Adaptability Support in Time- and Space-Partitioned Aerospace Systems

João Craveiro and José Rufino
Universidade de Lisboa, Faculdade de Ciências, LaSIGE
FCUL, Ed. C6, Piso 3, Campo Grande, 1749-016 Lisbon, Portugal
E-mail: jcraveiro@lasige.di.fc.ul.pt, ruf@di.fc.ul.pt

***Abstract*—The AIR (ARINC 653 in Space Real-Time Operating System) technology targets modern aerospace systems, where the concepts of time- and space-partitioning are applied. AIR features advanced timeliness control and adaptation mechanisms in its design, such as mode-based schedules, process deadline violation monitoring, and protection against event overload. The timing parameters of a space mission may vary throughout time, according to its mode/phase of operation, and the spacecraft may be exposed to unpredictable events and failures. In this paper we explore the adaptation potential of the advanced features included in AIR, analysing their code complexity (which influences software verification, validation and certification efforts) and computational complexity (which correlates to the temporal overhead impact on the system), and discussing how they can be applied to provide more adaptive, reconfigurable and self-adaptive AIR-based systems.**

***Keywords*-adaptive systems; aerospace industry; logic partitioning; processor scheduling; real-time systems**

I. Introduction

The computing infrastructure aboard spacecrafts employs embedded systems to cope with strict dependability and real-time requirements. Additionally, cost concerns call for flexible resource reallocation and the overall reduction of size, weight and power consumption (SWaP). To address SWaP requisites, functions which traditionally received dedicated resources are now integrated in a shared computing platform. As these functions may have different degrees of criticality and predictability and/or originate from multiple suppliers, this integration brings potential safety hazards, for which the architectural approach of time- and space-partitioning (TSP) has been proposed. Applications are separated into logical containers, *partitions*, for the benefit of fault containment, software integration, and independent verification, validation and certification processes. The aviation industry had already made a similar move, by transitioning from federated architectures to Integrated Modular Avionics (IMA) [1].

The AIR architecture [2] was defined in response to the interest of space agencies, namely the European Space Agency (ESA), and industry partners in applying TSP concepts to the spacecraft onboard computing resources [3]. Each partition hosts its own application and operating system, either a real-time operating system (RTOS) or a generic non-real-time one. AIR employs a two-level scheduling scheme. Partitions are scheduled under a predetermined, cyclically repeated, sequence of time windows. Inside each partition's time windows, the respective processes compete according to the native operating system's process scheduler policy.

In normal conditions, a mission goes through multiple phases (flight, approach, exploration) [4]. Additionally, unplanned circumstances may happen, such as unforeseeable external events and internal failures. Adaptation to changing/unexpected conditions is of great importance for a mission's survival. This kind of adaptation can include mechanisms of support for assisted adaptation, reconfiguration capabilities, and self-adaptation according to environmental information. Such need is more stringent in the case of unmanned missions. Flexible adaptation to unforeseen events is of paramount importance, and has been proven to prolong the lifetime of unmanned space vehicles by years [5].

In this paper, we explore the adaptation, reconfiguration and self-adaptation potential of the AIR architecture, with respect to timeliness control and failure detection/recovery.

Related work

To the best of our knowledge, the only contemporary approach to flexible scheduling in a TSP system is the mode-based scheduling feature provided by the commercial Wind River VxWorks 653 solution [6]. Previous academic research on TSP solutions [7] and works on scheduling analysis for TSP systems [8], [9], [10] do not state including or foreseeing mechanisms for timing parameters adaptation. Architectural alternatives to TSP/IMA are compared in [11] in terms of features which contribute to adaptability, and recommendations are made to include adaptive features in IMA-like architectures. Preliminary results are presented in [12] for a reconfigurable IMA platform. Emergence and increasing acceptance of adaptive and reconfigurable control in unmanned aerial systems is pointed out in [13].

Paper outline

Section II provides an overview of the AIR architecture for self-containment of the issues at hand. Section III describes the advanced timeliness control mechanisms provided by AIR, while Section IV discusses how they allow adaptive behaviour. Section V exposes the implementation and evaluation of a prototype demonstrator of these capabilities. Finally, Section VI closes the paper with concluding remarks and future work directions.

