An Approach for Supporting Temporal Partitioning and Software Reuse in Integrated Modular Avionics

Mohamed Younis Mohamed Aboutabl

Honeywell International Inc.
Advanced Systems Technology Group
9140 Old Annapolis Rd / MD 108
Columbia, MD 21045
Mohame.Younis@honeywell.com

Daeyoung Kim

Dept. of Computer and Info. Science and Eng
University of Florida
Gainesville, FL 32611-6120
dkim@cise.ufl.edu

Abstract

The Integrated Modular Avionics (IMA) approach can achieve lower overall hardware costs and reduced level of spares by getting multiple applications that have traditionally been implemented using separate, federated units to share hardware resources. However, the IMA approach also brings new challenges that did not exist in the federated setup. Avoiding unwanted dependencies among applications and managing the reuse of legacy code tops the list of challenges. This paper describes a two-layer software architecture, which enables the integration of multiple applications while maintaining strong spatial and temporal partitioning among application modules. In addition, the architecture allows the reusability of existent software modules by enabling the integration of applications written for different real-time operating systems.

1. Introduction

The recent advances in computer technology encouraged the avionics industry to take advantage of the increased processing and communication power and combine multiple federated applications into a shared platform. A new concept, called the Integrated Modular Avionics (IMA) [1], was developed for integrating multiple software components into a shared computing environment powerful enough to meet the computing demands of these traditionally separated components. This integration has the advantage of lower hardware costs and reduced level of spares. Further reductions in weight and power consumption are also expected. To capture the numerous advantages of the IMA approach, there are currently major market forces pushing the industry in this direction.

The Integrated Hazard Avoidance System (IHAS) and the Integrated Environmental Control System (IECS) are examples of IMA projects at Honeywell. The IHAS system integrates flight safety avionics such as Traffic Collision Avoidance System (TCAS), Enhanced Ground Proximity Warning System (EGPWS) and Weather Radar. The IECS system controls the operating environment to ensure safe use of the equipment on the aircraft and the comfort of passengers. For example, the IECS system adjusts (by cooling or heating) the operating temperature of hydraulic, electrical and mechanical power devices and equipment, de-ices and defogs windshield and controls cabin pressure and passengers' air-condition. The IHAS and IECS systems achieve substantial reduction in the very expensive flight-worthy hardware, in the weight and volume of the avionics and in the power consumption. Such reduction lowers the development cost of avionics and increases the efficiency of aircraft operation.

However, the IMA approach also brings new issues. Chief among these issues is the problem of avoiding unwanted dependencies among applications. It is necessary to be able to show, with a very high level of assurance, that a problem or failure in one application cannot have an adverse impact on any other application. Without such level of assurance the aircraft certification authorities (e.g. the Federal Aviation Administration) will be unwilling to certify the installation of such systems on an aircraft. Such a requirement is usually referred to as strong partitioning among the integrated applications.

Supporting the reuse of legacy applications is another very important issue that the IMA approach raises. Requiring the redevelopment and revalidation of existing application software can introduce excessive costs that diminish the advantage of the IMA approach. Statistics show that the cost of software re-development for safety critical systems is prohibitive [2]. Software reuse provides economical values and accelerates time to market [3].

Several efforts have been made to define a standard operating environment for integrated modular avionics applications [6,7]. This paper describes a new approach for implementing a software architecture that enables software reuse while complying with IMA specifications. Reusing

software includes legacy applications and third party modules for which the source code is not accessible.

The following subsections describe the partitioning issues and different approaches for supporting software reuse in an IMA environment. The fault model and system model assumed throughout the paper is also discussed. The remainder of the paper is organized as follows: Section 2 describes the functional components of the proposed software architecture. Enforcing strict time-based scheduling and handling of exceptions is discussed in section 3. A prototype implementation of the architecture is described in section 4. A survey of related work is given in section 5. The paper is concluded with a summary and discussion of future directions in section 6.

1.1 Strong Software Partitioning

Strong partitioning conceptually means that the boundaries among applications are well defined and protected so that operations of an application module will not be disrupted nor corrupted by erroneous behavior of another application [8]. Containing the effects of faults is very crucial for the integrated environment to guarantee that a faulty component may not cause other components to fail and risk generating a total system failure. For instance, in an ideal IMA-based avionics, a failure in the cabin's temperature control system must not negatively influence critical flight control systems such as the flight manager.

In federated avionics setup, applications share very little with each other and partitioning comes naturally, yet very costly due to the excessive use of computing resources. In an IMA environment, an application might share a resource with others and thus, its correct operation becomes dependent on their use of the resource. When multiple avionics application software coexist on the same computer, partitioning is particularly challenged in the way applications access memory, seize and release the CPU and interface with input and output devices. Usually applications are allocated different memory regions while CPU and I/O device access time are divided among them based on a schedule, commonly generated in avionics before the integration of these applications.

Although dividing memory and resource access time among applications forms enough boundaries and facilitates the integration, it cannot guarantee that those boundaries will not be violated under some faulty conditions. Therefore, the IMA environment needs to ensure strong partitioning among the integrated applications both spatially and temporally. The address space of each application should be protected against unauthorized access by other applications. In addition, an application should not be allowed to overrun its allocated quota of resource usage and delay the progress of other integrated applications. This paper focuses only on partitioning both the CPU and Memory and provides very brief insight for I/O handling.

1.2 Reusing Legacy Avionics Software

In recent years, the software industry has started to realize the cost of developing high quality software. Many design methodologies and techniques are proposed to achieve software reusability at different levels of the software development [3]. In avionics applications, the incentives for software reuse are very attractive since developing and certifying software, according to the DO-178B guidelines [4] is both time-consuming and expensive. However most legacy applications were developed without consideration of future reuse of modules, nor using standard interface such as POSIX [5] to operating system services. That leaves only reusing the entire application software as an option.

The same also applies to third party's applications that could potentially be integrated in the IMA environment and for which the source code are not accessible for competition's concerns. In addition, it is very hard to get all avionics vendors and aircraft integrators to agree on a common architecture. Therefore, it is very common that vendors select different software environments for their products. To capture the full potential of cost saving using the IMA approach the software architecture should allow the integration of applications, developed using a different software environment, without imposing change to neither the code nor the operating system interface.

