DYNAMIC SCHEDULING STRATEGIES FOR AVIONICS MISSION COMPUTING

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

Abstract
Avionics mission computing systems have traditionally been scheduled statically. Static scheduling provides assurance of schedulability prior to run-time and can be implemented with low run-time overhead. However, static scheduling handles non-periodic processing inefficiently, and treats invocation-to-invocation variations in resource requirements inflexibly. As a consequence, processing resources are underutilized and the resulting systems are hard to adapt to meet worst-case processing requirements.

Dynamic scheduling has the potential to offer relief from some of the restrictions imposed by strict static scheduling approaches. Potential benefits of dynamic scheduling include better tolerance for variations in activities, more flexible prioritization, and better CPU utilization in the presence of non-periodic activities. However, the cost of these benefits is expected to be higher run-time scheduling overhead and additional application development complexity. This report reviews the implications of these tradeoffs for avionics mission computing systems and presents experimental results obtained using the Maximum Urgency First dynamic scheduling algorithm.

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 avionics mission computing systems, impose severe constraints on the design and implementation of real-time OO middleware. Avionics mission computing applications manage sensors and operator displays, navigate the aircraft’s course, and control weapon release.

Middleware for avionics mission computing must support applications with both deterministic and statistical real-time QoS requirements. Support for deterministic real-time requirements is necessary for mission computing tasks that must meet all their deadlines, e.g., weapon solutions and navigation. Support for statistical real-time requirements is desirable for tasks such as built-in-test and low-priority display queues, which can tolerate minor fluctuations in scheduling and reliability guarantees, but nonetheless require QoS support.

1.2 Design and Implementation Challenges
Figure 1 illustrates the architecture of an avionics mission computing application developed at Boeing [1] using OO middleware components and services based on the Object Management Group’s Common Object Request Broker Architecture (CORBA) [2]. CORBA Object Request Brokers (ORBs) allow clients to invoke operations on target object implementations without concern for where the object resides, what language the object is written in, the OS/hardware platform, or the type of communication protocols and networks used to interconnect distributed objects [3]. To achieve these benefits for avionics applications, however, requires
the resolution of the following design and implementation challenges:

**Scheduling assurance prior to run-time:** In avionics applications, the consequences of missing a critical deadline at run-time can be catastrophic. For example, failure to process an input from the pilot within the necessary time frame could be disastrous, especially in critical situations such as air-to-air engagement or weapons release. Therefore, it is essential to validate that all critical processing deadlines will be met prior to run-time.

**Severe resource limitations:** Processing must be minimized due to limited resource availability, such as weight and power consumption restrictions. A consequence of using static, off-line scheduling is that worst-case processing requirements drive the schedule. Therefore, resource allocation and scheduling must always accommodate the worst case, even in non-worst case scenarios.

**Distributed Processing:** In complex avionics systems, mission processing must be distributed over several physical processors and computations on separate processors must communicate effectively. Clients running on one processor must be able to invoke operations on servants in other processors. Likewise, the allocation of operations to processors should be flexible, *e.g.*, it should be transparent whether a given operation resides on the same processor as the client that invokes it.

**Testability:** Avionics software is complex, critical, and long-lived. Maintenance is particularly problematic and expensive [4]. A large percentage of software maintenance involves testing. Current scheduling approaches are validated by extensive testing, which is tedious and non-comprehensive. Thus, analytical assurance is essential to help reduce validation costs by focusing the requisite testing on the most strategic system components.

**Adaptability across product families:** Current avionics applications are custom-built for a specific product family. Development and testing costs can be reduced if large, common portions can be factored out. In addition, validation and certification of components can be shared across product families, potentially reducing development time and effort.

The remainder of this paper is organized as follows: Section 2 reviews the drawbacks of off-line, static scheduling and introduces the dynamic scheduling strategy we are evaluating, *Maximum Urgency First* (MUF) [5]. Section 3 presents experimental results showing the cost of dynamic scheduling. Section 4 presents concluding remarks.