II. AIR ARCHITECTURE FOR TSP SYSTEMS

The AIR design [2] evolved from a proof of feasibility for adding ARINC 653 support to the Real-Time Executive for Multiprocessor Systems (RTEMS) to a multi-OS (operating system) time- and space-partitioned (TSP) architecture. Its modular design aims at high levels of flexibility, hardware- and OS-independence, and independent component verification, validation and certification.

Each partition can host a different OS (the partition operating system, POS), which in turn can be either a real-time operating system (RTOS) or a generic non-real-time one. We will now describe the AIR architecture in enough detail for the scope of this paper. A more in-depth description of AIR can be found in [2].

A. System architecture

The modular design of the AIR architecture is pictured in Figure 1. The *AIR Partition Management Kernel (PMK)* is the basis of the Core Software Layer of an AIR-based system. The AIR PMK hosts crucial functionality such as partition scheduling and dispatching, low-level interrupt management, and interpartition communication support.

The *AIR POS Adaptation Layer (PAL)* encapsulates the POS of each partition, providing an adequate POS-independent interface to the surrounding components.

The *APEX Interface* component provides a standard programming interface derived from the ARINC 653 specification [14], with the possibility of being subsetted and/or adding specific extensions for certain partitions [15].

AIR also incorporates *Health Monitoring (HM)* functions to contain faults within their domains of occurrence and to provide the corresponding error handling capabilities. Support to these functions is spread throughout virtually all of the AIR architecture's components.

B. Two-level scheduling scheme

The AIR technology employs a two-level scheduling scheme (Figure 2). The first level corresponds to partition scheduling and the second level to process scheduling. Partitions are scheduled on a cyclic basis, according to a partition scheduling table (PST) repeating over a major time frame (MTF). This table assigns execution time windows to

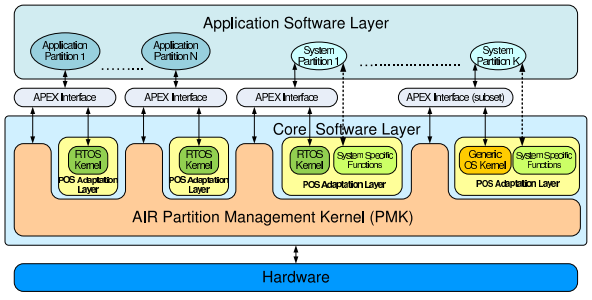

Figure 1. AIR system architecture

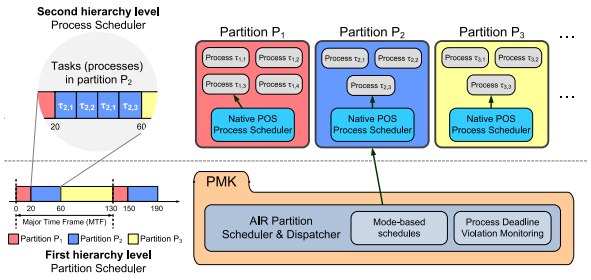

Figure 2. Two-level scheduling scheme

partitions. Inside each partition's time windows, its processes compete according to the POS's native process scheduler.

III. DESIGNING FOR ADAPTABILITY

The potential to adapt to changing environmental or operating conditions is crucial for space missions. Existing studies show that a number of failures may be mitigated by software reconfigurability [5].

AIR employs special design and engineering decisions to address specific adaptation requirements, namely on the temporal domain and on failure detection and recovery.

A. Mode-based schedules

Timing requirements are among the conditions which may change according to a mission's phase, since certain functions should only execute during certain phases. Under the basic mandatory scheme defined in ARINC 653 [14], there is a single fixed PST under which all partitions are scheduled. This PST must be tailored to attended to the temporal requisites of the partitions in all of the phases the mission goes through. Generating such a PST is a difficult task (even assisted by a tool), and the end result may be a mission with frequent resource utilization waste.

