CAUGHT IN THE MIDDLE, BETWEEN A ROCK AND A HARD PLACE

Dr. Ben A. Calloni, P.E., Lockheed Martin Aeronautics Company, Fort Worth, TX

Russell B. Kegley, Lockheed Martin Aeronautics Company, Fort Worth, TX

Kenneth Littlejohn, USAF, Air Force Research Lab/IFTA, Wright-Patterson AFB, OH

Jonathan D. Preston, Lockheed Martin Aeronautics Company, Fort Worth, TX

Introduction

Building reliable, high performance DoD avionics is simple: well maybe not simple but straightforward. However, doing the same, using COTS hardware and software products that allow the underlying hardware to be easily replaced, while also allowing the Computer Software Configuration Items (CSCI's) to be added, removed, and modified has forced software and avionics specialists to rethink how these systems are built. Add to that a customer that wants to see overall development and maintenance costs reduced by large margin over a 30 year life cycle and one can see that quite literally, contractors are caught in the middle. The answer is to architect the system using a set of middleware services that disconnects the traditionally highly integrated CSCI's from the underlying hardware.

Lockheed Martin Aeronautics Company in Fort Worth has been involved in just such research and technology transition for the past several years. With combinations of IRAD and CRAD money from AFRL and DARPA, many successful experiments have been completed with

several more currently underway to demonstrate the effectiveness of COTS based middleware. The Advanced Software Execution Platform (ASEP) is built upon COTS Real-time Operating Systems (RTOS) which support the POSIX real-time extensions. The baseline ASEP along with transition of these "experimental concepts" have found their way into designs on the Joint Strike Fighter Mission Systems as well as other advanced F-16 variants. Indeed, this middleware is being woven into the LM Proven Path initiative and is being reviewed for F-22 DMS problems.

Initial IRAD funding demonstrated that traditional tightly coupled cyclic executives could be replaced using commercial Real Time Operating

Relationship Between Research Projects

Systems (RTOS's) that support POSIX thread prioritization and priority inheritance protocols. Follow on CRAD projects demonstrated that these systems could be built and deadline assurance achieved by using Rate Monotonic Scheduling and Analysis techniques developed at Carnegie Mellon University. This formed the basis for middleware that allows applications to run "detached" from the underlying hardware. DARPA's INSERT project demonstrated the viability of extending middleware concepts by providing reliable, fault tolerant upgrades using the Software Engineering Institute's SIMPLEX architecture. Current research under the ASTD contract is adding yet another service to the middleware for looking at scheduling assurance in the face of diminished resource capability such as battle damage. This Quality of Service-based Resource Allocation Model (Q-RAM) is a "new" scheduling methodology in development at Carnegie Mellon University and the Software Engineering Institute. It is being evaluated in the JSF Middleware at LM-Aero.

Incremental Software Evolution for Real Time Systems, or INSERT, is a component of DARPA's Evolutionary Design of Complex Software program (Contract # F33615-97-C-1012, ARPA Order (AO) #D940). The DART's and ASTD projects were funded by AFRL/IFTA (Contract # F33615-95-C-1748 and F33615-97-D-1154 respectively.)

Incremental Software for Real Time Systems (INSERT)

Common Off the Shelf Components

Use of commercial componentry is seen as a significant avenue for reducing costs in DoD systems. As a result, the INSERT architecture (Figure 1) starts with a hardware platform based on commercially available CPU, memory, and I/O devices. On top of that, a commercial Real Time Operating System runs along with COTS interrupt handlers and device drivers. The RTOS is based on the IEEE POSIX standard, which provides maximum portability between Unix-based operating systems as well as real-time extensions to provide deterministic computing.

The INSERT middleware is a set of architectural constructs that makes maximum use of

the hardware and RTOS features to provide high assurance capabilities. The various avionics computer programs, known as Replacement Units in the INSERT system, operate on top of the INSERT middleware. The INSERT middleware insulates the applications from the underlying RTOS and hardware. The middleware also provides dynamic binding and activation of each Replacement Unit, provides run-time safety monitoring, prevents corruption of code and data in one program by another, and allows dynamic reconfiguration.