1.3 Fault Model

Strong partitioning implies that any erroneous behavior of a faulty application partition should not affect other healthy applications. This erroneous behavior of an application can be the result of a software fault or a failure in a hardware device used exclusively by that application. The fault can be generic, accidental or intentional in nature and permanent, transient or intermittent in duration. However, we assume that there are application-specific semantic checks, which verify the validity of the communicated data to detect errors due semantic-related generic faults in the application software. We also assume that the system is not liable to Byzantine faults, i.e. all faults manifest themselves into errors, which are detected in the same way by all the other healthy modules. In addition, we assume that faults occur one at a time with no simultaneity.

An attempt by a faulty component to corrupt other healthy system components should lead to an error. Only applications that communicate with that faulty application partition need to be aware of the error and perform recovery according to the nature of the application. On the other hand, operations on healthy applications that do not communicate with the faulty application will not be affected.

1.4 System Model

In our environment, we assume that the CPU will not receive any interrupts from I/O devices. The I/O device should either operate on a polling-basis or be supported by

a device controller that is included in the design to handshake with the device and buffer the data. In safety-critical real-time applications, such as avionics, frequent interrupts generated by I/O devices to the CPU risk the system predictability and complicate system validation and certification. In addition, the use of device controller or I/O co-processor is very common on modern computer architectures to off-load the CPU and boost the performance.

We also assume that the CPU either supports memory-mapped I/O or provides mechanism to enable partition-level access protection for IO-mapped devices. In all cases, access to I/O devices should not require the use of privileged instructions. In recent years, support of memory-mapped I/O devices has become almost standard on microprocessors. For example, the Motorola PowerPC processor supports memory-mapped devices only. Using the memory management unit, access to a memory-mapped device can be controlled by restricting the address space of an application. An application can access the device using regular memory access instructions if the device address is in its address space. On the other hand, the Intel Pentium processor supports both memory-mapped and IO-mapped devices. However, the I/O instructions of the Pentium processor are privileged. Thus only memory-mapped devices are allowed if the Pentium processor is to be used in our environment.

The underlying hardware is expected to provide certain services that are essential for the implementation of the software architecture. We assume the availability of a memory protection mechanism to define memory partitions and control the read, write, and/or execute access to these memory partitions. A high-resolution real-time clock and at least one countdown timer are required. The countdown timer should be able to generate hardware interrupts. It is worth noting that such features are almost standard on commercially available computer boards serving the embedded applications industry.

2. Two-Layer Software Architecture

In this section, we present new software architecture for integrating real-time safety-critical avionics applications. The architecture, depicted in figure 1, is fundamentally a two-layer operating environment that complies with the ARINC specification 653 [6] and the Minimum Operational Performance Standards for Avionics Computer Resource [7]. However, our approach goes one step further by enabling the integration of legacy software modules together with their choice of real-time operating systems in one CPU board. The following describes the functional components of the architecture.

System Executive: The bottom layer of the architecture, termed the System Executive (SE), provides each application module with a virtual machine [23], i.e. a protected partition, inside which the application can execute together with its choice of real-time operating system. This way, the application is isolated from other applications in the space domain (i.e. memory address space).

Generally, spatial partitioning can be enforced via hardware [9] or software means [10,11]. Ensuring the integrity of memory access via software means, typically through the insertion of checks in the code, can complicate the integration and the certification of the application. In order to insert such checks in the application's source code, if available, the application will have to be revalidated and re-certified. On the other hand, using a tool, such as [11] to automatically insert memory access checks in the binary image is intrusive to the validated temporal behavior of the avionics application. Therefore, we rely on hardware means such as the memory management unit available with most modern processors to enforce spatial partitioning.

Time-domain isolation is accomplished by sharing the CPU and other resources among applications based on a pre-computed static timetable. The system executive maintains a real-time clock to strictly implement the timetable in which each application is assigned well-defined time slices. Providing time services to the applications is quite challenging since applications used to run exclusively on a computer and had a continuous time notion. In the new architecture, the application will be preempted and internal clock (timekeeper) will be suspended until the application seizes the CPU again. We have developed an algorithm that provides the application with time services while enforcing temporal partitioning among the integrated applications. Details of the time management are described in section 3.

In addition to ensuring spatial and temporal partitioning, the SE handles context switching, facilitates inter-application / inter-processor communication, and initializes / monitors / terminates application partitions. The details of the inter-application communication mechanisms are not discussed in this paper.

It should be noted at this point that only the SE would have the ability to execute in the highest privileged CPU mode. All other partitions execute in a less privileged CPU mode thus ruling out the possibility of an application corrupting the memory protection set up or violating other applications' rights to use the CPU. Any function typically implemented by an operating system that requires a privileged CPU mode of operation, thus interfering with the responsibilities of the SE, is replaced by a call to the Interface Library. The Interface Library redirects such calls to the SE.

The SE is the main loop that controls the operation of the entire operation. It starts by initializing all hardware resources, and setting up the memory partitions. It then enters an indefinite loop in which it busy-waits for the starting time of the next partition, as indicated by the scheduling timetable, before it switches control to that partition. When the time slice of the selected partitions

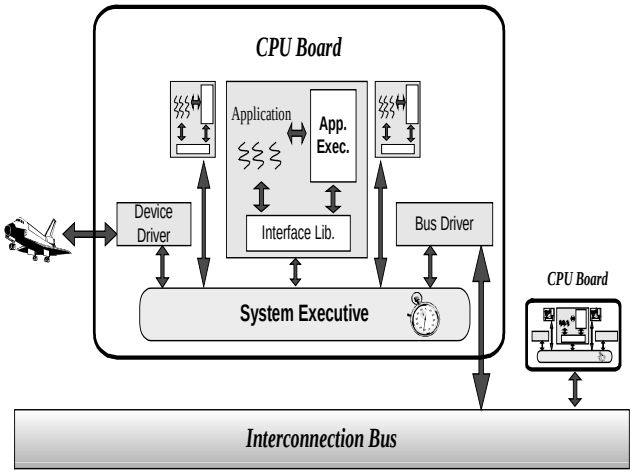

Figure 1: New avionics software architecture

expires, control of the CPU returns to the SE, and the loop continues. Before dispatching a partition, the CPU is downgraded to the less-privileged User mode of operation. The CPU is upgraded to the privileged Supervisor mode at the end of the partition's time slice.