## 2 Dynamic Scheduling Strategies

This section describes the limitations of purely static scheduling and outlines the potential benefits of applying dynamic scheduling. We also evaluate the limitations of purely dynamic scheduling strategies. This evaluation motivates the hybrid static/dynamic MUF scheduling approach for CORBA operations used by TAO’s real-time scheduling service (described in [6]).

### 2.1 Limitations of Static Scheduling

Many hard real-time systems have traditionally been scheduled statically using rate monotonic
scheduling (RMS) [7]. Static scheduling provides schedulability assurance prior to run-time and can be implemented with low run-time overhead [6]. However, static scheduling has the following disadvantages:

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

Unfortunately, with 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 harmonically related 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.

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

In general, static scheduling limits the ability of real-time systems to adapt to changing conditions and changing configurations. In addition, static scheduling compromises resource utilization to guarantee access to resources at run-time. To overcome the limitations of static scheduling, therefore, we are investigating the use of dynamic strategies to schedule CORBA operations for applications with real-time QoS requirements.

### 2.2 Synopsis of Scheduling Terminology

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

**RT\(_\text{Operation}\) and RT\(_\text{Info}\):** In TAO, an RT\(_\text{Operation}\) is a scheduled CORBA operation [6]. In this paper, we use operation interchangeably with RT\(_\text{Operation}\). An RT\(_\text{Info}\) struct is associated with each operation and contains its QoS parameters. The RT\(_\text{Info}\) structure contains the following operation characteristics shown in Figure 3 and described below.

\begin{verbatim}
struct RT_Info {
  Criticality criticality;
  Time worstcase_exec_time;
  Period period;
  Importance importance;
  Dependency Info dependencies;
};
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure3}
\caption{TAO's Real-time CORBA Operation Characteristics}
\end{figure}

- **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 the application if

---

\(^{1}\)A sporadic server [8] reserves a portion of the schedule to allocate to aperiodic events when they arrive.

\(^{2}\)Priorities can be changed via mode changes [6], but that is too coarse to capture invocation-to-invocation variations in the resource requirements of complex applications.
they fail to complete execution before their deadlines. Some scheduling strategies, such as MUF, take criticality into consideration, so that more critical operations are given priority over less critical ones.

- **Worst-case execution time:** This is the longest time it can take to execute a single dispatch of the operation.

- **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 distinguish between operations that otherwise would have identical priority.

- **Dependencies:** An operation depends on another operation if it is invoked only via a flow of control from the other operation.

**Scheduling Strategy:** A scheduling strategy (1) takes the information provided by an operation’s `RT_Info`, (2) assigns an urgency to the operation based on its static priority, dynamic subpriority, and static subpriority values, (3) maps urgency into dispatching priority and dispatching subpriority values for the operation, and (4) provides dispatching queue configuration information so that each operation can be dispatched according to its assigned dispatching priority and dispatching subpriority. The key elements of this transformation performed by the scheduling strategy are shown in Figure 2 and defined as follows:

- **Urgency:** Urgency [9] 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 2 illustrates these relationships.

- **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 “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. Op-
operations with higher dispatching priorities are dispatched 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 dispatched ahead of operations with the same dispatching priority but 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\(^3\)), a dispatching priority, and a thread priority.

**Dispatching Module**: A dispatching module constructs the appropriate type of queue for each dispatching priority. In addition, it assigns 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, e.g., the I/O subsystem [10], ORB Core [11], and/or the Event Service [1].

### 2.3 Survey of Dynamic Scheduling Strategies

Several other forms of scheduling exist beyond RMS. For instance, Earliest Deadline First (EDF) scheduling assigns higher priorities to operations with closer deadlines. EDF is commonly used for dynamic scheduling because it permits runtime modification of rates and priorities. In contrast, static techniques like RMS require fixed rates and priorities.

Dynamic scheduling does not suffer from the drawbacks described in Section 2.1. If these drawbacks can be alleviated without incurring too much overhead or non-determinism, dynamic scheduling can be beneficial for real-time applications with deterministic QoS requirements. However, many dynamic scheduling strategies do not offer the \textit{a priori} guarantees of static scheduling.

