ADAPTIVE SCHEDULING FOR REAL-TIME, EMBEDDED INFORMATION SYSTEMS

Bryan S. Doerr, The Boeing Company, St. Louis, MO

Thomas Venturella, The Boeing Company, St. Louis, MO

Rakesh Jha, Honeywell Technology Center, Minneapolis, MN

Christopher D. Gill, Washington University, St. Louis, MO

Douglas C. Schmidt, Washington University, St. Louis, MO

Introduction

One way to increase software system adaptability is to allocate resources dynamically at run-time rather than statically at design time. For example, fine-grained run-time allocation of processor utilization and network bandwidth creates an opportunity to execute multi-modal operations. This allocation strategy enhances adaptability by combining deterministic and non-deterministic functionality. In general, adaptability is essential to improve versatility and decrease lifecycle maintenance costs for embedded real-time systems.

Consider the following dynamic resource allocation example: as the pilot of an aircraft nears a weapon release point, he must maneuver the aircraft based on information computed by an embedded mission computer. It is also important, however, to simultaneously monitor for and react to possible air threats, using the same mission computer. In legacy mission computing systems, which were designed according to static resource allocation techniques, the pilot must allocate processing resources manually by switching between different mission computer operating modes to perform both of these functions.

Underlying the design of these legacy systems are resource allocation strategies that optimize resource utilization for each mode. These strategies suffer, however, from embedding allocation decisions within application components, which complicates reuse. Likewise extensive system testing must be conducted whenever these strategies change.

Dynamic resource allocation has the potential to provide pilots more operational capability with less manual intervention. In the prior example, for instance, the use of dynamic allocation could simultaneously provide the pilot continuously varying qualities of air threat data and release point information, as measured by target count, accuracy, etc. While implementing this capability within a specific solution would be straightforward, it is more challenging to support *operator* configured *generic* adaptation, using information known only at mission time. It is even more challenging to perform generic run-time adaptation while still ensuring correct overall system operation. It is this type of adaptation, however, that allows mission computing system developers to produce application components that are more stable in the presence of resource variability.

The example above is just one instance of the general evolution of embedded systems from pre-configured “point-solutions” (with strictly controlled inputs and outputs) to operator-configurable systems capable of operating in less deterministic situations. In general, real-time information systems, such as those within fighter aircraft, multimedia applications, and manufacturing plants, are becoming increasingly interconnected with

other real-time and non-real-time systems. Due to the desire to react to information derived from this increased data sharing, real-time information systems are evolving to be more supportive of functional customization later in their deployment lifecycles. This evolution motivates our research into *adaptive* software systems.

In summary, *adaptability* can be defined as an aggregate measure of key software characteristics that support customization of software functionality after initial development. High adaptability occurs when the application provides numerous, significant customization options based on user or system input. While many examples of solution-specific adaptability exist in prior work, our goal is to make adaptive characteristics an integral part of real-time, embedded system software architectures.

Many software architectural precepts are mechanized within a software *framework* that supports application development. A software framework is "a set of cooperating classes that make up a reusable design for a family of related software applications.[1]" Our resulting framework will enable developers to produce adaptive components, even for functionality that does not have initial requirements for adaptability. Our hypothesis is that the resulting applications will be more extensible to future requirements, thereby lowering costs for maintenance and enhancement over the lifecycle of the software.

Project Objectives

The objectives of our research are to develop, demonstrate, and assess adaptive software technologies. Our current phase of this work has focused on the following aspects of adaptability:

- Achieving higher resolution of existing computations and/or adding functionality by exploiting varying CPU availability that arises from changes in the environment
- Increasing the ability to handle functions with non-deterministic execution time, such as those present in non-real-time networks and information management activities
- Increasing flexibility for functional customization during system deployment
- Using commercial standards and products as building blocks so that the resulting solution can be widely used

To achieve these goals, our research has focused on the development of overall application architecture concepts, prototypes of adaptive algorithms within a supporting CORBA-based run-time framework, and the investigation of patterns for developing application functionality that can operate within the framework we have developed. Specifically, we have prototyped an architecture for avionics mission computing that possesses the following adaptive attributes:

- A component model that minimizes dependencies of application components on variable timing and logical characteristics
- Frame-to-frame real-time guarantees for high criticality functions
- Flexible and adaptive dynamic scheduling for low criticality functions
- Mission customization for both high and low criticality functions.

