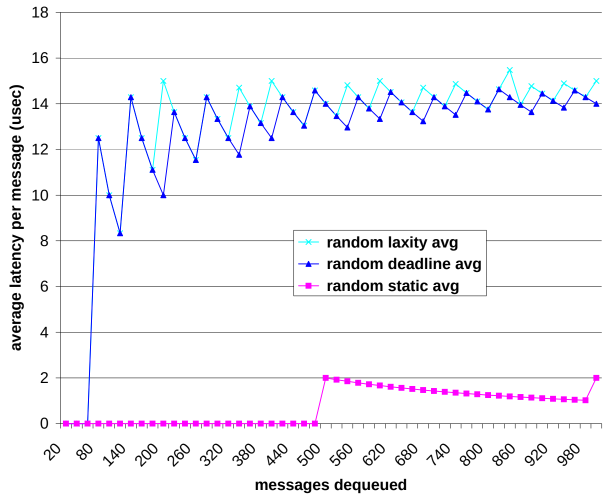

Figure 24: Static and Dynamic Dequeue Overhead

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 25, the overhead per-enqueue operation increased linearly 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.

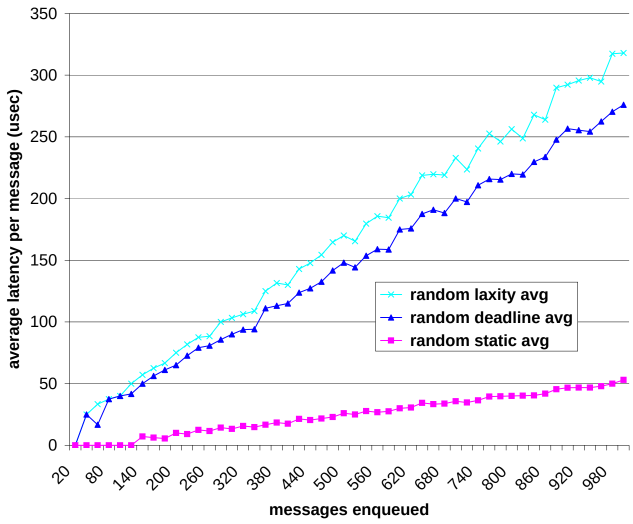

Figure 25: Static and Dynamic Enqueue Overhead

5.3 Analysis of Empirical Results

The tests described in Section 5.1 and Section 5.2 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, degrade performance.

The remainder of this section (1) considers the implications of these results for systems with either moderate or heavy queueing, and (2) discusses alternative dispatching implementations and the conditions under which each may be preferable.

5.3.1 Moderately-loaded systems

Figure 23 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 24, 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 ~ 12 μsecs in the benchmark configuration described in Section 5.2. 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 25, the overhead for laxity- 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 in the experiment described in Section 5.2, the overhead per-enqueue operation for the dynamic queues jumped to ~ 20 μsecs. Although the laxity- 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.

5.3.2 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 25 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 5.3.3 discusses two alternative dispatching priority queue implementations and describes when each are optimal for different numbers of enqueued messages and different application characteristics.

5.3.3 Alternative dispatching mechanisms

The dispatching queues described in Section 3.4.6 are implemented as linked lists. This minimizes the dequeue overhead for the static-, deadline-, 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 $O(n)$ and dequeue operations are $O(1)$. Conversely, in the heap-based priority queue implementation, both enqueue and dequeue operations are $O(\log n)$.

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 $O(1)$ 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.

5.4 Conclusions from Empirical Experiments

The following conclusions can be drawn from the empirical results of our experiments with TAO's strategized scheduling service:

Minimal end-to-end overhead: The minimal end-to-end overhead for dynamic scheduling strategies is comparable to that for 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 ~150 messages enqueued at one time. TAO's strategized scheduling service and dispatching modules can adapt flexibly to alternative queueing implementations, so that for heavier loads, heap-based queues may be preferable.