Application Executive: Each application, which may consist of multiple tasks, is assigned a protected memory partition, thus preventing a fault in one application partition to negatively effect (propagate to) other applications. The SE will be able to support applications written for different operating systems. To accomplish this feature, each application is accompanied by its own Application Executive (AE) as well as an Interface Library (IL) to the system executive's services. The application executive, which represents the top layer of the software architecture, is actually a customized version of the operating system of choice. The customization ensures that specific options of the real-time operating system, acting as a partition executive, are included (excluded or replaced) in order for the application to run in non-privileged mode. The AE handles intra-application communication and synchronization. The AE also manages the dynamic memory requirements of the application within the boundaries of the application's own memory partition. The AE may also implement its own strategy for scheduling the application's tasks. All the application executive's functions related to inter-application and inter-processor communications are handled through the Interface Library to the SE.

While application loading is performed by the SE, all application-specific initialization is done by the AE. The AE will not perform any boot sequence procedures such as probing for or initialization of hardware devices, initializing interrupt tables, setting up the memory management unit (MMU) registers, etc. Such procedures are replaced by the proper initialization of the corresponding data structures.

Interface Library: Since operating systems in general assume privileged access to the hardware, the system executive needs to export services to application executives that emulate privileged instructions [23]. These services include exception handling, interrupt enabling and disabling and access to processor internal state e.g. during thread context switching. The Interface Library (IL) encapsulates these services. The application is linked with the IL instead of some of the original operating system's libraries. A different version of the IL is built for each supported operating system. The IL acts as a gateway between the application and the computer's hardware services, which are now provided by the SE instead of the application's native operating system. Section 3 includes detailed discussion of the IL

Device drivers: Device handling, according to our system model, can be performed by either the SE or the AE. Handling I/O devices within the SE will require the implementation of synchronization mechanisms to maintain correct order of operations among the applications and thus complicate the design of the SE. Maintaining the simplicity of the SE is a design goal in order to facilitate the SE certification. In addition including device handling within the SE increases its the sensitivity to device changes and might mandate the re-certification of the SE when a new device is added or removed. On the other hand, application executives cannot handle shared I/O devices without coordination among themselves.

Our approach is to allow the AE to handle I/O devices that are exclusively used by that application (partition). The AE synchronization primitives can be used to manage access to a device made by tasks within the partition. The SE will ensure that every device in the system is mapped to one and only one partition. In order to support a shared device among partitions such as a backplane data bus, a device daemon (handler) will be allocated to a dedicated partition. The device daemon serves access requests to that device made by the application partitions. The shared device manager partition still has exclusive access to the device. Application partitions that need read or write access to a shared device communicate with the device daemon via inter-partition communication primitives. This paper focuses only on partitioning both the CPU and Memory. I/O device handling is not discussed any further.

2.1 Architecture's Effectiveness for IMA

The presented two-layer architecture supports robust partitioning, allows reuse of legacy software and facilitates system validation. The architecture enforces strict spatial boundaries and CPU time quota among the integrated applications. Ensuring spatial and temporal partitioning guarantees that the system will continue to operate safely under the presence of software faults and failures in hardware devices, which are exclusively used by an individual application. Since each application cannot

exceed its allocated time and cannot write to any memory region outside its permissible space, the effect of a software fault in that application will be limited to the application itself and would not propagate to other partitions.

The architecture increases reusability of existent application modules that may have been developed by multiple vendors for different real-time operating systems. The System Executive provides a virtual machine for applications to run. A set of library routines would need to be linked with the application in order to be executed in the integrated environment. The approach requires neither modifying application programs nor the availability of the source code, thus overcoming an important hurdle in reusing legacy application and integration of third-party software. Modifying legacy applications tends to be very costly due to the need for re-validation of previously certified development. Source code of third-party software is either radically expensive or simply inaccessible. The architecture simplifies the integration of such a diverse mix and minimizes the need to re-validate previously certified software.

In addition, the architecture reduces overall certification costs by maintaining design simplicity and eliminating the need to re-test a whole set of applications when only a single module is added, upgraded, or removed from the system. The System Executive is designed to be largely independent from the integrated application modules. Such design approach maintains the stability of the system executive and does not require the certification of the System Executive every time a new application module is integrated or an old module is upgraded. The architecture allows application modules to be developed separately and facilitates incremental integration. An application can be developed and tested using the Application Executive's environment before integration. By ensuring partitioning, the application would not be negatively impacted with any erroneous behavior of other integrated application in our environment.

Although the two-layer architecture introduces overhead and penalizes system performance, the flexibility and reduced development cost justifies such performance degradation. In addition, the recent published research about layered µ-kernel-based systems proved that the overhead is not radically high [12,13,22]. Experiments on the L4 Linux [13] systems showed an overhead of 5% to 10% compared to monolithic Linux.

In addition, in our architecture multiple copies of the same real-time operating system need to exist if multiple application partitions were developed using that particular RTOS. We do not see this as a major concern since the price of RAM has dramatically gone down in the recent years and the footprint of an embedded real-time operating system is often small. On the other hand, The alternative approach is to integrate tasks from multiple applications, which requires the revalidation of all of them, a job that is way more expensive than the increase in the required RAM size.

3. SE and AE Interface

The system executive's main functions are the management of partitions, serving privileged requests for hardware access and the handling of inter-partition communication, or IPC. The description of the IPC services is out of the scope of this paper. The focus here is to describe how the SE guarantees each partition its quota of the CPU time and maintain the consistency between their internal clock and the overall system clock. This is a difficult challenge since each partition has its own AE. In this section, we describe how the time slice of each partition is strictly enforced without adversely affecting the operation of the AE. In addition, algorithms for emulating privileged functions for application executives are also discussed.