The avionics application software is partitioned into separable units with each Computer Software Configuration Item (CSCI) hosted in its own virtual memory partition. The CSCI's interact and communicate necessary information using asynchronous messages. To ensure that each of the independently scheduled, dynamic programs are able to meet necessary deadlines, the well-established techniques of Rate Monotonic Scheduling[1] [2] and Rate Monotonic Analysis[3] are used. If a fault is detected, backup Replacement Units can be activated to guarantee a specific level of performance. New capabilities can be safely integrated into the system while it is online.

Figure 1. INSERT Architecture

Basic INSERT Operation

INSERT's basic structure is the Replacement Unit. By using a name-tagged publisher-subscriber communication technique, INSERT can dynamically add or delete tasks from runtime execution. These tagged communications are built upon IEEE POSIX message queues supported by LynxOS. In addition, LynxOS supports 256 different priority levels for process and threads.

This approach provides flexibility in dynamically activating or deactivating the processes as well as their communication links. The middleware is comprised of these data tagged communication processes, a Replacement Manager task, a Decision Logic process, and a timer process to enable the STAT timer.

The INSERT PC runs three different algorithms that can control weapons delivery: the baseline, upgrade, and backup controllers. The baseline controller uses a reliable algorithm that can fly the aircraft through a weapon delivery that has previously undergone extensive testing to ensure a high degree of reliability. The upgrade controller implements an "improvement" to the baseline that the developers need to evaluate. Examples of improvements include increasing the accuracy of the algorithm, modifying it for a moving target, or changing the aircraft maneuvering parameters. The backup or safety controller, which is used only when severe errors are detected in both the baseline and upgrade controllers, discontinues the attack and transitions the aircraft into a terrain following, ground collision avoidance mode. As a result of multiple versions coupled with monitoring and switching infrastructure, INSERT provides a guaranteed level of system performance at all times. Residual errors existing within experimental components are prevented from propagating and the system recovers in real-time by invoking a trusted backup component.

The data flow is shown in Figure 2 with the sequence numbers indicating the order of messages. For a fuller description of the INSERT architecture see Calloni, et.al.[4] .

Figure 2. INSERT Messaging

Performance Statistics

Experimentation determined the "cost" of the fault tolerant INSERT middleware is about 2.2% of the CPU utilization when both baseline and upgrade controllers are executing. The middleware utilization for test runs that has only the baseline controller active is 1.73%. This difference would suggest that the cost of adding extra controller variants would increase the middleware computing expense by about 0.5% utilization for each.

Dynamic Real-Time Scheduling (DARTS)

The Dynamic Avionics Real-Time Scheduling investigated the introduction of dynamic allocation techniques into embedded, resource-constrained real-time systems such as those found in avionics or other military and aerospace systems[5] [6] . The benefits of such a technology throughout system design and implementation are manifold. Releasing system designers from the requirement to bind tasks to processors early in the design process, long before realistic estimates of resource needs can be formulated, frees designers to focus their efforts on solving domain problems. Delaying the time at which task-to-processor binding decisions must be made lets the system automatically cope with the addition of task interactions which were not in the

original design. Dynamic allocation simplifies subsequent system upgrades by building into the platform the ability to relocate tasks as other tasks grow or shrink in resource requirements. Finally, automatic rescheduling of tasks to processors provides system designers with a powerful tool for fault reconfiguration to recover from failures, enhancing overall mission success.

Dynamically allocating tasks to processors can be accomplished in many ways, but typical solutions to the problem have not concentrated on hard real-time environments, i.e., those computing situations where missing a deadline for a task to be executed can result in serious loss of system function or harm to personnel. This research thus focused on fusing Generalized Rate Monotonic Scheduling (GRMS) theory with fast heuristic algorithms for dynamic allocation. The GRMS theory gives system designers the mathematical tools for deciding on whether or not a given allocation of tasks to processors guarantees that all tasks will meet their deadlines. We investigated relatively simple (and thus fast) bin-packing and load balancing heuristic allocation algorithms, since run-time performance is a concern if the work is to support fault recovery reconfiguration. The novel feature was the addition of schedulability checks into the heuristics, effectively preventing the heuristics from ever generating an allocation which would result in a task missing a deadline.