Our empirical results validate the simulation results presented in Section 4. 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 TAO's strategized scheduling service framework can be achieved both efficiently and predictably.

6 Related Work

Real-time middleware is an emerging field of study. An increasing number of research efforts are focusing on integrating QoS and real-time scheduling into middleware like CORBA. This section compares our work on TAO with related QoS middleware integration research.

CORBA-related QoS research:

• **Mitre Real-time CORBA:** Krupp, *et al.*, [34] at MITRE Corporation were among the first to elucidate the requirements of real-time CORBA systems. A system consisting of a commercial off-the-shelf real-time OS, a CORBA-compliant ORB, and a real-time object-oriented database management system is under development [35]. Similar to TAO's original static scheduling service [9], their initial static scheduling approach used RMS, though a strategy for dynamic deadline monotonic scheduling support has been designed [36].

• **URI TDMI:** Wolfe, *et al.*, are developing a real-time CORBA system at the US Navy Research and Development Laboratories (NRaD) and the University of Rhode Island (URI) [37]. The system supports expression and enforcement of dynamic end-to-end timing constraints through timed distributed operation invocations (`TDMIs`) [38]. A `TDMI` corresponds to TAO's `RT_Operation` [9]. Likewise, an `RT_Environment` structure contains QoS parameters similar to those in TAO's `RT_Info`.

One difference between TAO and the URI approaches is that `TDMIs` express required timing constraints, *e.g.*, deadlines relative to the current time, whereas `RT_Operations` 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 TAO's 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 uses importance as a "tie-breaker" following the analysis of other requirements such as data dependencies. Both NRaD/URI and TAO 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 [39]. QuO is based on CORBA and provides the following support for agile applications running in wide-area networks: (1) provides *run-time performance tuning and configuration* through the specification of operating 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) gives *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) supports *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) [40]. The QuO middleware adds significant value to adaptive real-time ORBs such as TAO. We are currently collaborating the BBN QuO team to integrate the TAO and QuO middleware as part of the DARPA Quorum integration project.

• **UCSB Realize:** The Realize project at UCSB [41] 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 [41].

• **UIUC Epiq:** The Epiq project [42] defines an open real-time CORBA scheme that provides QoS guarantees and run-time scheduling flexibility. Epiq explicitly extends TAO's 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. The Epiq project is work-in-progress and empirical results are not yet available.

• **UCI TMO:** The Time-triggered Message-triggered Objects (TMO) project [43] 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 [44]: (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. Timer-based invocation capabilities are provided through TAO's Real-Time Event Service [33]. Where the TMO model creates new ORB services to provide its time-based invocation capabilities [44], TAO provides a subset of these capabilities by extending the standard CORBA COS Event Service. We believe TMO and TAO are complementary technologies since (1) TMO extends and generalizes TAO's existing time-based invocation capabilities and (2) TAO provides a configurable and dependable connection infrastructure needed by the TMO CNCM service. We are currently collaborating with the UCI TMO team to integrate the TAO and TMO middleware as part of the DARPA Quorum integration project.

Non-CORBA-related QoS research:

• **ARMADA:** The ARMADA project [45, 46] 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.

TAO differs from ARMADA in that most of the real-time infrastructure features in TAO are integrated into its ORB Core [12] and I/O subsystem [11], rather than in a micro-kernel. In addition, TAO implements the OMG CORBA standard, while also providing the hooks necessary to integrate with an underlying real-time I/O subsystem and OS. Thus, the real-time services provided by ARMADA's communication system can be utilized by TAO to support standards-based applications running over a vertically and horizontally integrated real-time system.

• **CMU Publisher/Subscriber:** Rajkumar, *et al.*, [47] at CMU developed a real-time Publisher/Subscriber model that is similar to the TAO's Real-time Event Service [2], *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, TAO's Scheduling Service and real-time Event Service utilize QoS parameters from suppliers and consumers to assure resource access via priorities. One interesting aspect of the CMU Publisher/Subscriber model is the separation of priorities for subscription and data transfer. By handling these activities with different threads, with possibly different priorities, the impact of on-line scheduling on real-time processing can be minimized.