The remainder of this paper is structured as follows. The Design Forces section examines the key design forces originating from legacy mission computing system behavior. This section also outlines design goals and non-functional requirements that must be resolved by any solution. The Adaptive Architecture section describes our adaptive architecture in detail. Within this section, we outline the overall adaptive framework and highlight several features of this framework. We then focus on four of these features specifically,

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

Quality of Service Expression, Dynamic Processor Scheduling, Adaptive Resource Management, and Application Adaptation Control. Finally, the Concluding Remarks section presents a summary of the work and describes our plans for future development and measurement.

Design Forces

Legacy Evolution

Hard real-time applications have traditionally been designed with very a high degree of determinism. This has often meant embedding all decisions related to the order of execution and the layout of the functional call tree within the design, i.e., using a cyclic executive. The resulting system provided a very reliable, but relatively rigid, set of system capabilities. Any adaptability that is present within these systems must be implemented within the application components. Unfortunately, this design couples application logic and resource allocation - a poor encapsulation of independent design forces that are subject to change.

To make legacy real-time systems more adaptable, it is necessary to decouple the functional and non-functional aspects of the design so that functionality can be changed without affecting the timing constraints of the system adversely. For example, moving the scheduling decisions from a cyclic executive into a static scheduling algorithm, such as Rate Monotonic Scheduling[2] (RMS), relaxes the strict ordering dependencies between operations while still maintaining *a priori* deadline guarantees.

[2] Liu, C.L., and Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", JACM, Vol. 20 No. 1, January 1973, pp. 46-61.

Managing Non-Functional Requirements

Within the constraints of legacy design paradigms, adaptability and determinism are opposing forces with limited options for reconciliation. In part, this tension is created by the perception that *all* functionality within the application requires the same degree of determinism. However, a central premise of our research is that this constraint is overly restrictive for many real-time systems. Within the mission computing domain for example, certain functionality, e.g., update of a navigation solution, is indeed required to be activated and completed while satisfying hard real-time constraints. However other functions, e.g., built-in testing, can be run according to more lenient criteria.

Just as relaxing the strict ordering constraint enabled the evolution from cyclic executives to RMS scheduling, relaxing the determinism constraints for *individual* operations enables the evolution to adaptively scheduled systems. This evolution allows real-time systems greater flexibility for adaptation in both functional and non-functional aspects, while still resolving key design forces in various real-time domains. In addition, static scheduling strategies such as RMS may under-utilize the CPU, especially when resource requirements vary significantly at run-time. Using a hybrid static/dynamic scheduling strategy allows greater overall CPU utilization while preserving hard real-time requirements for tasks that require them.

The principal design forces that must be resolved by adaptive architectures for real-time systems include 1) continuing to provide deterministic guarantees for all tasks that require these guarantees, while 2) relaxing guarantees for operations with less stringent requirements, in order to permit flexible and adaptive responses to changing situational factors, and 3) increasing utilization of resources such as the CPU.

Adaptive Architecture

Resolving Key Design Forces

Our approach requires that many scheduling decisions be delayed and determined based on adaptive control information provided at run-time. To accomplish this, our research has the following architectural goals:

- Allow run-time determination of contents and ordering of the CPU schedule
- Increase stability in the presence of dependency and timing variability
- Maximize utilization of available CPU for performing application functions

These goals are discussed in the following paragraphs.

Execution Order: The CPU schedule is an ordered list of functions performed each period or at harmonic periods from a base rate. To satisfy the adaptive objectives described above, real-time system design must support the definition and re-prioritization of application functions by operators and defer concrete scheduling of these functions until later parts in the deployment lifecycle, i.e., until run-time.

Stability: Dependency and timing stability are non-functional aspects of the architecture. Software possessing the quality of *dependency stability* isolates the implementation of the functional algorithm from the effects of changes in the context under which it is executed. High stability is present when an algorithm implementation is only dependent upon information directly expressed in its public interface. Software possessing the quality of *time stability* isolates the consequences of timing changes on the correctness of the application. High stability is present when changes occur in relative execution order, rate, or timing of components, without affecting correct system operation adversely.

Maximization of value: The “value” of an application is a measure of how closely it approximates performing precisely those functions desired by the user. Increased value can be accomplished in two ways: 1) using otherwise idle resources to perform any mission functions and 2) allocating resources to favor higher value computations, where value is determined by a specific set of mission conditions, e.g., proximity to target, presence or absence of airborne threats, etc.