The fused allocation algorithms and GRMS analysis were implemented in a PC-based evaluation tool which allowed system designers to produce models of the tasks and processors in their system and see the results of allocations in real time. As part of this study, application suites from three existing and forward-looking avionics systems were modeled using this evaluation tool. Analysis of these application suites revealed concrete ways they could be improved upon with better design practices.

Advantages of Constraint-Based Modeling

A major theme of the DARTS program was to develop a minimal, orthogonal constraint-based abstraction for expressing task-to-processor and task-to-task relationships. While the targeted system for implementing and exercising these constraints was a PC-based simulation environment,

the abstractions were intended to be directly usable in a real-time embedded avionics execution environment.

Scheduling via expressing constraints is somewhat new in the avionics domain. Past avionics software suites have been designed around rigid scheduling models in which the assignment of tasks to processors and the order of task executions are predetermined by system designers. In this sort of scheme, a task requiring a particular type of processor or I/O device would have been hand-placed by the system designers on the proper processor type. Since the DARTS program was to investigate ways of dynamically assigning tasks to processors based on resource and mission requirements, these implicit decisions needed to be made explicit. Advantages in expressing constraints explicitly include:

- Expressing constraints explicitly invites system designers to carefully consider task-to-processor and task-to-task allocation decisions. Rather than just assign a task to a given processor due to organizational or historical reasons, the designers are given the opportunity to state the reason for a constraint in terms directly usable by an avionics execution platform.
- Explicit constraint specification enriches the design rationale captured during system design. Representing this information in the design artifacts is quite easy to do at system design time.
- Capturing explicitly the constraints on task allocation and scheduling provides critical information about the system design to downstream maintenance and upgrade activities.

Task and Processor Modeling Primitives

The task model for DARTS characterized tasks in terms of their worst-case execution time, task period, deadline, and blocking and jitter factors, if applicable. Processors were modeled by their physical memory, execution rate relative to other processors, and status (i.e., up/down).

In addition to these generally accepted descriptions, we added several other primitives.

We modeled the dependencies between tasks that share a local memory resource, or which incur heavy intertask communication which should not be carried out over a network, as a group. In the modeling language the tasks are simply tagged with an arbitrary descriptive name. The scheduling algorithms recognize and co-locate tasks which are identified as a group.

Task-to-processor constraints were modeled by the addition of a "class tag" attribute to both tasks and processors. A task with a given class tag can be allocated only to processors with that tag; multiple class tags specify constraints in an AND manner. This construct proved to be powerful, capable of expressing differences in instruction set architectures, special I/O connections, functional groupings, etc. Tasks with no class tags are assumed to be allocatable to any processor.

Heuristic Allocation Algorithms

We investigated variations on bin-packing and load-balancing algorithms. Typical varieties of these algorithms sort the items to be allocated in decreasing size and place them, one at a time, into the containers, which are sorted by decreasing fullness (bin-packing) or increasing fullness (load-balancing.)

Figure 3. Bin-Packing and Load-Balancing Contrasted

For this work, however, the interactions between task periods make it difficult to determine the "size" of a task in terms of CPU usage. For this reason the heuristic algorithms run a GRMS scheduling test to make sure that a task which seems to fit on a CPU results in a schedulable task set. In addition, we extended the notion of "biggest" task to "most constrained", which included the notion of grouping and class tag constraints. These algorithms proved to be fast

enough to be practical for dynamic use in an avionics platform.

DARTS Conclusions

The goal of demonstrating that real avionics application designs could be automatically scheduled using relatively simple and fast heuristic allocation algorithms was achieved. Capturing and representing task and processor information invites the system designers to document the scheduling constraints. We believe that removing artificial, platform-induced requirements from applications development will assist in producing more robust, reusable avionics applications.