• **UCI RED-Linux Scheduling Framework:** Wang, *et al.* [16], 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*. Wang, *et al.*, decompose scheduling behavior into policy (*allocator*) and mechanism (*dispatching*) components, which are similar to the TAO 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, TAO's scheduling service relies only on commonly available OS features, such as preemptive thread priorities. Therefore, TAO's dispatching mechanisms can leverage 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, TAO's strategized scheduling service 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, TAO's strategized scheduling service implementation 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 TAO and RED-Linux scheduling approaches are complementary.

• **OSU Share-based Scheduling:** Tyan, *et al.* [17], 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.

TAO's strategized scheduling service differs in that it uses priority based scheduling approaches, in order to address applications with hard real-time requirements. In our future research, we are investigating share-based scheduling and its interaction with priority-based scheduling for various classes of real-time applications.

7 Concluding Remarks

Many hard real-time systems, such as avionics mission computing and manufacturing process control systems, have traditionally been scheduled statically using variants of rate monotonic scheduling (RMS). Static scheduling provides assurance of schedulability prior to run-time and can be implemented with low run-time overhead. However, static scheduling handles non-periodic processing inefficiently and treats invocation-to-invocation variations in resource requirements inflexibly. As a consequence, scheduled resources are underutilized and the resulting systems are hard to adapt to meet worst-case processing requirements.

Dynamic scheduling alleviates many limitations of static scheduling. However, purely dynamic scheduling strategies offer little or no control over which operations will miss their deadlines in an overloaded schedule. In addition, dynamic scheduling has a higher run-time cost because certain computations must be performed on-line, so it is necessary to measure this additional overhead and assess its significance.

To quantify the tradeoffs between static and dynamic scheduling algorithms, we have developed a *strategized scheduling service framework* and integrated this framework with TAO [9], which is our high-performance, real-time ORB. TAO offers applications the flexibility to specify and use different scheduling strategies, according to their specific needs. A C++ implementation of TAO's strategized scheduling service framework is available with the TAO ORB at the following URL: `www.cs.wustl.edu/~schmidt/TAO.html`.

This paper describes how we then used TAO's scheduling service to generate simulated dispatching timelines for four scheduling strategies, RMS, EDF, MLF, and MUF, and analyze the latency, laxity, and missed deadlines for the operations dispatched in each simulation. In addition, we used TAO's Event Service and run-time Scheduling Service to empirically measure end-to-end latency with and without queueing. Our results indicate that hybrid static/dynamic scheduling strategies can be used in real-time CORBA applications to (1) offer higher resource utilization than purely static scheduling strategies with acceptable run-time cost, (2) preserve the scheduling guarantees for critical operations even under an overloaded schedule, and (3) provide applications the flexibility to adapt to varying application requirements and platform features.

Our simulations and empirical measurements provide a foundation upon which we will develop practical guidelines for configuring and using appropriate scheduling strategies for real-time CORBA applications. We are currently exploring the following areas in our future research on dynamic scheduling of real-time CORBA operations:

Varying operation characteristics: Additional simulations and empirical measurements are needed to assess the impact of varying the values of different operation characteristics on the performance of the scheduling strategies.

Distributed scheduling behavior: Further empirical measurements are needed to determine the impact of factors such as network latency on the end-to-end performance of dynamically scheduled distributed systems.

Available platform features: We are exploring the impact of various platform-specific features, such as preemptive multi-threading, on run-time scheduling behavior.

Application requirements: A detailed examination of the impact of application specific requirements, such as policies for handling missed deadlines, will help guide the development of additional protocols for dynamically scheduled systems.

8 Acknowledgments

This work was funded in part by Boeing. We gratefully acknowledge the support and direction of the Boeing Principal Investigator, Bryan Doerr. In addition, we would like to thank Fred Kuhns for his guidance on the details of TAO's real-time I/O subsystem, and Priya Narasimhan for her extensive comments on this paper.