For instance, purely dynamically scheduled systems can behave non-deterministically under heavy loads. Therefore, 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, e.g., if admission control of dynamically generated operations is not performed.

The remainder of this section reviews several strategies for dynamic and hybrid static/dynamic scheduling. These include purely dynamic strategies such as EDF and MLF, and hybrid approaches such as MUF and two-level scheduling.

#### 2.3.1 Purely Dynamic Scheduling Strategies

This section reviews two well known purely dynamic scheduling strategies, Earliest Deadline First (EDF) [12, 7], and Minimum Laxity First (MLF) [9]. These strategies are illustrated in Figure 4 and discussed below. In addition, Figure 4 depicts the hybrid static/dynamic Maximum Urgency First (MUF) [9] scheduling strategy discussed in Section 2.3.2.

\[\text{Figure 4: Dynamic Scheduling Strategies}\]

---

\(^3\)An operation’s laxity is the time until its deadline minus its remaining execution time.
Earliest Deadline First (EDF): EDF [12, 7] 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 dead-
lines are dispatched before those with more distant
deadlines. The EDF scheduling strategy is invoked
whenever a dispatch of an operation is requested.
The new dispatch may or may not preempt the cur-
rently executing operation, depending on the imple-
mentation strategy.

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.

If the operation is dispatched even though it
cannot complete its execution prior to the deadline,
the operation consumes CPU time that could other-
wise be allocated to other operations. If the result of
the operation is only useful to the application prior
to the deadline, then the entire time consumed by
the operation is essentially wasted.

Minimum Laxity First (MLF): MLF [9] refines
the EDF strategy by taking into account operation
execution time. It dispatches the operation whose
laxity is least, as shown in Figure 4. Laxity is de-

defined as the time-to-deadline minus the remaining
execution time.

Using MLF, it is possible to detect that an op-
eration will not meet its deadline prior to the dead-
line itself. If this occurs, a scheduler can reevaluate
the operation before allocating the CPU for the re-
mainning computation time. For example, one strat-

ey is to simply drop the operation whose laxity is
not sufficient to meet its deadline. This strategy may
decrease the chance that subsequent operations will
miss their deadlines, especially if the system is over-
loaded transiently.

Evaluation of EDF and MLF:
\(^4\)A dispatch is a particular execution of an operation.

- **Advantages:** From a scheduling perspec-
tive, 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 2.1 that can occur in RMS is not a factor
since EDF and MLF prioritize operations according
to their dynamic run-time characteristics.

EDF and MLF also handle harmonic and
non-harmonic periods comparably. Moreover, they
respond flexibly to invocation-to-invocation vari-
a
tions 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 [12]. In ad-
dition, both EDF and MLF can dispatch operations
within a single static priority level and need not pri-
oritize operations by rate [12, 9].

- **Disadvantages:** From a performance per-
spective, one disadvantage to 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.
Therefore, the risk of missing a deadline increases
for every operation as the system become over-
loaded.

2.3.2 Maximum Urgency First

The Maximum Urgency First (MUF) [9]
scheduling strategy supports both the determi-

nistic rigor of the static RMS scheduling approach and the
flexibility of dynamic scheduling approaches such
as EDF and MLF.

MUF can assign both static and dynamic pri-

ority components. In contrast, RMS assigns all prior-
ity components statically and EDF/MLF assign
all priority components dynamically. 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, they offer little or no control over which operations will miss their deadlines under overload conditions.

In contrast, MUF gives applications the ability to distinguish operations arbitrarily. MUF allows control over which operations will miss their deadlines. Therefore, it can protect a critical subset of the entire set of operations.

**Dynamic Subpriority:** An operation’s dynamic subpriority is evaluated whenever it must be compared to another operation’s dynamic subpriority. For example, 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 the laxity of an operation.

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 [9]. For example, if an operation has negative laxity prior to being dispatched, it can be demoted in the priority queue, allowing operations that can still meet their deadlines to be dispatched instead.