The simple, fast heuristic dynamic scheduling algorithms proved to be of sufficiently low computational complexity that we believe it is feasible to implement them in a future avionics operating system. The quality of the results of our limited number of experiments was excellent, in all cases equaling or improving on the hand-optimized original solutions.

Adaptive Software Technology Demonstration

The ASTD project is analytically examining the relationships between avionic application software semantic performance and the computer processing resources applied to respective applications. It is well known that certain application performance metrics are related to the amount of processing resources applied. Additionally, many emerging applications are expected to have highly dynamic resource utilization characteristics. However, currently fielded scheduling mechanisms only have the capability to deal with static demand cases. The combination can lead to resource utilization inefficiency and gross over design of physical capacity. Understanding this relationship is of particular concern to avionic system developers because of premium that must be paid to add computational resources in the constrained environment of a combat aircraft. Having a detailed characterization of application resource utilization and means for dynamic on-line management will enable higher performing embedded avionics systems and improve overall

on-board physical resource utilization and system affordability.

Avionics application software uses several different process dispatching paradigms. These may be thought of as existing at various levels of priority within the process dispatching system. Figure 4 illustrates these layers. Interrupts and event driven and data flow driven processing execute at priorities above the sampled data system (timeshare) processing. The timeshare processing is the fixed rate driven processing that executes with hard real time deadlines. These deadlines are typically 1/rate, so that the deadline for a 50 Hz rate would be 1/50 or 0.020 seconds.

The research addresses the problem of process dispatching in the layer below timeshare. This layer is the "continuous background" layer of Figure 4. Each processor may have multiple processes in this layer. All of these processes should run concurrently, and hence must be "timesliced" onto the single processing CPU available on the processing node. A simple approach might be to allocate the background processing resources (all the CPU left when higher priority is complete) to all background processes equally. While this approach might be acceptable on a general purpose workstation, such an approach to avionic applications would most likely result in a significant amount of processing being wasted on some processes while others produced output so infrequently that it is not necessarily useful.

Statement of the Problem

Basic assumptions:

1. Because each process is "continuous", it is referred to as iterative. At some point in its processing, it begins an iteration, and at some point, that iteration is complete. The processing
2. Each of these processes requires an amount of processing per iteration that is different from the others.
3. An individual process may require an amount of processing that varies significantly from iteration to iteration based on external conditions (number of targets tracked, etc.).
4. For each process, there is a maximum rate R_{max} above which further iterations either do not produce significantly improved results, or are actually counterproductive.
5. For each process, there is a minimum rate R_{min} below which performance deteriorates below levels promised for system performance. This does not mean that the process results are totally useless below this number. If that were so, then a hard deadline would be implied and the entire process should be considered for timeshare (hard deadline) processing.
6. In between R_{min} and R_{max} , the performance (quality of the resulting output) of the process improves as the iteration rate is increased. Processing resources are not wasted in moving the R value of a process from its minimum toward its maximum. While the performance improvement may not be linear with respect to an increase in iteration rate, we assume that it is non-decreasing.
7. There is a process with infinitely low priority which absorbs CPU time when all background processing process are executing at their individual R_{max} rates.

Figure 4. Execution Paradigm Layers

8. The processing time allocated to each background process should be as spread out over the total available time as practical. We know that purely greedy time allocation approaches may not necessarily take this into account. However, a "paced" approach for time allocation is of great importance in avionic systems. This is due to the fact that