Adaptation Architecture

In current research[3], the term *Quality of Service* (QoS) describes the concept of using selected attributes of software entities as a means of arbitrating access to system resources. Our research has explored several architectural enhancements necessary for the development of a real-time QoS framework. The architectural features discussed below outline these enhancements:

- An interface that allows applications to express the QoS attributes for their operations
- A dynamic scheduling mechanism to utilize frame-to-frame processing availability, based on the operation QoS specified by applications
- A resource management function to perform run-time allocations based on defined execution contexts
- Execution contexts, capable of independent scheduling, to provide adaptation control information to the resource management function

Three additional features are needed to fully support developers in implementing our adaptive architecture:

- Additional classifications for various types of application functions for the purposes of distinguishing between critical and non-critical functions
- A catalog of patterns for implementing adaptable application functions

---

[3] John A. Zinky and David E. Bakken and Richard Schantz, "Architectural Support for Quality of Service for CORBA Objects", Theory and Practice of Object Systems, John Wiley and Sons, 1997, Vol. 3, No. 1,

- Initialization-time services that allow selected component dependencies to be delayed until run-time

The remainder of this section focuses on the first four features of this list. The last three features are work in progress associated with developing QoS-aware applications and are beyond the scope of this discussion.

QoS Expression

In many real-time applications, operations are scheduled at specific rates that are determined during system analysis. Rate selection depends on a number of factors, including required accuracy of the solution, the minimum resources required to achieve this accuracy, and the effects on other operations. An application typically employs an executive that "hard codes" the sequence of all operations in the application and enforces the rate decisions made during system analysis.

After the design activity outlined above is completed, the viability of a specific application to meet its deadlines is determined empirically. Historically, this validation has occurred in a lab after the application coding was virtually complete. Thus, at runtime, the application essentially has no information useful for adaptation.

Due to the testing-intensive nature of this system design process, there have been attempts to migrate to static analysis techniques using Rate Monotonic Scheduling (RMS) algorithms. This migration requires that the application provide QoS information for each executable operation. The required QoS information includes the rate at which the operation will execute and its worst case execution time. The RMS algorithm provides a pessimistic, but predictable, decision about the schedulability of the system tasks. An online dispatcher uses the operation rates to determine the priority at which the different operations should be dispatched at runtime.

While RMS permits static analysis, it does not provide a mechanism for fine-grained adaptation at runtime. The migration to static analysis does provide a framework, however, using the QoS information to specify the adaptation control data critical for fine-grained adaptation. For our current work, we started with this framework to represent operation QoS characteristics, specifically the real-time information descriptor[4] (RT_INFO). We then extended it to support multiple rates of execution.

The primary fields in the RT_Info descriptor are shown in Figure 1. Worst case execution time is a bound on the time consumed by a single execution of the operation. Period indicates the interval between successive invocation requests for the operation. Criticality indicates an operation's significance, i.e., whether violating an operation's timing constraints will compromise the application's essential integrity. Importance is a lesser indicator of significance, which is used as a tiebreaker when other scheduling factors are equal. Finally, dependencies indicate which other operations must execute prior to a particular operation, e.g., to produce an input that is needed by the operation. These parameters are used by the scheduler to assign static priority and sub-priority values, and to configure the dynamic run-time operation dispatching behavior. Some of these fields are determined automatically through configuration runs. Others are supplied by application developers.

Dynamic Processor Scheduling

The first step toward developing an adaptive architecture is to incorporate a mechanism to order application functions flexibly. This mechanism enables the adaptive software architecture to determine a CPU

---

[4] Schmidt, D.C., Levine, D. L., and Mungee, S., "The Design and Performance of Real-Time Object Request Brokers", Computer Communications, Elsevier", Vol. 21, No. 4, April 1998, pp. 294—324.

schedule late in the deployment lifecycle, i.e, at runtime. In conjunction with the development of application components capable of adjusting their behavior according to changing CPU allocation, this mechanism must provide analysis and specification of varying CPU allocation parameters to maintain application wide guarantees.

The next step toward a fully adaptive scheduling mechanism is the introduction of dynamic scheduling capabilities. In spite of the increased adaptation, real-time system must still provide static guarantees for *critical* operations with strict determinism constraints. Therefore, it is desirable to integrate both static and dynamic scheduling capabilities. Hybrid static/dynamic scheduling algorithms, such as *Maximum Urgency First*[5], offer the ability to increase CPU utilization while isolating the effects of non-critical operations on critical ones. By statically assigning higher preemptive priorities to more critical operations, and then dynamically ordering operations within certain static priority levels, it is possible to preserve strict determinism constraints for critical operations while increasing overall CPU utilization.