AIR approaches this issue with the notion of *mode-based schedules*, inspired by the optional service defined in ARINC 653 Part 2 [16]. Instead of one fixed PST, the system can be configured with multiple PSTs, which may differ in terms of the MTF duration, of which partitions are scheduled, and of how much processor time is assigned to them. The system can then switch between these PSTs; this is performed through a service call issued by an authorized and/or dedicated partition. To avoid violating temporal requirements, a PST switch request is only effectively granted at the end of the ongoing MTF.

The AIR Partition Scheduler, with the addition of mode-based schedules, functions as described in Algorithm 1. The first verification to be made is whether the current instant is a partition preemption point (line 2). In case it is not, the execution of the partition scheduler is over; this is both the best case and the most frequent one. If it is a partition preemption point, a verification is made (line 3) as to whether there is a pending scheduling switch to be applied and the current instant is the end of the

Algorithm 1 AIR Partition Scheduler featuring mode-based schedules

1: $ticks \leftarrow ticks + 1$ ▷ $ticks$ is the global system clock tick counter
2: **if** $schedules_{currentSchedule}.table_{tableIterator}.tick = (ticks - lastScheduleSwitch) \bmod schedules_{currentSchedule}.mtf$
3: **if** $currentSchedule \neq nextSchedule \wedge (ticks - lastScheduleSwitch) \bmod schedules_{currentSchedule}.mtf = 0$
4: $currentSchedule \leftarrow nextSchedule$
5: $lastScheduleSwitch \leftarrow ticks$
6: $tableIterator \leftarrow 0$
7: **end if**
8: $heirPartition \leftarrow schedules_{currentSchedule}.table_{tableIterator}.partition$
9: $tableIterator \leftarrow (tableIterator + 1) \bmod schedules_{currentSchedule}.numberPartitionPreemptionPoints$
10: **end if**

MTF. If these conditions apply, then a different PST will be used henceforth (line 4). The partition which will hold the processing resources until the next preemption point, dubbed the heir partition, is obtained from the PST in use (line 8) and the AIR Partition Scheduler will now be set to expect the next partition preemption point (line 9). Generation of different partition schedules can be aided by a tool that applies rules and formulas to the temporal requirements of the constituent processes of the necessary partitions [2], [17].

B. Process deadline violation monitoring

During system execution, it may be the case that a process exceeds its deadline. This can be caused by a malfunction, by transient overload (e. g., due to abnormally high event signalling rates), or by the underestimation of a process's worst case execution time (WCET) at system configuration and integration time. Factors related to faulty system planning (such as the time windows not satisfying the partitions' timing requirements) could, in principle, also cause deadline violations. However, such issues can be predicted and avoided using offline tools that verify the fulfilment of timing requirements [2], [17].

In addition, it is also possible that a process exceeds a deadline while the partition in which it executes is inactive. This violation will only be detected when the partition is being dispatched, just before invoking the process scheduler.

Deadline verification, which is invoked for the currently active partition immediately following the announcement of elapsed system clock ticks, obeys to Algorithm 2. In the absence of a violation, only the earliest deadline is checked. Subsequent deadlines may be verified until one has not been missed. This methodology is optimal with respect to deadline violation detection latency, which upper bound for each partition is the maximum interval between two consecutive time windows.

Figure 3 provides a use-case scenario for the process deadline violation monitoring functionality. The exemplified process (with the identifier pid) has a relative deadline of $t_3 - t_1$. When it becomes ready at $t = t_1$ via the START primitive, its absolute deadline time is set to t_3. At $t = t_2$,

Algorithm 2 Deadline verification at the AIR PAL level

1: *PAL_CLOCKTICKANNOUNCE($elapsedTicks$)
2: **for all** $d \in PAL\_deadlines$ **do**
3: **if** $d.deadlineTime \geq$PAL_GETCURRENTTIME( )
4: **break**
5: **end if**
6: HM_DEADLINEVIOLATED($d.pid$) ▷ pid: process identifier
7: PAL_REMOVEPROCESSDEADLINE(d)
8: **end for**