3.1 Time Management

It is essential to keep track of the passage of time both at the system executive level as well as at the application executive level. In a stand-alone federated configuration, an application executive keeps track of the progress of time in units of clock ticks. A tick counter is incremented upon the receiving of an interrupt from a periodic timer indicating the passage of clock tick. In the integrated setup the AE will be preempted and will no longer regularly receive the timer interrupt to track the time and it has to rely on the SE to be the timekeeper. Therefore, the system executive needs to maintain a global time notion while successfully enforcing the time-based scheduling of partitions. Tracking system-wide global time is essential to provide the application with a reference clock so that, application dependent events could be tracked inside the partition.

There are two alternative approaches to maintain global time and enforce temporal partitioning. In the first approach, the SE may rely on an independently running high-resolution real-time clock built into the CPU or the board to keep track of global time while using a countdown timer to enforce the execution schedule of partitions. The second approach calls for using a periodic countdown timer to generate interrupts at equidistant points in time (SE-ticks) so that the SE updates the proper data structures representing the current time and schedule partitions. In the second approach, all time-dependent events may only occur at the start of one of these SE-ticks.

The size of the SE-tick will be constrained by the clock tick sizes of the application executives. In the absence of an independent real-time clock, the SE tick period is selected as the greatest common divisor of the clock tick periods of the different application executives that share the same CPU board. For example, if there are three application partitions with tick periods of 10, 13 and 20 milliseconds, the SE-tick has to be 1 millisecond.

Receiving interrupts at that much higher rate might cause thrashing since the CPU will end up spending most of the time serving timer interrupts. Given the right duration for a SE-tick the second approach can be used as well. Assuming the first approach, the algorithm for tracking time at the AE level is presented section 3.1.2.

3.1.1 Time-Based Dispatching. During the system integration phase, an off-line scheduler builds a static timetable indicating when, and for how long, each partition shall use the CPU. This timetable depends on the real-time constraints and resource requirements of each individual partition as well as on the resource availability. A partition may be allocated multiple disjoint time slices with possibly different duration each.

At run time, the SE dispatches and preempts partitions according to the timetable. First, the SE inspects the timetable to determine the next partition. Next, the SE sets up the necessary environment so that a countdown timer generates an interrupt at the end of the time slice of the selected partition. Finally, the SE switches context to the selected partition, which then starts executing.

The application executive always acts as the entry point of every application partition. The AE applies its own scheduling policy to determine which application task to dispatch. Once the partition's time slice expires the context-switching back to the system executive is immediately started. The current time is recorded and the current state of the partition is saved, to be used in restoring the application context when later scheduled.

The next time the partition is dispatched by the system executive, the partition's own clock is adjusted and the CPU is handed over to the application executive. The latter will determine which task to dispatch next. The time-based partition dispatching and preemption is illustrated in the control-flow diagram of figure 2. This figure also emphasizes that on re-entry, control of the CPU is handed over to the AE, even if a regular application task was interrupted by the most recent time-slice expiration event.

While a partition is executing, a timer will be set to interrupt every clock tick of the corresponding AE so that the application execution behavior would not change. Figure 3 demonstrates the fact that the countdown timer may be required to generate several interrupts before the expiration of the time slice of a partition. These interrupts help the application executive monitor the progress of time. This is discussed in more details in the next section.

3.1.2 The Timer Interrupt Service Routine. The timer interrupt-service-routine (ISR) is the main routine that enforces temporal partitioning, handles context switching and updates the AE internal time. The algorithm of the timer ISR presented in this section assumes the availability of a high-resolution independent real-time clock enabling the SE to keep track of the global time. Figure 4 presents a flowchart of the timer (ISR) and the partition dispatcher of

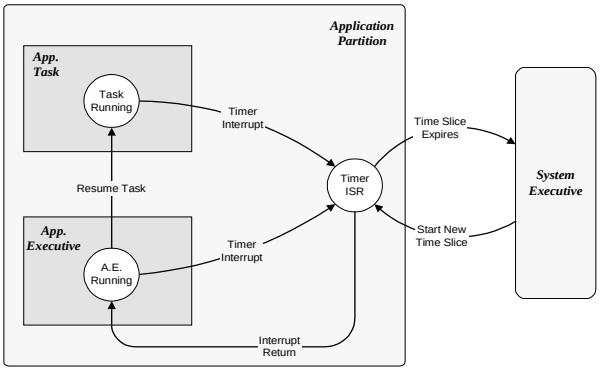

Figure 2: Control flow diagram of the partition dispatching process

the SE, side-by-side. The length of a time slice is not restricted by the clock tick of application executives.

In preparation to the dispatching of the next partition, the SE checks the allocated time slice and determines the number of full-length AE clock ticks 'Nticks', as well as any partial tick, 'parTick'. The SE calculates 'Nticks' and 'parTick' as functions of the selected partition's clock tick length and the allocated time slice. After waiting for the starting time, the SE sets up the countdown timer according to the computed values of 'Nticks' and 'parTick'.

As the partition executes the timer ISR periodically decrements 'Nticks', then invokes the application executive's appropriate routine to update the partition's current time. Once all full-length ticks have been exhausted, the countdown timer is programmed to generate one final interrupt after duration of 'parTick'. The ISR starts the context-switching procedure once the time slice expires. In order to simplify the implementation and reduce the overhead of the timer ISR, the time slice of each partition is forced to start at the boundary of the AE's clock tick.

Except for the very first time, the partition's re-entry point is always in the middle of the timer ISR. The ISR adjusts the AE's time data structures to account for the

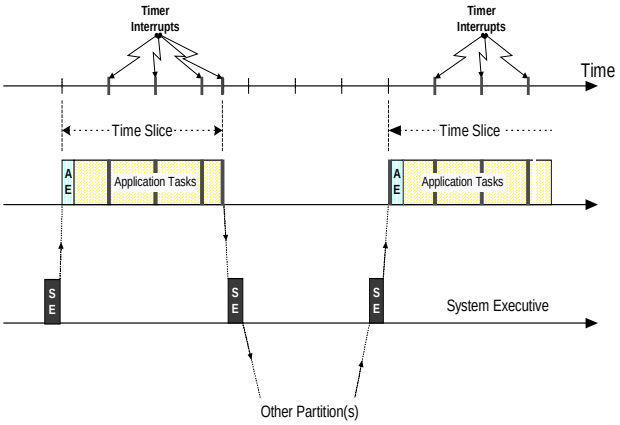

Figure 3: Time-based partition dispatching

elapsed time during which the partition was preempted, and then immediately returns to the AE. The reason behind having the AE start each time slice is to check for any partition-related events, e.g. arriving messages or watchdog timers, that may have occurred while the partition was preempted by the SE. Such events, if exist, may affect the scheduling decisions of the AE and make a different task eligible to acquire the CPU instead of the one that was interrupted by the expiration of the previous time slice. The time slice need not be a multiple of the AE's clock tick length.