the avionics systems as a whole is functioning on a sampled data basis. This includes background process processing as well, since background processes generally exchange data with timeshare processes. New data are presented to the system via external I/O message arrivals, which occur at various periodic rates. This I/O data may be processed by timeshare processes, but in any case is eventually made available to the background process. Each background process typically performs a sample of this data at the beginning of each iteration. Two back-to-back iterations on exactly the same input data do not contribute to system accuracy, and is a needless waste of processing resources. For example, suppose a background process is able to get enough processor time to perform two iterations per second on input data actually changing 6 times per second. Allocating all the processing time in one contiguous lump would most likely result in processing the same sample twice, while ignoring the next five. A more spread allocation might result in the use of sample 1 and 3, or 1 and 4. This would generally result in more desirable overall system behavior. (Processing of guaranteed specific samples would require the use of a timeshare data capture process.). As a result of the nature of avionic system processing and typical scheduling algorithm properties, we introduce the concept of an answer period and iteration (pacing) rate to further clarify the requirements for a variable background scheduling approach.

Figure 5 shows the concepts stated above graphically. Each background process should be given the processor execution time necessary to keep its execution frequency within its designated range. A process executing faster than its R_{max}

Figure 5. Desired Execution Range

indicates the processor time is being wasted if all other processes are not at least at their own R_{max} . A process executing below its R_{min} indicates deteriorated performance. This should be allowed only if all other background processes are already at their own R_{min} .

Specifically:

1. No process should be allocated time such that its execution rate is above its R_{max} .
2. When all processes are operating within their respective ranges, surplus time is allocated to the process with the highest priority, but not to the extent that it violates (1) above.
3. No process is allowed to have its rate fall below its R_{min} unless all other processes are at their own R_{min} .
4. When there is insufficient processing resources to maintain all processes at their R_{min} rate, the process with the lowest priority is denied processing resources first. When the lowest priority process is receiving zero processing, then the next lowest priority process is denied processing resources, etc..

In addition to the above, project design goals add:

5. Background processes may be allocated to different processors on each power cycle of the system based on the detected configuration of the avionics systems as a whole.
6. The computing nodes of the avionics system may not have uniform processing capabilities. New and old hardware modules may be installed in the same aircraft, with old modules being less capable than new ones. A background process should not be restricted to a specific hardware type unless that type module is the only type that allows for its execution at all.
7. A background process should not require code changes unless the functionality of that specific process requires changes. That is,

changes to one process should not require changes to another solely for the purpose of process resource management. This is an extension of the basic "capability modularity" requirement of the software in general.

Dynamic Resource Management In Avionics Systems

Introduction

Dynamic resource management is a requirement for certain types of hard real-time systems. Such systems are characterized by the fact that they have a minimal set of resource requirements in order to fulfill their mission. Some of the system tasks are able to provide better results within the time constraints if additional resources are allocated to them. Dynamic resource management is responsible for determining how spare resources, i.e., resources available after the minimum service requirements are met, should be allocated to achieve the highest overall system benefit and for managing their allocation at runtime.

Avionics systems, a class of embedded hard real-time systems, are increasingly migrating from federated hardware configurations to integrated computing environments. In federated hardware configurations real-time processing of different subsystems is performed on dedicated processors that communicate via a bus or network connection. In an integrated computing environment a set of computing resources (processors and memory) are available to perform a collection of real-time tasks.

As more computing power is available for hard real-time applications, the capabilities of these applications are continuously improved. In addition to basic hard real-time processing to handle input from sensors and provide output to actuators (sampled data stream processing or SDSP), intelligent processing of the data provides value-added application services (continuous background processing). Examples of continuous application processing are identification and tracking of objects from a radar signal data stream.

The computing demands of SDSP tasks are largely driven by the rates and bandwidth of the sensors and actuators being serviced. Since they are physical devices they are known and do not change dynamically other than in form of known

operational modes. For these situations scheduling analysis techniques such as GRMA can determine the schedulability of systems and provide runtime support for appropriate utilization of computing resources.

Continuous background processing tasks have different characteristics. Similar to SDSP tasks they are periodic tasks, though typically operating at a lower rate. While SDSP tasks are critical to the operation of the system, continuous background tasks show more flexibility in their resource demands. Typically, such tasks require a minimum amount of processing to produce any useful results. They also have a practical maximum processing requirement above which the incremental benefit is negligible. If minimum resources are not available it may even be acceptable for some tasks not to produce results. Examples of variability are increasing precision in the output of a task when produced with increased resources, and changing resource demands due to change in the observed environment such as change in number of objects to be observed.