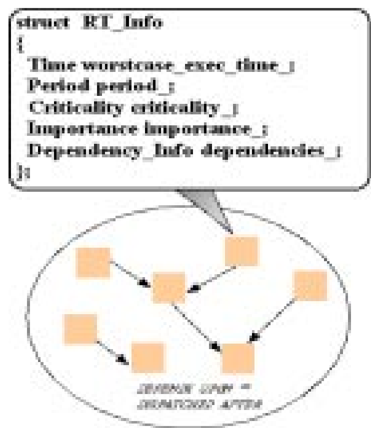

Figure 1. Application QoS Data

The second evolutionary step is the encapsulation of these static and dynamic scheduling policies within a consistent interface, according to the *Strategy*[6] design pattern. This allows different applications, with distinct patterns of criticality and determinism constraints for operations, to instantiate customized scheduling policies within our adaptive CORBA middleware framework. As described above, the application specifies the characteristics of its operations in the RT_INFO descriptor, and the scheduling strategy uses these characteristics to assign static priority information to the operations. The scheduling strategy also assigns configuration information for the dispatching mechanism, which identifies the static or dynamic ordering policy for each static priority level.

The third step toward a fully adaptive scheduling mechanism is performance optimization in the scheduler itself. For off-line scheduling, the computational overhead activities, such as assigning priorities and looking up real-time information descriptors, do not impact the timing behavior of the system itself. However, the scheduler must support on-line schedulability analysis to an on-line adaptive resource manager (ARM) to evaluate adaptation options. The scheduler implementation achieves this functionality by storing internal flags to isolate which portions of the scheduling information must be recomputed, and using O(1) data structures for operation lookup.

Finally, as discussed in the next section, the dynamic scheduler provides real-time feedback about operation progress to ARM.

The scheduling framework described in this section is implemented in the context of *The ACE ORB* (TAO)[7]. TAO is open source CORBA-compliant real-time ORB middleware

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

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

[7] http://www.cs.wustl.edu/~schmidt/TAO.html

developed at Washington University in St. Louis, Missouri, in the Center of Distributed Object Computing headed by Dr. Douglas C. Schmidt. The *Reconfiguration Scheduler* implementation distributed with TAO provides the strategized dynamic scheduling and performance optimized portions of this framework.

By itself, dynamic scheduling improves upon static scheduling by improving utilization in the presence of behaviorally dynamic application components. Our experience suggests that dynamic scheduling likely increases complexity in the testing process, though our research has not systematically analyzed this issue yet. Therefore, if dynamic scheduling is simply used as part of an otherwise point-solution-based architecture, the additional complexity involved in its use could be of questionable benefit.

The real advantage of an underlying dynamic scheduling capability becomes apparent when we consider that it enables us to build a family of solutions, formerly emulated by modal behaviors, but now possible with more flexibility and finer granularity. This capability is discussed further in the next section.

Adaptive Resource Management

In contrast to dynamic scheduling, which adapts to cycle-to-cycle variations in CPU consumption by various operations, adaptive resource management (ARM) adapts to longer-term variations in resource demands and availability. ARM provides two types of adaptation – 1) contraction and expansion of feasible QoS regions, and 2) changing the operational point within a QoS region based on real-time feedback of actual QoS. For example, in a system of rate-adaptive periodic operations, QoS contraction decreases maximum operation rates, whereas feedback adaptation varies operation rates within the currently active min-max range.

Applications are increasingly built on top of services that in turn may be built on top of lower-level services. Therefore, ARM is a hierarchical architecture. Each level provides QoS-based adaptive resource management as described in the preceding paragraph. The ARM implementation allows very flexible construction of adaptive applications. Adaptive applications can be built on top of services that may or may not be adaptive. In addition, different algorithms can be plugged in for resource allocation, detection of adaptation triggers, and processing of real-time feedback.

In the subject implementation, ARM uses several interfaces provided by the dynamic scheduler to access information about operation progress and scheduling feedback. An *Upcall Monitor* provides real-time feedback about which operations meet or do not meet their deadlines. The ARM can query the Upcall Monitor for deadline success and failure information for each operation.

In addition to the real-time feedback provided by the Upcall Monitor, the ARM gathers adaptive control information about schedulability analysis from the scheduler in order to guide its adaptation. The scheduler provides two types of adaptive control information: 1) feasibility of the set of operations presented to it by ARM and 2) sensitivity analysis of schedule feasibility to changes in the operations' parameters.

The adaptive scenario explored in our research consists of operations with either fixed or variable periods of execution. Operations with variable execution times will be considered in our future research. Because CPU utilization by an operation is a function of its execution time and period, the utilization of operations with fixed periods is also fixed. The ARM can adjust utilization by operations with variable periods, however, and the scheduler must provide adaptive control information to guide the ARM in making such adjustments. The scheduler recognizes three primary utilization regions, as shown in Figure 2.

The first utilization region in Figure 2 captures *overload* situations in which the fixed period operations require more than the schedulable utilization bound. This reflects a system failure, and should be corrected prior to deployment. It is useful for the adaptive scheduler to detect and report such situations to the ARM, particularly during pre-deployment testing.