Since global time continuously advances even when a partition is not running, the next time that partition is dispatched its tick counter is properly updated. This lump sum update, which is performed by ISR , should account for the time elapsed since the most recent time slice the partition has used the CPU. The delivery of possibly multiple clock ticks is not interrupted by the application executive's scheduler until all ticks, and messages, have been successfully delivered. It is required that the clock tick rate of all application executives must be known at the creation of the scheduling timetable. In addition, application executives are not permitted to change their clock tick rates at run time.

During the execution of a critical section, the AE usually disables interrupts to ensure that updating internal data structure is an atomic operation and thus maintains the consistency of the data. Interrupt disabling is a privileged operation that will be trapped, as we explain next, and will be replaced by setting an indicator 'AE_Interrupt_Disabled'. If the AE is preempted during the critical section, the data consistency will be ensured when execution resumes to where it stopped. Since the delivery of the clock tick(s) to the AE might result in awakening a higher priority task, a different thread could be scheduled. Therefore, the clock tick will be delivered only if the AE did not indicate that it wanted the interrupts disabled so that data consistency could be guaranteed. In such case, the delivery of the clock tick will be deferred until the AE indicates its readiness to accept interrupts. As shown in figure 4, deferred clock ticks are queued for future delivery.

3.2 Emulated Privileged Operations

Execution of privileged instruction while running in User mode generates an exception. The details of exception handling heavily depend on the hardware platform. However, the concepts remain almost unchanged across

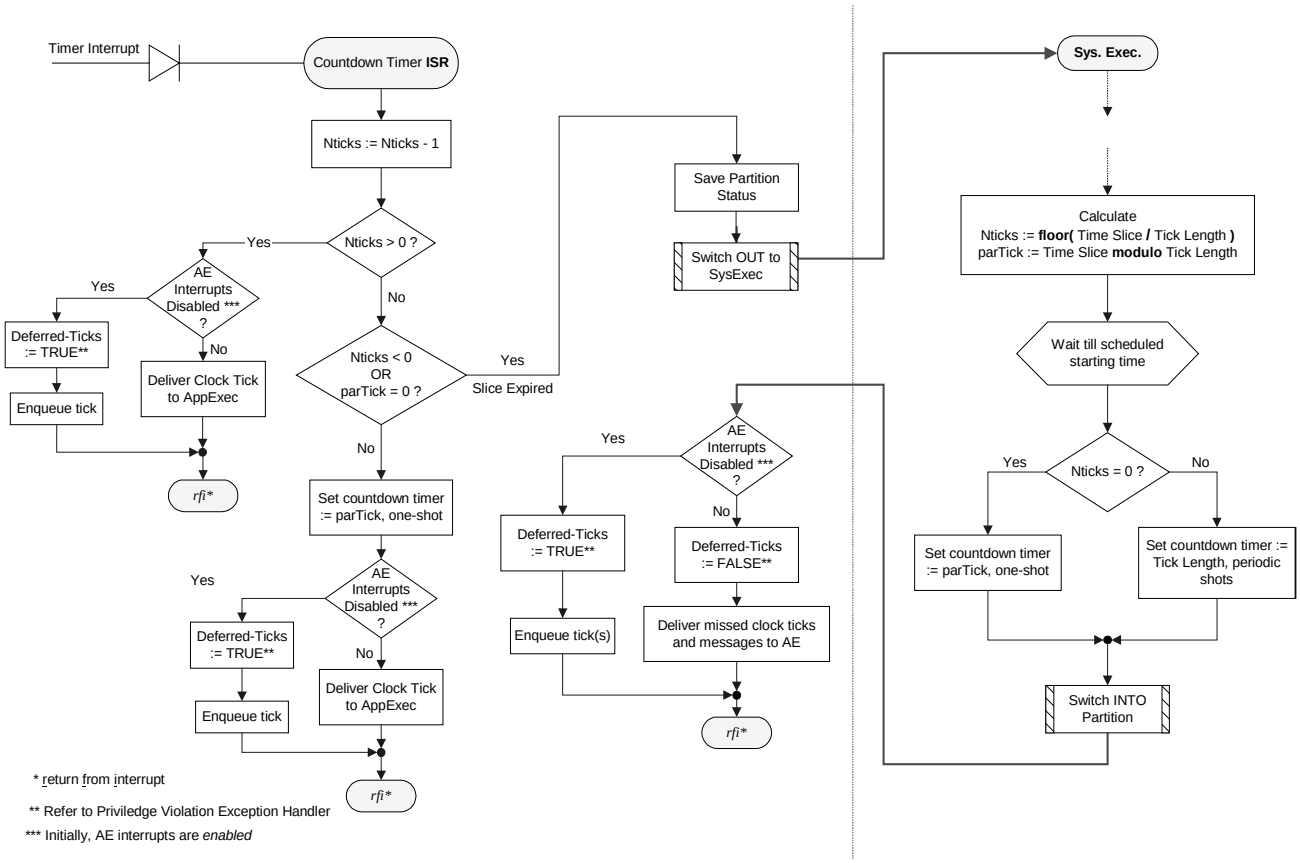

Figure 4: The timer ISR and partition dispatching in SysExec.