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] 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), ACM, October 1997.

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

[4] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997.

[5] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[6] J. R. Newport, *Avionics Systems Design*. Boca Raton, Florida: CRC Press, 1994.

[7] D. 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)*, vol. 2, no. 1, 1996.

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

[9] 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.

[10] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the 5^{th} Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[11] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the 5^{th} IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), pp. 154–163, IEEE, June 1999.

[12] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems*, To appear 1999.

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

[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] S. Wang, Y.-C. Wang, and K.-J. Lin, "A General Scheduling Framework for Real-Time Systems," in *IEEE Real-Time Technology and Applications Symposium*, IEEE, June 1999.

[17] H.-Y. Tyan and J. C. Hou, "A rate-based message scheduling paradigm," in *Fourth International Workshop on Object-Oriented, Real-Time Dependable Systems*, IEEE, January 1999.

[18] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Norwell, Massachusetts: Kluwer Academic Publishers, 1993.

[19] 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.

[20] 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.

[21] A. K. Mok and D. Chen, "A Multiframe Model for Real-Time Tasks," *IEEE Transactions on Software Engineering*, vol. 23, pp. 635–645, October 1997.

[22] C.-C. J. Han, "A Better Polynomial-Time Schedulability Test for Real-Time Multiframe Tasks," in *IEEE Real-Time Systems Symposium*, (Madrid, Spain), IEEE, Dec. 1998.

[23] D. B. Stewart, D. E. Schmitz, and P. K. Khosla, "Implementing Real-Time Robotic Systems using CHIMERA II," in *Proceedings of 1990 IEEE International Conference on Robotics and Automation*, (Cincinnatti, OH), 1992.

[24] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the 2^{nd} C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[25] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 1999, to appear.

[26] Z. D. Dittia, G. M. Parulkar, and J. R. Cox, Jr., "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), pp. 179–187, IEEE, April 1997.

[27] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[29] R. Lachenmaier, "Open Systems Architecture Puts Six Bombs on Target." www.cs.wustl.edu/∼schmidt/TAO-boeing.html, Dec. 1998.

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

[31] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

[32] 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), IEEE, June 1998.

[33] T. H. Harrison, C. O'Ryan, D. L. Levine, and D. 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.

[34] B. Thuraisingham, P. Krupp, A. Schafer, and V. 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, Oct. 1994.

[35] "Statement of Work for the Extend Sentry Program, CPFF Project, ECSP Replacement Phase II," Feb. 1997. Submitted to OMG in response to RFI ORBOS/96-09-02.

[36] G. Cooper, L. C. DiPippo, L. Esibov, R. Ginis, R. Johnston, P. Kortman, P. Krupp, J. Mauer, M. Squadrito, B. Thuraisingham, S. Wohlever, and V. F. 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), IEEE, December 1997.

[37] V. F. Wolfe, L. C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh, and R. Johnston, "Real-Time CORBA," in *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, (Montréal, Canada), June 1997.

[38] V. Fay-Wolfe, J. K. Black, B. Thuraisingham, and P. Krupp, "Real-time Method Invocations in Distributed Environments," Tech. Rep. 95-244, University of Rhode Island, Department of Computer Science and Statistics, 1995.

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

[40] G. Kiczales, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[41] V. Kalogeraki, P. Melliar-Smith, and L. 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), IEEE, December 1997.

[42] 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.

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

[44] K. Kim and E. Shokri, "Two corba services enabling tmo network programming," in *Fourth International Workshop on Object-Oriented, Real-Time Dependable Systems*, IEEE, January 1999.

[45] A. Mehra, A. Indiresan, and K. G. Shin, "Structuring Communication Software for Quality-of-Service Guarantees," *IEEE Transactions on Software Engineering*, vol. 23, pp. 616–634, Oct. 1997.

[46] 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), IEEE, December 1997.