Fault tolerance technology such as INSERT[4] in support of dependable upgrade adds a dimension of variability. Replication due to fault tolerance adds to the resource demands. Different degrees of fault tolerance can be achieved in Simplex by adjusting the number of variants active at any one time. This provides the opportunity to trade fault tolerance against other benefit functions.

Since the continuous background layer of the application system is the primary source of value-added services it is most frequently affected by upgrades. Existing continuous background services may be improved and new services may be introduced. In order to accommodate such upgrades the computing demands of the existing must be reduced to make available resources for the upgrade. This presents an interesting optimization problem in that use of available resources is to be maximized in order to achieve a high degree of benefit. At the same time resource utilization must be kept flexible in order to accommodate future upgrades. In other words, it is desirable to maximize spare capacity with minimal impact on the benefit of existing services.

Model

Avionics systems are typically architected such that interrupt handling takes highest priority. The next priority layer is assigned to SDSP tasks. Scheduling analysis such as GRMA can determine their schedulability and the remaining available resources, i.e., spare capacity or slack. Spare capacity indicated the amount of resources that can be allocated to lower level priority tasks. Slack scheduling determines how much additional resource capacity can be allocated at each priority level while maintaining schedulability. This capacity is available to continuous background tasks to meet their minimum service requirements. Continuous background tasks are generally viewed as less critical than SDSP tasks, i.e., continuous background resource requirements must be satisfied after SDSP resource requirements. Again scheduling analysis can determine whether the continuous minimum resource requirements can be met and what the remaining spare capacity is. This remaining spare capacity (plus any resources that are allocated to tasks but not fully utilized) are available to continuous background tasks to improve the overall system benefit.

To achieve DRM's objective of highest overall system benefit or quality of service (QoS), we can characterize each task with a benefit function. Some tasks may be interdependent in that the benefit function of one task is affected by resource allocation to another task. The collection of benefit functions determine the policies to be used by DRM to allocated spare resource capacity to achieve highest incremental benefit.

Let us characterize avionics continuous background tasks more formally in terms of scheduling properties. We will initially focus on describing tasks that produce results with increasing precision as more compute time is expended. Such tasks are periodic, i.e., they are expected to produce an answer periodically. This answer period T_j of task j is assumed to be known and constant. The processing within one answer period is modeled as execution of a number of iterations R_j . The execution time of one iteration, A_j , is known and constant. A task may have a lower (R_{jMIN}) and upper (R_{jMAX}) bounds on outside which allocation of resources has no significant benefit. Execution time C_j of task j is determined by $R_j * A_j$.

Within one answer period a task can operate on input in one of two ways:

1. A task reads input data at the beginning of the answer period and iterates over it until the end of the answer period. In this case we have the option to allocate the necessary resources in *lump-sum* - as long as the allocation maintains schedulability, i.e., dispatch the task once every answer period.
2. A task reads input data at every iteration to reflect any changes in the input during the answer period. In this case it is desirable to *pace* the execution of iterations within an answer period, i.e., to spread the execution of iterations as evenly as possible cross the answer period. An iteration can be viewed as a "quasi-periodic" task, and be dispatched at the appropriate rate.

Different continuous background tasks may have different answer periods, iteration execution times, and iteration limits.

When all continuous background tasks operate at R_{MIN} , spare capacity can be used to improve the system performance by increasing R_j to R_{MAX} . Various policies can be envisioned for allocating the spare capacity. A simple allocation policy is to specify a semantic priority order, i.e., to increase continuous background tasks to R_{MAX} one at a time in a specified priority order. Other options include proportional allocation of spare capacity across all continuous background tasks, i.e., increase their rates proportionally to their R_{MAX} , or to distribute spare capacity according to benefit functions associated with continuous background tasks. Similarly, policies can be envisioned for handling situations when R_{MIN} cannot be reached by all continuous background tasks. In this case, continuous background tasks may refrain from executing in an answer period in a reverse priority order. Notice, that these semantic priorities are in tension with the priorities assigned to tasks according to their scheduling paradigm, e.g., rate monotonic assignment of priorities.