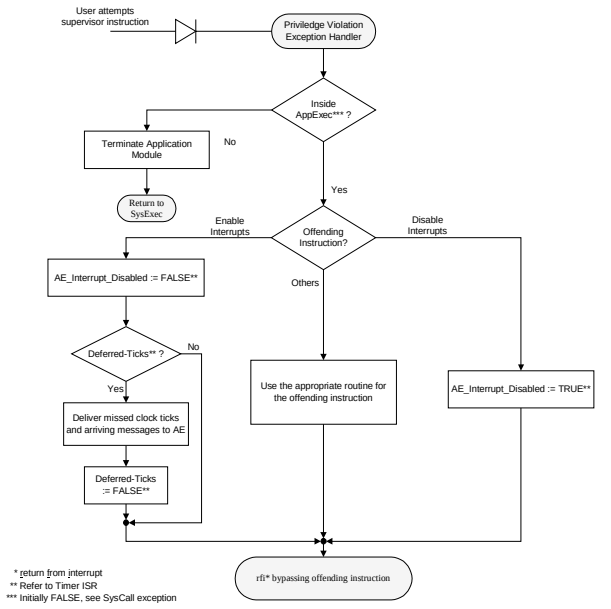

Figure 5: *Privilege Violation* exception handler

platforms. Some platforms, such as the Intel processor family, use the term "interrupt" to refer to external asynchronous events that are not related to the currently executed instruction, such as an asynchronous signal from a device. The term "exception" is reserved for events related to the currently executed instruction, such as "divide by zero" or "illegal opcode". Other platforms, such as the PowerPC, use the term "exceptions" to refer to both types of events. In the following discussion, we will follow the PowerPC approach for using the term "exception".

Fundamentally, all platforms provide mechanisms to recognize and respond to different kinds of exceptions. Any processor should be able to identify the source of the arriving exception, and may elect, under program control, to postpone handling of this exception.

When an exception occurs, the CPU jumps to a predetermined fixed location in memory where an Exception Handling Routine (EHR), or at least an entry point to one, resides. Before the first instruction in such EHR is started, the CPU is promoted to the supervisor mode. Such high privilege is required to properly handle the exception. The mode of operation that existed prior to taking the exception is restored at the end of the EHR.

Typically, an exception handler has three code segments; namely an entry segment, a body segment, and an exit segment. The entry and exit segments take care of saving and restoring the processor status upon entry and exit from the exception handler, respectively. The body segment provides the core function of responding to the event causing the exception. Exception handlers should be coded to allow nesting of multiple exceptions.

In our approach of a two-layer operating environment, it is essential that the system executive be the first to respond to any exception. However, in order to maintain application portability, the SE may invoke the procedures of the currently running AE to perform application-specific handling of some of the exceptions. Privilege violations and system calls exceptions are of particular interest, since they are being trapped in order to emulate privileged operations for the AE.

3.2.1 Privilege Violation Exception. Contemporary processors provide security mechanisms for the operating system by having at least two privilege levels; namely the User and Supervisor levels. Certain instructions require the supervisor privilege to execute, otherwise the CPU generates a privilege violation exception. The proposed two-layer software architecture calls for running the entire application partition, including the application executive itself, at the user level. This is essential to protect the isolation environment, setup by the system executive from being corrupted by any partition.

As part of its legitimate operation, however, the AE may require all external asynchronous interrupts to be disabled for a short period before they are re-enabled. Such requirement arises, for example, when handling semaphores, dynamic memory allocation, linked list pointers, and other critical sections of the AE code. In general, the AE disables the interrupts to enforce its own concurrency control policy among the tasks running inside its partition. Since the enforcement of the time-based scheduled depends on the countdown timer interrupt being recognized as soon as it arrives, the AE must not disable the interrupts. This is one of the reasons the entire AE is to run in a less privileged mode than the SE. Therefore, any attempt by the AE to disable/enable interrupts will be treated as a privilege violation exception.

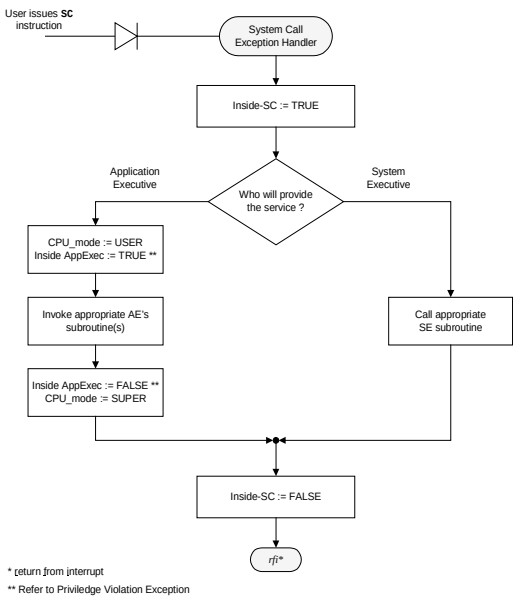

Figure 6: The *System Call* exception handler

To meet these two conflicting demands, we introduce the Privilege Violation exception handler, shown in figure 5. Having confirmed that the violation was made by the AE, the handler will set a flag, AE_Interrupt_Disabled, but will not physically disable the interrupts. One such flag exists for each AE in the system. If the timer generates an interrupt before the AE re-enables the interrupts, the AE_Interrupt_Disabled flag instructs the timer ISR not to deliver the clock tick(s), as indicated in figure 4. At the proper time, the AE will re-enable interrupts, and the privilege violation exception will deliver the missed clock tick(s), if any.

3.2.2 System Call Exception. User mode applications should be able to request services from the system executive, which runs in Supervisor mode. This is usually supported by the System Call instruction. The system call is considered a software-generated exception, thereby automatically elevating the processor to supervisor mode. Many system executive's services require such high privilege in order to access hardware resources. The handling of the system call exception is demonstrated in figure 6. Certain system calls will be served by the system executive. Others will be relayed to the AE for service after resetting the CPU back to User privilege level. In fact, the appropriate AE functions are invoked as regular subroutines from within the System Call exception handler. These subroutines are identified by studying the exception handlers of the each application executive. For each AE, there exists a flag, called Inside-AppExec, which indicates whether the CPU is executing inside that AE code. This flag is used by the Privilege Violation exception handler, as demonstrated in figure 5.

4. Prototype Implementation

In order to prove and demonstrate the feasibility of the proposed architecture, we implemented a prototype using a commercial embedded board. The board houses an 80 MHz Motorola PowerPC 603e and 32MB of RAM. The PowerPC 603e provides a memory management unit (MMU) and a high-resolution 64-bit time base register. The PowerPC also contains a decrementer register that can be programmed to generate an interrupt after a pre-specified period. Both the time base and the decrementer registers have a resolution of 120 nanoseconds. On top of the hardware platform, we implemented the system executive that cyclically schedules heterogeneous application executives such as Wind River's VxWorks [14] and Cygnus's eCos [15]. VxWorks and eCos are two extreme cases of applications executives from the perception of our design. VxWorks provides binary kernel and modifiable board support packages written in C, while eCos provides all C++ kernel sources to the designer.