**Static Subpriority:** In MUF, static subpriority is a static, application-specific, optional priority. 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 to operation that have the same criticality ensures a total dispatching ordering of operations at run-time, for any operation laxity values having the same criticality. A total dispatching ordering ensures that for a given arrival pattern of operation requests, the dispatching order will always be the same. This, in turn, helps improve the reliability and testability of the system.

The variant of MUF used in TAO’s strategized scheduling service enforces a complete dispatching ordering by providing an importance field in the TAO RTInfo CORBA operation QoS descriptor [6]. 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.

### 2.3.3 Hybrid Approaches

Hybrid static and dynamic approaches may be used to combine the benefits of both. Multi-level scheduling integrates different approaches at different scheduling levels. One example is two-level hierarchical scheduling, which allows real-time applications to coexist with non-real-time applications in an open OS environment [13]. Another is standardized in the ARINC Avionics Application Software Standard Interface (APEX) for Integrated Modular Avionics (IMA) [14]. One level consists of partitions, which are executed cyclically and scheduled statically and off-line. Within each partition, application processes are scheduled using potentially more flexible approaches.

Each task in a partition is characterized statically by period (for periodic tasks), deadline within the period, and worst-case execution time. Aperiodic tasks are supported; Audsley and Wellings offer an analysis approach assuming minimum arrival
time for aperiodic task periods [15]. TAO used this same approach initially to handle aperiodic tasks with rate monotonic scheduling and analysis.

APEX Partitions are scheduled cyclically. Each partition is characterized statically by parameters including criticality level, period, and duration. Therefore, a straightforward static scheduling approach can be used.

The APEX approach provides static schedulability analysis and fault tolerance across partitions. However, it suffers from the drawbacks of static scheduling described in Section 2.1. In particular, it is not clear how APEX can appreciably improve resource utilization when compared to conventional static scheduling approaches. For instance, jitter may be high when the period of a task is not a multiple of its partition’s period [15]. In that case, the task could become ready to run at a time when another partition was executing, and therefore would have to wait for its partition’s activation.

3 Dynamic Scheduling Overhead

To assess the run-time cost of dynamic scheduling, we used an experimental setup based on TAO’s Event Channel [16]. It consisted of a single high-priority supplier/consumer pair, and a varied number of low-priority event suppliers and consumers. We measured the latency in event delivery between the high-priority supplier and consumer. This latency included the time required for the TAO run-time scheduler to satisfy the Event Channel dispatch module scheduling request.

The test was run on a Sun Ultra 30 in the RealTime scheduling class, with a single 300 MHz UltraSPARC CPU, for two different scheduling strategies. The static scheduling strategy used off-line RMS and table lookup at runtime. The dynamic strategy used MUF, and therefore required an additional laxity calculation at runtime.

As shown in Figure 5, there appears to be a small (up to 10 percent) overhead for dynamic scheduling.

![Figure 5: The Cost of Dynamic Scheduling](image)

4 Concluding Remarks

As mission system computing evolves to address increasingly complex user requirements, runtime variation and non-periodic activities will become increasingly common. In addition, required system flexibility, in support of changing mission conditions, will no longer be achievable with static modal behavior. Conventional static approaches to scheduling mission computer activities will not support these increasingly complex behaviors.

Dynamic scheduling approaches offer potential solutions to these increasingly complex requirements, but at some run-time cost. The contribution of this work has been the implementation and analysis of selected dynamic scheduling algorithms within a strategized scheduling framework. From this work, we have concluded that the Maximum Urgency First scheduling algorithm has desirable properties for use within a mission computing application and acceptable run-time overhead.

In future work, we plan to measure the run-time cost of dynamic scheduling in TAO’s scheduling service under a variety of operational conditions. Further work will also explore scheduling in distributed systems. Dynamic scheduling appears to be a prerequisite for distributed system scheduling, due to the loose coupling between operations on separate processors.
5 Acknowledgments

This work was funded in part by Boeing. We gratefully acknowledge the support and direction of the Boeing Principal Investigator, Bryan Doerr.

References