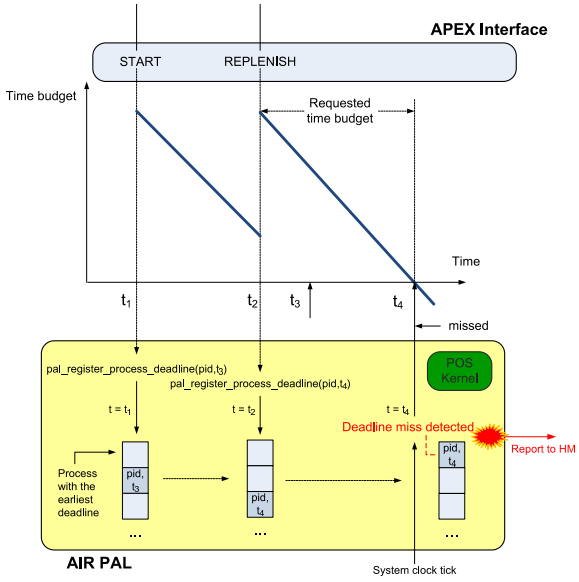

Figure 3. Process deadline violation monitoring example

the REPLENISH primitive is called, requesting a deadline time replenishment, so that the new absolute deadline time is t_4. When the instant $t = t_4$ arrives and the process pid has not yet finished its execution (e. g., having called the PERIODIC_WAIT primitive to suspend itself until its next release point), its deadline will be the earliest registered in the respective AIR PAL data structure. The violation is then detected, and reported to the Health Monitor.

C. Health Monitor and error handling

In the context of fault detection and isolation, ARINC 653 classifies process deadline violation as a process level error (that impacts one or more processes in the partition, or the entire partition) [14]. The action to be performed in the event of an error is defined by the application programmer through an appropriate error handler [2].

The error handler is an application process tailored for partition-wide error processing. The occurrence of process-/partition-level errors may be signalled through interpartition communication to a (system partition) process performing a Fault Detection, Isolation and Recovery (FDIR) function for the spacecraft [4]. A system-wide reconfigurability logic should be included in FDIR.

Table I
ESSENTIAL APEX SERVICES TO SUPPORT MODE-BASED SCHEDULES

Primitive	Short description
SET_SCHEDULE	Request for a new PST to be adopted at the end of the current MTF
GET_SCHEDULE_STATUS	Obtain current schedule, next schedule (same as current schedule if no PST change is pending), and time of the last schedule switch

Table II
APEX SERVICES IN NEED OF MODIFICATIONS TO SUPPORT PROCESS DEADLINE VIOLATION MONITORING

Primitive	Short description
Need to register/update deadline	
[DELAYED_]START	Start a process [with a given delay]
PERIODIC_WAIT	Suspend execution of a (periodic) process until the next release point
REPLENISH	Postpone a process's deadline time
Need to unregister deadline	
STOP[_SELF]	Stop a process [itself]

Table III
ESSENTIAL APEX SERVICES FOR HEALTH MONITORING

Primitive	Short description
RAISE_APPLICATION_ERROR	Notify the error handler for a specific error type

D. Impact on APEX services

To allow application programmers to use the mode-based schedules functionality, the APEX interface is extended with additional primitives presented in Table I.

Process deadline monitoring calls for adaptation of process management services, encapsulated by appropriated AIR PAL functions. Table II shows the APEX primitives which need to register deadline information (either updating the deadline information for the process, or inserting it if it does not exist yet) or unregister deadline information (removing any information on the respective process from the AIR PAL deadline verification data structures). The APEX primitive RAISE_APPLICATION_ERROR, described in Table III, is used to report a process deadline violation to the HM and trigger the defined error handler.

E. Low-level event overload control

The AIR PMK includes advanced adaptation mechanisms to control the timeliness of asynchronous events signalled through processor interrupts. These mechanisms use the discrete event processing methodology described in [18], and have been adapted for application on the AIR architecture. The resulting approach, integrating with the Health Monitor, is presented in Algorithm 3. The continuous time, t, is transformed into a discrete time n through a *sample/hold* function (line 2). The discrete time is used for the determination of the event (interrupt) metrics, such as its rate of occurrence, using recursive digital filters (line 3). The complexity of computing some filtering functions inside the interrupt service routing (ISR) is removed resorting to previously built lookup tables [18]. If a specific metric upper threshold M is exceeded, interrupt service is disabled and the HM is notified to take action. The corresponding error handler then polls events at a predefined fixed rate. The effectiveness of this approach is illustrated in Figure 4, using an Infinite Impulse Response (IIR) filter for rate determination [18]. When the overload state is over, and the relevant metric value y_n decreases below the lower threshold m, the error handler enables the interrupts and returns to the state of waiting for further overload notifications.