[47] R. Rajkumar, M. Gagliardi, and L. 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.

[48] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

A Overview of the CORBA Reference Model

CORBA Object Request Brokers (ORBs) allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware. Figure 26 illustrates the key components in the CORBA reference model that collaborate to provide this degree of portability, interoperability, and transparency.[7] Each component in the CORBA reference model is outlined below:

[7]This overview only focuses on the CORBA components relevant to this paper. For a complete synopsis of CORBA's components see [3].

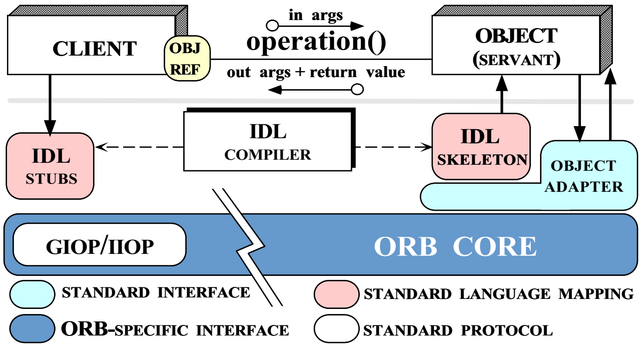

Figure 26: Key Components in the CORBA 2.x Reference Model

Client: A client is a role that obtains references to objects and invokes operations on them to perform application tasks. Objects can be remote or collocated relative to the client. Ideally, a client can access a remote object just like a local object, *i.e.*, `object→operation(args)`. Figure 26 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object.

Object: In CORBA, an object is an instance of an OMG Interface Definition Language (IDL) interface. Each object is identified by an *object reference*, which associates one or more paths through which a client can access an object on a server. An *Object Id* associates an object with its implementation, called a servant, and is unique within the scope of an Object Adapter. Over its lifetime, an object has one or more servants associated with it that implement its interface.

Servant: This component implements the operations defined by an OMG IDL interface. In languages like C++ and Java that support object-oriented (OO) programming, servants are implemented using one or more class instances. In non-OO languages, like C, servants are typically implemented using functions and `structs`. A client never interacts with servants directly, but always through objects identified by object references.

ORB Core: When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. An ORB Core is implemented as a run-time library linked into client and server applications. For objects executing remotely, a CORBA-compliant ORB Core communicates via a version of the General Inter-ORB Protocol (GIOP), most commonly the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol. In addition, custom Environment-Specific Inter-ORB protocols (ESIOPs) can also be defined.

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern [28] and provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common data-level representation. Conversely, skeletons implement the *Adapter* pattern [28] and demarshal the data-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language like C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [48].

Object Adapter: An Object Adapter associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. CORBA 2.2 portability enhancements [3] define the Portable Object Adapter (POA), which supports multiple nested POAs per ORB. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a smaller and simpler ORB that can support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.

B Generalized Schedulability Analysis

It is not strictly necessary to know all operations in advance in order to schedule them using the canonical definitions of EDF or MLF. However, the real-time applications we have worked with do exhibit this useful property. If all operations are known in advance, off-line analysis of schedule feasibility is possible for RMS, EDF, MLF, and MUF.

The output of each of the scheduling strategies in TAO is a *schedule*. This schedule defines a set of operation dispatching priorities, dispatching subpriorities, and a minimum critical dispatching priority. Our goal in this appendix is to present a feasibility analysis technique for these schedules, that is independent of the specific strategy used to produce a particular schedule. Such an analysis technique must establish invariants that hold across all urgency and dispatching priority mappings. By doing this, the off-line schedule feasibility analysis (1) decouples the application from the details of a particular scheduling strategy, and (2) allows alternative strategies to be compared for a given application.

The remainder of this appendix is organized as follows. Section B.1 discusses the notion of a schedule's *frame size*. Section B.2 describes how we measure a schedule's CPU utilization. Finally, Section B.3 describes the generalized schedule feasibility analysis technique, which is based on a schedule's utilization, frame size, and the respective priorities of the operations.