We were confronted by several implementation issues that we had to address. First, there was the issue of temporal and spatial isolation of application partitions. To guarantee temporal partitioning, we built a cyclic scheduler in the SE that dispatches partitions based on statically built timetable. Necessary timer interrupts for the cyclic scheduling were generated using the decrementer facility of the PowerPC. Reference timing was provided by the 64-bit time base register of the PowerPC. To guarantee spatial partitioning, we strictly applied memory protection rules among applications and between the SE and application partitions. Both the BAT (Block Address Translation) and Segmentation/Paging schemes of the MMU on the PowerPC were applied in order to implement spatial partitioning. The BAT mechanism was used to protect the SE's address space from the applications whereas Segmentation/Paging mechanism is used to protect each application's address space from other application partitions. Combined use of the two different techniques enabled us to achieve very low overhead in context switching. Since significant invocations of SE's exception handling services are expected, the BAT scheme was used to map SE addresses to expedite access to these services. All these invocations require context switching between AE and SE because the execution environment is different.

We confronted with at least five other implementation issues for porting a new RTOS in our software environment. These issues include the initialization, time tick service, exception handling, the SE's services, and user mode restriction. Since the SE exclusively initializes sharable hardware resources such as CPU, MMU and Interrupt Controller, the AE is allowed to initialize its own kernel information only. In regard to the time tick service, the SE should inform the AE of the arrival of a local time tick. The frequency of local time ticks is a parameter of each AE. Then it is the responsibility of the AE to run its

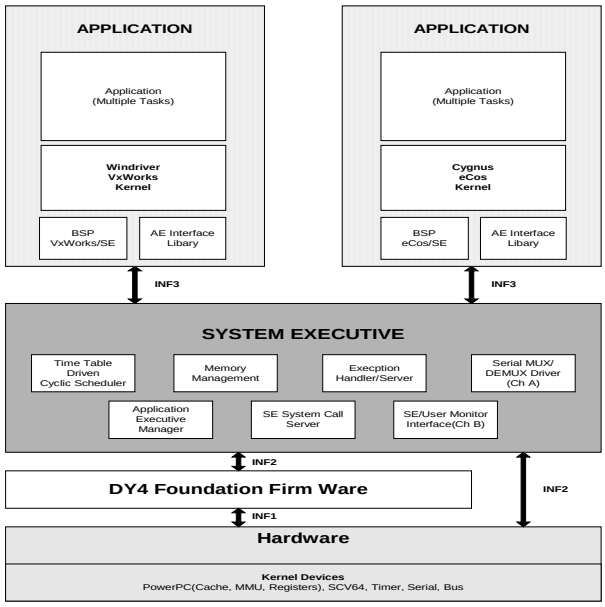

Figure 7: Prototype detailed design

own time tick service routine that may cause, for example, thread switching. Exceptions are handled at two stages. First, the SE traps all exceptions and handles only those that threaten the temporal or spatial isolation. The SE relays the remaining exceptions to the currently running AE, which then performs its own exception handling strategy. The most important porting challenge of all, was to force that the entire code of an AE to run in user mode instead of the privileged supervisor mode. Kernel routines that attempt to enable or disable the external interrupts or to perform thread switching inside the application partition were all replaced by interface library routines. Such a way prevents violation of application partitioning.

As expected, it was easier to port an open-source RTOS, such as eCos, than a one with only a binary kernel such as VxWorks. This was because most of the porting efforts were related to the kernel itself not the board support packages of VxWorks. However, we were able to successfully port VxWorks by analyzing its binary kernel and by trapping and replacing the routines, that assumes privilege execution level. The design of the prototype environment is depicted in figure 7.

The two-layer architecture introduces overhead that slows the system performance. The overhead generally affects AE's operations related to clock tick handling and privileged instructions. Through the experiment, we measured the following SE's operations to quantify the impact on the application performance:

- *Clock tick overhead:* The execution time for the timer ISR related to partition scheduling and time management is found to be about 8 µs. That overhead affects the AE's tick handling.
- *Partition context switching:* The average inter-partitioning context switching time was observed to be about 40 µs. This time reflects saving the context (about 40 CPU registers) of the preempted partition, setting the hardware timer to accommodate the new AE's time slice and local time tick frequency, and finally loading of the new partition's context.
- *Intra-AE task context switching:* Generally the machine-status saving and loading is no longer done by the AE and a SE library is to be invoked to provide such a service. Switching to the SE usually extends the time for saving and restoring the machine status. If the context switching is triggered by the termination or suspension of a task, the overhead is minimal (in the order of 2-3 µs) compared to the case where the AE runs in privileged mode. Task switching triggered by tick events and scheduling will be increased by about 8 µs since the SE timer ISR is invoked.
- *Emulating privileged instruction:* Since both VxWorks and eCos applications usually run with the RTOS in privileged mode, the system call exception did not affect the AE operations in our experiment. For privilege violation exception such as interrupt disable instruction, the overhead of trapping such instruction was observed to be about 5 µs.

5. Related Work

There have been a growing number of contemporary real-time operating systems (RTOSs) both in the academia and in the commercial marketplace. Systems such as MARUTI [16] and MARS [17] represent a time-triggered approach in the scheduling and dispatching of safety-critical real-time applications, including avionics. However these systems, either provide no partitioning scheme, as in the case of MARUTI, or they rely completely on proprietary hardware, as in the case of MARS.

Commercial RTOSs such as Neutrino [18] provide varying levels of memory isolation for applications. However, applications written for any of them are RTOS specific. In addition, it is not possible for applications, which are written for one RTOS to coexist on the same CPU with a different RTOS without a considerable effort of re-coding and re-testing. Our approach overcomes those drawbacks by ensuring both spatial and temporal partitioning while allowing the integration of application developed using a contemporary RTOS.