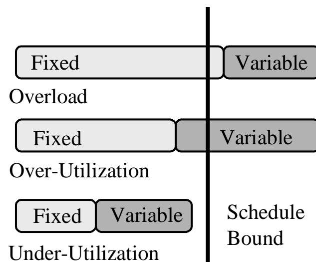

Figure 2.: Operating Regions

The second utilization region shown in Figure 2 captures *over-utilization* situations in which the fixed period operations require less than the schedulable bound, but the variable period operations require more than the remaining available utilization. In these cases the ARM must increase the periods of certain operations, and the scheduler can provide the ARM either a lower bound on the periods of a set of variable operations, or a scaling factor by which to multiply them.

The third and final utilization region shown in Figure 2 captures *under-utilization* situations in which the fixed and variable period operations together require less than the schedulable bound. In these cases, the ARM can decrease the period of certain operations, and the scheduler can provide the ARM either an upper bound on the periods of a set of variable operations, or a scaling factor by which to multiply them

Application Adaptation Control

Figure 3 shows the ARM and dynamic scheduling components discussed earlier as part of the complete application architecture. The final aspect of our QoS framework is related to how the application expresses adaptive control data to framework components dependent upon this information to perform the desired adaptation. We have already discussed how application operation characteristics are expressed to the framework in the *QoS Expression* Section. What remains is a description of the data provided to the ARM at run-time for it to maximize utilization and provide the greatest processing value to the operator.

To accomplish the goals described above, the application be capable of identifying discrete transitions when operations change their relative priorities, at any time before of during runtime. The application provides this adaptation data to a Resource Manager, which works in conjunction with a runtime Scheduler to analyze the state of the application dynamically in order to make more optimal adaptation decisions.

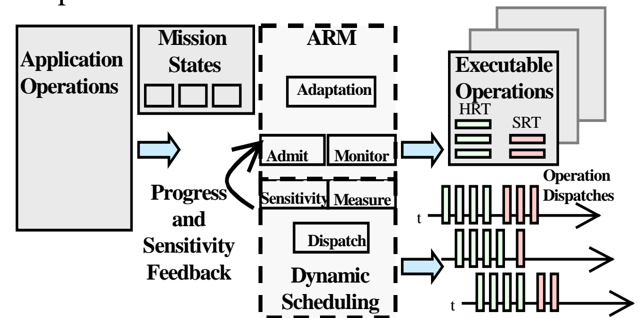

Figure 3. Adaptation Architecture

The mechanism used by an application to express these transition points is called an *operating region*. An operating region is defined as a set of operations the application wishes to execute during a particular phase of execution. The application state can be made up of a variety of domain specific parameters, which for a mission processing example, may include information such as aircraft position or velocity. In addition to the set of operations, the operating region also specifies to the resource manager whether the operations are critical or non-critical and the set of allowable rates at which these operations can be executed.

At runtime, the resource manager determines the operating region in which the

application should be executing. It then queries the dynamic scheduler, using the sensitivity interface, to determine an acceptable set of rates at which to execute the operating region's operations. As previously described, the ARM can seek to optimize operation rates according to a number of different algorithms. Through this mechanism, the resource manager more effectively utilizes the resources of the system by performing functions most valuable to a specific situation.

As part of our work, we also investigated mechanisms for the specification of operating regions. In an avionics mission-processing environment for example, operating regions could be downloaded from a mission planner at the beginning of a flight, or entered by a pilot through an upfront control. In a non-real-time environment, the data could be read from a hard disk or entered through a keyboard. The ability of the architecture to support the specification of the operating region late in the application deployment lifecycle, as opposed to application design time, decouples design of the application from its adaptation, and also provides for an application that can be customized on a situation to situation basis. This was the overall objective of the project.

Finally, the resource manager has been enhanced with an operation progress monitor. The purpose of this monitor is to determine how frequently an operation is being executed. The resource manager can use this additional adaptation data to make a decision to adjust the rate of the operation.

Concluding Remarks

In this work, we have developed a QoS framework that satisfies the goals of maximizing resource allocation according to deployment specific needs and more effectively integrating deterministic and non-deterministic functions. Our approach was to blend macro-level adaptive resource management techniques with micro-level run-time scheduling algorithms within the context of an application that can supply data to guide the adaptation late in the deployment lifecycle, i.e., during system configuration or even at run-time. The algorithms supporting this framework have been developed to be suitable for embedded real-time operation according to the constraints present within the target application domain - avionics mission computing. Future work will focus on measuring qualitative and quantitative improvements resulting from the use of this framework.