B.1 Frame Size

The frame size for a schedule is the minimum time that can contain all possible phasing relationships between all operations. The frame size provides an invariant for the largest time within which all operation executions will fit. This assumes, of course, that the scheduling parameters, such as rates and worst-case execution times, specified by applications are not exceeded by operations at run-time.

When the periods of all operations are integral multiples of one another, *e.g.*, 20 Hz, 10 Hz, 5 Hz, and 1 Hz, the operations are said to be *harmonically related*. Harmonically related operations have completely nested phasing relationships. Thus, the arrival pattern of each subsequently shorter period fits exactly within the next longer period. For harmonically related operations, the frame size is simply the longest operation period.

Operations that are not harmonically related come into and out of phase with one another. Therefore, they do not exhibit the nesting property. Instead, the pattern of arrivals only repeats after all periods come back into the same phasing relationships they had at the beginning.

This observation leads to the invariant that covers both the harmonic and non-harmonic cases. The frame size in both cases is the product of all non-duplicated factors of all operation periods. For non-harmonic cases, we calculate this value by starting with a frame size of one time unit and iterating through the set of unique operation periods. For each unique period, we (possibly) expand the frame size by multiplying the previous frame size by the greatest common divisor of the previous frame size and the operation period. For harmonic cases, all operation periods are factors of the longest operation period. Therefore, the longest operation period is the frame size.

Figure 27 depicts the relationships between operation periods and frame size for both the harmonic and non-harmonic cases. For harmonically related operation rates, all of the smaller periods fit evenly into the largest period. Therefore, the largest operation period *is* the frame size. For non-harmonically related rates, the frame size is larger than the largest operation period, because it is a multiple of all of the operation periods.

B.2 Utilization

Total CPU utilization is the sum of the actual execution times used by all operation dispatches over the schedule frame size, divided by the frame size itself. TAO's strategized scheduling service calculates the maximum total utilization for a given

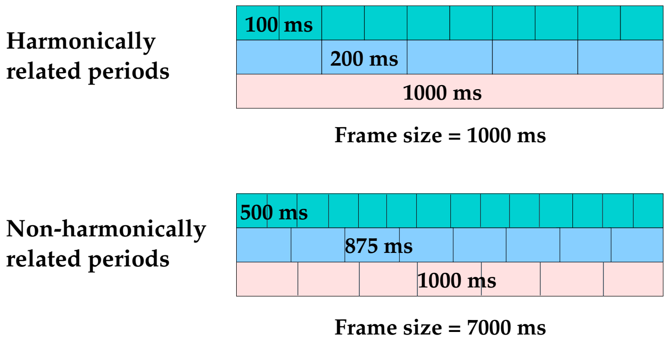

Figure 27: Frame Size Examples for Harmonic and Non-Harmonic Cases

schedule by summing, over all operations, the fraction of each operation's period that is consumed by its worst-case execution time, according to the following formula:

$$\mathcal{U} = \sum_{\forall k} C_k / T_k$$

where, for each operation k, C_k is its worst case execution time, and T_k is its period.

In addition to total utilization, TAO's scheduling service calculates the CPU utilization by the set of critical operations. This indicates the percentage of time the CPU is allocated to operations whose completion prior to deadline is to be enforced. Operations whose assigned dispatching priority is greater than or equal to the minimum critical priority bound are considered to be in the critical set. In the RMS, EDF, and MLF scheduling strategies, the entire schedule is considered critical, so the critical set utilization is the same as total utilization.

If the total utilization exceeds the *schedulable bound*, TAO's scheduling service also stores the priority level previous to the one that exceeded the schedulable bound. This previous priority level is called the *minimum guaranteed priority level*. Operations having dispatching priority greater than or equal to the minimum guaranteed priority level are assured of meeting their deadlines. In contrast, operations having dispatching priority immediately below the minimum guaranteed priority level may execute prior to their deadlines, but are not assured of doing so. If the total utilization does not exceed the schedulable bound, the lowest priority level in the system is the minimum guaranteed priority level, and all operations are assured of meeting their deadlines.