The above iteration model for characterizing continuous background tasks can be refined to model more than one dimension of variability. Execution time of a task may not only be

determined by the desired precision, i.e., number of iterations, but also by the number of objects to be processed during an answer period. The execution time per iteration may be characterized as a function of the number of objects to be processed. If this function is linear, we can define an execution time per object and iteration, B_j , which is assumed to be known and fixed. O_j indicates the number of objects, which may vary over time. The execution time for one iteration is determined as $A_j = B_j * O_j$. There may be a minimum and maximum number of object that should be tracked. In this context, one challenge is to appropriately trade tracking a number of objects with the precision at which they are tracked.

Q-RAM

The QoS-based Resource Allocation Model (Q-RAM) presents an analytical approach for satisfying multiple quality-of-service dimensions in a resource-constrained environment. Using this model, available system resources can be apportioned across multiple continuous background tasks such that the net utility that accrues to the end-users of those applications is maximized.

We now provide a brief overview of Q-RAM referring the reader to Rajkumar[7] for more details. The goal of Q-RAM is to address two problems:

1. Satisfy the simultaneous requirements of multiple applications along multiple QoS dimensions such as timeliness, cryptography, data quality and reliable packet delivery, and
2. Allow applications access to multiple resources such as CPU, disk bandwidth, network bandwidth, memory, etc. simultaneously.

Q-RAM uses a dynamic and adaptive application framework where each application requires a certain minimum resource allocation to perform acceptably. An application may also improve its performance with larger resource allocations. This improvement in performance is measured by a utility function.

Q-RAM considers a system in which multiple applications, each with its own set of requirements along multiple QoS dimensions, are contending for resources.

- Each application may have a minimum and/or a maximum need along each QoS dimension (such as timeliness, security, data quality, dependability, etc).
- An application may require access to multiple resource types such as CPU, disk bandwidth, network bandwidth, memory, etc.
- Each resource allocation adds some utility to the application and the system, with utility monotonically increasing with resource allocation.
- System resources are limited so that the maximal demands of all applications often cannot be satisfied simultaneously.

Q-RAM algorithm

With the Q-RAM specifications, a resource allocation decision will be made for each application such that an overall system-level objective (called utility) is maximized. Q-RAM is defined as follows. The system consists of n applications $\tau_1, \tau_2, \dots, \tau_n$, $n \geq 1$, and m resources with finite capacities, $R_1, R_2, \dots, R_m$, $m \geq 1$, which can be shared, either temporally or spatially. CPU and network bandwidth, for example, would be time-shared resources, while memory would be a spatially shared resource. Let the portion of resource R_j allocated to application τ_i be denoted by R_{ij} . The sum of R_{ij} over i must be less than or equal to R_j .

The application utility, U_i , of an application τ_i is defined to be the value that is accrued by the system when τ_i is allocated $R_{i1}, R_{i2}, \dots, R_{im}$. U_i is referred to as the utility function of τ_i . This utility function defines a surface along which the application can operate based on the resources allocated to it.

Each application τ_i has a relative importance specified by a weight w_i . The *total system utility* is defined to be the sum of the weighted sum of the application utilities. Each application τ_i needs to satisfy requirements along d QoS dimensions $Q_1, Q_2, \dots, Q_d$, $d \geq 1$. Each application is said to be feasible if it is allocated a minimum set of resources for every QoS dimension.

Given convex utility functions, the basic Q-RAM algorithm computes an optimization trajectory through utilization space by ensuring that the derivatives of the utility functions are equal along the trajectory. If the derivatives are not equal at the minimum acceptable level of utilization, the utilization of the task with the greater derivative is increased until either the resource is consumed or its derivative becomes equal to the derivative of another task.