The Airplane Information Management System (AIMS) on board the Boeing 777 commercial airplane is among the few examples of IMA based systems [24]. Although, the AIMS and other currently used modular avionics setup offer strong partitioning, they lack the ability of integrating legacy and third party's applications. In our approach, a partition is an operating system encompassing its application. All previous architectures have the application task(s) as the unit of partitioning.

The system executive, in our approach, can be viewed as virtual machine monitor that exports the underlining hardware architecture to the application partitions. The concept of virtual machines has been used for variety of reasons such as cross-platform development [19], fault tolerance [20], and for the development of hardware-independent software [21]. Although some of these virtual machines restrict memory access to maintain partitioning, we are not aware of any that enforces temporal partitioning or supports real-time operating systems. Needless to say that, it does not guarantee any hard real-time constraints any better that the host operating system does.

Decomposing the operating system services into a µ-kernel augmented with multiple user-level modules has been the subject of extensive research, such as SPIN [10], Flux [23], Exokernel [22], and L4 [12]. The main objective of these µ-kernel-based systems is to efficiently handle domain-specific applications by offering flexibility, modularity, and extendibility. However, none of these systems is suitable for hard real-time applications.

RT-Mach and CVOE are among the few µ-kernel based RTOSs that supports spatial and temporal partitioning. Temporal guarantees in RT-Mach are

provided through an operating system abstraction called processor reserve [26]. Processor reservation is accepted through an admission control mechanism employing a defined policy. CVOE is an extendable RTOS that facilitates integration through the use a callback mechanism to invoke application tasks [25]. Yet tasks from different applications will be mixed and thus legacy application can not smoothly be integrated without substantial modification and revalidation.

6. Conclusion and Future Work

In this paper, we described the design and implementation of a two-layer architecture to support the Integrated Modular Avionics approach. The presented architecture allows for the safe integration of applications including their real-time operating systems as well. Legacy application software need not be changed to work in the new architecture. We consider that encapsulating the application and its RTOS in one partition and having multiple such partitions to safely exist in the same CPU is one of the strongest achievements of our work.

As a natural extension to this work, we are currently designing and implementing the required inter partition communication schemes that will provide a reliable channel between partitions and allow for both one-to-one and broadcast type of communication. In addition, we also plan to conduct fault-injection experiments to test the resilience of the presented software architecture.

References

[1] "Design Guide for Integrated Modular Avionics", ARINC report 651, Published by Aeronautical Radio Inc., Annapolis, MD, November 1991.

[2] B. W. Boehm, *Software Engineering Economics*. Prentice Hall, 1981.

[3] C. McClure, *Software Reuse Techniques: Adding Reuse to the Systems Development Process.* Prentice Hall, 1997.

[4] RTCA Inc., *Document RTCA/DO-178B*. Federal Aviation Administration, January 11, 1993.

[5] *IEEE 1003.13-1998*. IEEE Standard for Information Technology--Standardized Application Environment Profile (AEP)--POSIX®--Real-time Application Support, The Institute of Electrical and Electronics Engineers, 1998.

[6] "Avionics Application Software Standard Interface", ARINC Specification 653, Published by Aeronautical Radio Inc., Annapolis, MD, January 1997.

[7] "Minimum Operational Performance Standards for Avionics Computer Resource," RTCA SC-182/EUROCAE WG-48, November 1998.

[8] J. Rushby, "Partitioning in Avionics Architecture: Requirements, Mechanisms and Assurance," *Technical Report CR-1999-209347*, NASA, 1999.

[9] N. Carter, et al., "Hardware Support for Fast Capability-based Addressing," in the *Proceedings of the 6th International Conference on Architectural Support for Programming Lang. and Operating Systems*, pp. 319-327, San Jose, CA, October 1994.

[10] B. Bershad, et al., "Extensibility, Safety and Performance in the SPIN Operating System," in the *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pp. 267-284, Copper Mountain, Colorado, December 1995.

[11] R. Wahbe et al., "Efficient Software-Based Fault Isolation," in the *Proceedings of 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pp. 203-216, Asheville, NC, Dec. 1993.

[12] H. Härtig, et al. "The Performance of µ-Kernel-based Systems," P*roc. of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malô, France, Oct. 1997.

[13] J. Liedtke, "On Micro-Kernel Construction," in the *proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, Copper Mountain Resort, CO, Dec. 1995.

[14] WindRiver System. VxWorks. www.wrs.com

[15] Cygnus. embedded Cygnus operating system (eCos). www.cygnus.com/ecos

[16] Systems Design and Analysis Group, University of Maryland, College Park. The Maruti Hard Real-Time Operating System. www.cs.umd.edu/projects/maruti

[17] H. Kopetz, et. al. "Distributed fault-tolerant real-time systems: The MARS Approach," *IEEE Micro*, Vol. 9, No.1, pp. 25-40, February 1989.

[18] QNX, *Neutrino*. www.qnx.com

[19] E. P. Goldberg, "*Survey of Virtual Machine Research*," IEEE Computer, pp. 34-45, June 1974.

[20] T. C. Bressoud and F. P. Schneider, "Hypervisor-based Fault-tolerance" in the *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, pp.1-11, Dec. 1995.

[21] J. Gosling and H. McGilton, "The Java Language Environment: A White paper," *Technical Report*, Sun Microsystems Computer Company, 1996.

[22] M. Kaashoek, et al., "Application performance and flexibility on exokernel systems," In *the Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pp. 52-65, Saint-Malô, France, October 1997.

[23] Bryan Ford, et al., "Microkernels Meet Recursive Virtual Machines," *Proceedings. of the* 2nd Symposium on Operating Systems Design and Implementation (OSDI'96), October 1996.

[24] M. Johnson, "Boeing 777 Airplane Information Management System – Philosophy and Displays", *Proceedings of the Royal Aeronautical Society's Advanced Avionics Conference on Aq330/A340 & the Boeing 777 aircraft*, London, UK, Nov. 1993.

[25] "Architecture Document for the Combat Vehicle Operating Environment," Technical report, Texas Instruments, Defense Systems & Electronics Group, 1997.

[26] C. Mercer, R. Rajkumar, and J. Zelenka, "Temporal Protection in Real-Time Operating Systems," in the proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software, May 1994.