B.3 Schedule Feasibility

It may or may not be possible to achieve a *feasible* schedule that utilizes 100% of the CPU. Achieving 100% utilization depends on the phasing relationships between operations in the schedule, and the scheduling strategy itself. The maximum percentage of the CPU that can be utilized is called the *schedulable bound*.

The schedulable bound is a function of the scheduling strategy and in some cases of the schedule itself. A schedule is *feasible* if and only if all operations in the critical set are assured of meeting their deadlines. The critical set is identified by the minimum critical priority. All operations having dispatching priority greater than or equal to the minimum critical priority are in the critical set.

The schedulability of each operation in the critical set depends on the worst-case operation arrival pattern, which is called the *critical instant*. The critical instant for an operation occurs when the delay between its arrival and its completion is maximal [14]. For the preemptive-by-urgency dispatching model described in Section 3.4.5, the critical instant for an operation occurs when it arrives simultaneously with all other operations.

For other dispatching models, the critical instant for a given operation differs slightly. It occurs only when the operation arrives immediately after another operation that will cause it the greatest *additional* preemption delay was dispatched. Further, it only occurs when the operation arrives simultaneously with all operations other than the one causing it additional preemption delay. If an operation is schedulable at its critical instant, it is assured of schedulability under any other arrival pattern of the same operations.

A key research challenge in assessing schedule feasibility is determining whether each operation has sufficient time to complete its execution prior to deadline. The deadline for an operation at its critical instant falls exactly at the critical instant plus its period. Not only must a given operation be able to complete execution in that period, it must do so in the time that is not used by preferentially dispatched operations. All operations that have higher dispatching priority than the current operation will be dispatched preferentially. All operations that have the same dispatching priority, but have deadlines at or prior to the deadline of the current operation, must also be considered to be dispatched preferentially.

The goal of assessing schedule feasibility off-line in a way that (1) is independent of a particular strategy, and (2) correctly determines whether each operation will meet its deadline, motivates the following analysis. TAO's strategized scheduling service performs this analysis for each operation off-line. We call the operation upon which the analysis is being performed the *current operation*. The number of arrivals, during the period of the current operation, of an operation having higher dispatching priority than the current operation is given by $\lceil T_c / T_h \rceil$, where T_c and T_h are the respective periods of the current operation and the higher priority operation. The time consumed by the higher priority operation during the period of the current operation is given by

$\lfloor T_c/T_h \rfloor C_h + \min(T_c - \lfloor T_c/T_h \rfloor T_h, C_h)$, where the min function returns the minimum of the values, and C_h is the computation time used for each dispatch of the higher priority operation.

Similarly, the number of deadlines of another operation having the same dispatching priority as the current operation is given by $\lfloor T_c/T_s \rfloor$, where T_s is the period of the other operation having the same dispatching priority as the current operation. The time consumed by the other same priority operation over the period of the current operation is given by $\lfloor T_c/T_s \rfloor C_s$, where C_s is the computation time used by the other same priority operation [14]. Figure 28 illustrates the various possible relationships between the periods of operations in two priority levels.