Algorithm 3 Event (interrupt) overload control

1: $t \leftarrow$ GETTIME()
2: $n \leftarrow$ SAMPLEHOLD(t)
3: $y_{n+1} \leftarrow$ FILTERING(n, n_{last}, y_n) ▷ Event metric determination
4: $n_{\text{last}} \leftarrow n$
5: **if** $y_n > M$ ▷ Overload detection
6: DISABLEINTERRUPT(irq) ▷ irq: interrupt request number
7: HM_EVENTOVERLOAD(irq)
8: **end if**

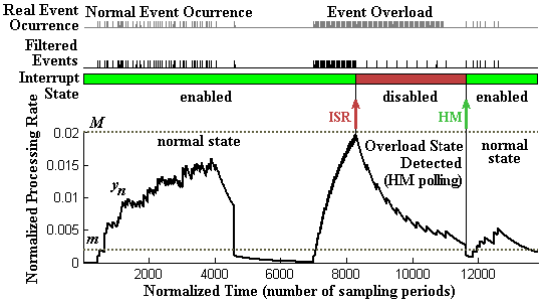

Figure 4. Operation of overload control mechanisms [18]

IV. ACHIEVING ADAPTABILITY

The described features can be used to allow the adaptation of the system to changing conditions, either by the action of a human operator (adaptability) or autonomously after processing acquired information (self-adaptability).

A. Adaptability and reconfigurability

By offering the possibility to host multiple PSTs and switch among them by demand during the execution of the system, AIR allows for guided adaptation of the system to the mission's different phases and to detected operational condition changes.

The proposed reconfigurability model also allows updating the set of available PSTs during the operation of the mission, by issuing an update from ground control. This process can be extended to allow updating the applications running inside the partitions [15]. Process deadline violation monitoring information can be useful in detecting the need for PSTs and/or applications to be updated.

B. Self-adaptability

Besides ground control commands, a request to use a different PST can also be autonomously issued from the interpretation of the internal and external operating conditions of the mission. The Attitude and Orbit Control Subsystem (AOCS) [4] should detect when the mission should transit from flight to approach and from approach to exploration, for instance, and (propose to) switch schedules accordingly.

Another self-adaptation use of mode-based schedules can profit from process deadline violation monitoring. The system integrator can include several PSTs that fulfil the same set of temporal requirements for the partitions, but distribute additional spare time inside the MTF among these partitions in different ways. This set of PSTs, coupled with an appropriate HM handler for the event of process deadline violation, can be used to temporarily or permanently assign additional processing time to a partition hosting an application which has repeatedly observed to be missing deadlines.

An additional degree of self-adaptability is obtained by applying event overload control mechanisms with reconfigurable parameters to preserve timeliness [18].

V. Proof-of-concept prototype and evaluation

The core mechanisms for adaptability and reconfigurability of the AIR architecture have been prototyped on both Intel IA-32 and SPARC LEON platforms. Each partition executes an RTEMS-based [19] mockup application representative of typical functions present in a spacecraft.

A. Code complexity

Critical software, namely that developed to go aboard a space vehicle, goes through a strict process of verification, validation and certification. Code complexity increases the effort of this process and the probability of there being bugs.

One metric for code complexity is its size, in lines of source code. Since equivalent code can be arranged in ways which account for different lines of code counts, standardized accounting methods must be used. We employ the *logical source lines of code (logical SLOC)* metric of the Unified CodeCount tool [20]. Another useful metric is *cyclomatic complexity*, which gives an upper bound for the number of test cases needed for full branch coverage and a lower bound for those needed for full path coverage.