ASTD Conclusions

This initial ASTD experiment had several goals:

1. To identify the categories of information required for designing an adaptive software application.
2. To demonstrate that this information can be collected and to identify and document useful techniques for collecting the information.
3. To develop and demonstrate a design notation for summarizing key portions of the information collected.
4. To scout out a resource-intensive application domain which can be used in later ASTD experiments.

The specific lessons learned are beyond the scope of this document. (The reader is directed to contact Mr. Kenneth Littlejohn at AFRL/IFTA for a copy of the report.) However some generalized statements can capture the essence of the results.

There is no generic solution to building adaptive software applications. At the application level, adaptive software technology amounts to a problem solving perspective to use in approaching requirements analysis and software design. Each application will need to be made adaptive in a way that is consistent with the semantics of the problem being solved. We do believe application-independent solutions are possible for the adaptive resource manager.

Adaptive Software Technology is not a coding technique. Rather, it is a philosophy to be followed in the algorithm selection process, in requirements specification, in software design, and in system

validation. Unless there are adaptive requirements and system validation criteria, there can be no adaptive software.

There are economic factors to be considered in the use of adaptive software technology. Adaptive software is software that can operate in more than one mode and each such mode may require additional effort to develop, test, and maintain. These costs must be weighed against the benefit of improved utilization of hardware resources. Rapid advances in computer hardware tend to raise the break-even point at which adaptive software technology becomes worthwhile. New requirements for resource-intensive software functionality tends to lower it. The balance of these pressures will determine the future benefit of adaptive software technology.

Conclusions

The basis underpinning of these middleware services are built upon POSIX compliant RTOS's using POSIX message queues in a publisher-subscriber mode of data push-pull and dynamic binding of applications. As previously mentioned the basic middleware capability of ASEP has transitioned to two of LM Aero's product lines.

The overall focus of these numerous research projects has been to extend the ASEP and provide scalable, reliable, resource optimal techniques that can be applied to embedded avionics applications. It is not the authors' intentions to imply that all problems have been addressed and that these concepts are fully developed and ready for deployment on avionics systems. However, successful proofs of concept have been achieved which forms a foundation of risk reduction for those programs going into E&MD that could benefit from the need for reliable upgrade and fault tolerance, dynamic reconfiguration of applications across a distributed processing core, and optimization of the family of applications that can benefit from the Q-RAM techniques outlined. Each of these "middleware" components can be omitted, added, or tuned as needed by the avionics architect.

References

- [1] Liu, C. L., and Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, *JACM* 20 (1) (1973), 46-61.
- [2] Sha, L., and Goodenough, J. B., "Real-Time Scheduling Theory and Ada", Technical Report CMU/SEI-89-TR-14, Software Engineering Institute (1989).
- [3] Mark H. Klein, John P. Lehoczky, Ragunathan Rajkumar: Rate-Monotonic Analysis for Real-Time Industrial Computing. *IEEE Computer* 27(1): 24-33 (1994) 24-33.
- [4] Calloni, Ben A., et. al., "INSERT: A COTS Based Solution for Building High Assurance Applications", *18th Digital Avionics Systems Conference*, St. Louis, MO, 24-29 October 1999.
- [5] Preston, J. D., and Kegley, R. B., "Designing for Determinism: Lessons Learned from Modern Real-Time Avionics Applications," *Proceedings 17th Digital Avionics Systems Conference* (1997).
- [6] Kegley, R. B., Preston, J. D., and Dang, E. T., "Dynamic Avionics Real-Time Scheduling Final Report", Air Force Research Labs, Information Directorate, Doc # WL-TR-97-1200, 27 Oct 97.
- [7] Rajkumar, Raj, Chen Lee, John Lehoczky and Dan Siewiorek, "A Resource Allocation Model for QoS Management", *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997
- [8] Lui Sha, Shirish S. Sathaye: A Systematic Approach to Design Distributed Real-Time Systems. *IEEE Computer* 26(9): 68-78 (1993)