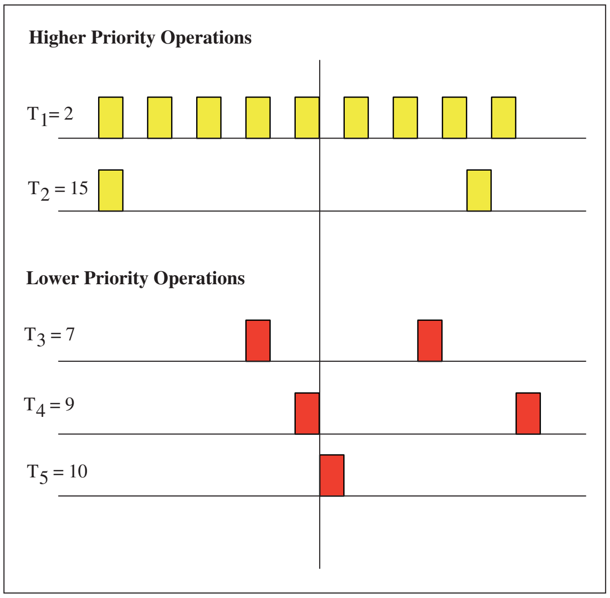

Figure 28: Schedulability of the Current Operation

Choosing the fourth operation, with period T_4, as the current operation, the number of arrivals of each of the higher priority operations is as expected: $\lceil T_4/T_1 \rceil = \lceil 9/2 \rceil = \lceil 4.5 \rceil = 5$ and $\lceil T_4/T_2 \rceil = \lceil 9/15 \rceil = \lceil 0.6 \rceil = 1$. The number of deadlines of operations having the same priority level is also as expected: $\lfloor T_4/T_3 \rfloor = \lfloor 9/7 \rfloor \doteq \lfloor 1.3 \rfloor = 1$ and $\lfloor T_4/T_4 \rfloor = \lfloor 9/9 \rfloor = \lfloor 1.0 \rfloor = 1$ and $\lfloor T_4/T_5 \rfloor = \lfloor 9/10 \rfloor = \lfloor 0.9 \rfloor = 0$.

Having established the time consumed by an operation having higher dispatching priority than the current operation as $\lfloor T_c/T_h \rfloor C_h + \min(T_c - \lfloor T_c/T_h \rfloor T_h, C_h)$, and the time consumed by an operation having the same dispatching priority as the current operation as $\lfloor T_c/T_s \rfloor C_s$, it is now possible to state the invariant that must hold for all operations having dispatching priority λ to be schedulable:

$$\forall \{ j, k \in \mathcal{S} \mid (p(j) = \lambda) \wedge (p(k) >= \lambda) \}$$

$$\left(\left[\begin{array}{c} C_{wcpd(j)} \;+ \displaystyle\sum_{p(k)>=\lambda} \lfloor T_j/T_k \rfloor C_k \;+ \\ \displaystyle\sum_{p(k)>\lambda} \min\,(T_j - \lfloor T_j/T_k \rfloor T_k \;,\; C_k) \end{array} \right] <= T_j \right)$$

$\mathcal{S}$ is the set of all operations in the schedule. The function $p(j)$ simply returns the priority assigned to operation j. $C_{wcpd(j)}$ is the worst-case preemption delay for operation j. Operation j suffers a preemption delay if and only if it arrives while an operation in the same dispatching priority level that does not have a deadline within operation j's period is executing. Operations that have deadlines within operation j's period must be counted anyway, and thus do not impose any *additional* delay, should operation j arrive while they are executing. The worst-case preemption delay for operation j is the longest execution time of any operation that has a longer period: if there are no such operations, $C_{wcpd(j)}$ is zero.

For each current operation having dispatching priority λ to be schedulable, the following must hold. All deadlines of operations having the same dispatching priority or higher, including the deadline of the current operation itself, plus $C_{wcpd(j)}$, plus any time scheduled for higher priority operations that arrive within but do not have a deadline within the period of the current operations, must be schedulable within the period of the current operation. This invariant is evaluated for each decreasing dispatching priority level of a schedule, from the highest to the lowest. The lowest dispatching priority level for which the invariant holds is thus identified as the minimum priority for which schedulability of all operations can be guaranteed, known as the *minimum guaranteed priority*.

In summary, the schedule feasibility analysis technique presented in this appendix establishes and uses invariants that hold across all urgency and dispatching priority mappings. This gives applications the ability to examine different scheduling strategies off-line, and discard those that do not produce feasible schedules for their particular operation characteristics. Further, it decouples applications from the details of any particular scheduling strategy, so that changes in strategies to not require changes in their operation characteristics.