The C implementation of Algorithm 1 is accounted for in Table IV, which shows its logical SLOC count and cyclomatic complexity. Code introduced at the AIR PAL level to achieve process deadline violation monitoring is accounted for in Table V. The total complexity added in terms of code executed during a clock tick ISR is a small fraction of that already present in the underlying ISR code.

B. Computational complexity analysis

Since the verifications of deadlines and of the need to apply a new PST are executed inside the system clock tick ISR, they must have a bounded execution time; computational complexity is a good indicator thereof.

Table IV
Logical SLOC and cyclomatic complexity (CC) for the AIR Partition Scheduler with mode-based schedules

	Logical SLOC	CC
AIR Partition Scheduler [a]	13	4
Underlying clock tick ISR	>190 [b]	>20

[a] Algorithm 1
[b] C code only; plus >182 assembly instructions

Table V
Logical SLOC and cyclomatic complexity (CC) for the implementation of deadline violation monitoring in AIR PAL

	Logical SLOC	CC
Register deadline	34	6
Unregister deadline	12	3
Verify deadlines [a]	16	2

[a] Algorithm 2

Table VI
AIR Partition Scheduler (with mode-based schedules) execution time — basic metrics

Minimum (ns)	Maximum (ns)	Average (ns)
32	186	36

In the AIR Partition Scheduler (Algorithm 1), all instructions are $\mathcal{O}(1)$. Accesses to multielement structures are made by index, and thus their complexity does not depend on the number of elements or the position of the desired element.

Linear complexity is easily achievable for the majority of the executions of the process deadline verification at the AIR PAL level (Algorithm 2). By placing deadlines in a linked list in ascending order of deadline times, the earliest deadline is retrieved in constant time. The removal of a violated deadline from the data structure can also be made in constant time, since we already hold a pointer for the said deadline. In case of deadline violation, the next deadline(s) will successively be verified until reaching one that has not expired. Thus, the worst case wields $\mathcal{O}(n)$, where n is the number of processes in the partition. However, by design, the number of processes is bounded, and these worst cases are highly exceptional and mean the total and complete failure of the partition, which should be signalled to the HM.

C. Basic execution metrics

AIR Partition Scheduler execution time has been measured, resulting in the basic metrics shown in Table VI. These values were obtained executing the AIR prototype demonstrator on a native machine equipped with an Intel IA-32 CPU with a clock of 2833 MHz. Time readings are obtained from the CPU Time Stamp Counter (TSC)

64-bit register, being rounded to 1 ns. Measurements showed negligible variation between sample executions.

Though encouraging, these results should be enriched. Further work will compare them with the execution of the whole clock tick ISR, dissect the execution time by subcomponents, and identify trends in execution time variation.

VI. Conclusion and future work

In this paper, we detailed and analysed fundamental mechanisms providing support for adaptive and self-adaptive behaviour to aerospace missions based on the AIR architecture for time- and space-partitioned systems. Support for mode-based partition schedules directly provides the capacity for adaptation by direct order from ground control, and also for self-adaptation according to the perceived operational conditions of the mission. The mechanisms for process deadline violation monitoring and protection against event overload allow controlling the overall timeliness, through self-adaptation in the presence of exceptional and uncertain operational conditions.

Besides implementing self-adaptation as a function of process deadline violations, future work includes adding AIR the capability to receive and apply updates to the currently installed applications and PSTs [15]. We are also currently exploring the approach of taking advantage of multicore platforms in AIR [2]. The availability of multiple cores can be applied for fault tolerance, with the system adapting to a hardware failure by assigning an application to a different core than the (failed) current one.

Acknowledgment

This work was partially developed within the scope of the European Space Agency Innovation Triangle Initiative program, through ESTEC Contract 21217/07/NL/CB, Project AIR-II (ARINC 653 in Space RTOS – Industrial Initiative). This work was partially supported by FCT (Fundação para a Ciência e a Tecnologia), through the Multiannual Funding and CMU-Portugal Programs and the Individual Doctoral Grant SFRH/BD/60193/2009.

References

[1] AEEC, "Design guidance for Integrated Modular Avionics," Aeronautical Radio, Inc., ARINC Report 651-1, Nov. 1997.

[2] J. Rufino, J. Craveiro, and P. Verissimo, "Architecting robustness and timeliness in a new generation of aerospace systems," in *Architecting Dependable Systems 7*, ser. LNCS, A. Casimiro, R. de Lemos, and C. Gacek, Eds. Springer, 2010, accepted for publication.

[3] TSP Working Group, "Avionics time and space partitioning user needs," ESA, European Space Research and Technology Centre, Technical Note TEC-SW/09-247/JW, Aug. 2009.

[4] P. W. Fortescue, J. P. W. Stark, and G. Swinerd, Eds., *Spacecraft Systems Engineering, 3rd edition*. Wiley, 2003.

[5] M. Tafazoli, "A study of on-orbit spacecraft failures," *Acta Astronautica*, vol. 64, no. 2–3, pp. 195–205, Jan.–Feb. 2009.

[6] Wind River, "Wind River VxWorks 653 Platform 2.3," Apr. 2010, retrieved Jun 17, 2010. [Online]. Available: http://www.windriver.com/products/product-notes/PN_VE_653_Platform2_3_0410.pdf

[7] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: the XtratuM approach," in *Proc. Eighth European Dependable Computing Conf.*, Valencia, Spain, Apr. 2010.

[8] N. Audsley and A. Wellings, "Analysing APEX applications," in *Proc. 17th IEEE Real-Time Systems Symp.*, Washington DC, USA, Dec. 1996, pp. 39–44.

[9] Y. Lee, D. Kim, M. Younis, and J. Zhou, "Partition scheduling in APEX runtime environment for embedded avionics software," in *Proc. 5th Int. Conf. on Real-Time Computing Systems and Applications*, Hiroshima, Japan, 1998.

[10] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal, "A compositional scheduling framework for digital avionics systems," in *Proc. 15th IEEE Int. Conf. Embedded Real-Time Computing Systems and Applications*, Beijing, China, Aug. 2009.

[11] B. Ford, P. Bull, A. Grigg, L. Guan, and I. Phillips, "Adaptive architectures for future highly dependable, real-time systems," in *Proc. 7th Ann. Conf. on Systems Engineering Research*, Loughborough, United Kingdom, Apr. 2009.

[12] P. Bieber, E. Noulard, C. Pagetti, T. Planche, and F. Vialard, "Preliminary design of future reconfigurable IMA platforms," in *Second Int. Workshop on Adaptive and Reconfigurable Embedded Systems*, Grenoble, France, Oct. 2009, pp. 21–24.

[13] B. Vanek, "Future trends in UAS avionics," in *Proc. 10th Int. Symp. of Hungarian Researchers on Computational Intelligence and Informatics*, Budapest, Hungary, Nov. 2009.

[14] AEEC, "Avionics application software standard interface, part 1 - required services," Aeronautical Radio, Inc., ARINC Specification 653P1-2, Mar. 2006.

[15] J. Rosa, J. Craveiro, and J. Rufino, "Exploiting AIR composability towards spacecraft onboard software update," in *Actas do INForum - Simpósio de Informática 2010*, Braga, Portugal, Sep. 2010.

[16] AEEC, "Avionics application software standard interface, part 2 - extended services," Aeronautical Radio, Inc., ARINC Specification 653P2-1, Dec. 2008.

[17] J. Craveiro and J. Rufino, "Schedulability analysis in partitioned systems for aerospace avionics," in *Proc. 15th IEEE Int. Conf. on Emerging Technologies and Factory Automation*, Bilbao, Spain, Sep. 2010.

[18] M. Coutinho, J. Rufino, and C. Almeida, "Control of event handling timeliness in RTEMS," in *Proc. 17th IASTED Int. Conf. on Parallel and Distributed Computing and Systems*, Phoenix, USA, Nov. 2005.

[19] *RTEMS C Users Guide*, 4.8 ed., OAR Corp., Feb. 2008.

[20] V. Nguyen, S. Deeds-Rubin, and T. Tan, "A SLOC counting standard," in *The 22nd Int. Ann. Forum on COCOMO and Systems/Software Cost Modeling*, Los Angeles, USA, 2007.