Towards Integration of Adaptability and Non-Intrusive Runtime Verification in Avionic Systems *

José Rufino
jmrufino@ciencias.ulisboa.pt
LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal

ABSTRACT

Unmanned autonomous systems (UAS) avionics call for advanced computing system architectures fulfilling strict size, weight and power consumption (SWaP) requisites, decreasing the vehicle cost and ensuring the safety and timeliness of the system. The AIR (ARINC 653 in Space Real-Time Operating System) architecture defines a partitioned environment for the development and execution of aerospace applications, following the notion of time and space partitioning (TSP), preserving application timing and safety requisites.

The plan for a UAS mission may vary with the passage of time, according to its mode/phase of operation, and the vehicle may be exposed to unpredictable (environmental) events and failures, calling for the advanced adaptability and reconfigurability features included in the AIR architecture. This paper explores the potential of non-intrusive runtime verification (RV) mechanisms, currently being included in AIR, to improve system safety and to decrease the computational cost of timeliness adaptability and of the corresponding overhead on the system.

Categories and Subject Descriptors

C.4 [**Computer System Organisation**]: [Fault tolerance]; C.3 [**Special-Purpose and Application Based Systems**]: Real-time and embedded systems; D.4.7 [**Operating systems**]: Organization and Design—*Real-time systems and embedded systems*

Keywords

dependability, timeliness, adaptability, runtime verification, time and space partitioning, integrated modular avionics.

*This work was partially supported by FCT, through project PTDC/EEI-SCR/3200/2012 (READAPT) and through LaSIGE Strategic Project PEst-OE/EEI/UI0408/2014. This work integrates the activities of COST Action IC1402 - Runtime Verification beyond Monitoring (ARVI).

EWiLi'15, October 8th, 2015, Amsterdam, The Netherlands.

1. INTRODUCTION AND MOTIVATION

Avionic systems have strict safety and timeliness requirements as well as strong size, weight and power consumption (SWaP) constraints. Modern unmanned autonomous systems (UAS) avionics follow the civil aviation trend of transitioning from federated architectures to Integrated Modular Avionics (IMA) [1] and resort to the use of partitioning.

Partitioned architectures implement the logical separation of applications in criticality domains, named partitions, and allow hosting both avionic and payload functions in the same computational infrastructure, thus fulfilling both SWaP and safety/timeliness requirements [24]. The notion of temporal and spatial partitioning (TSP) implies that the execution of functions in one partition does not affect other partitions' timeliness and that separated addressing spaces are assigned to different partitions [21]. The design and development of AIR (ARINC 653 in Space Real-Time Operating System) has been motivated by the interest in applying TSP concepts to the aerospace domain [20].

Usually, an UAS mission goes through multiple phases (e.g., launch, flight, approach, exploration). Adaptation to changing temporal requirements throughout all the mission phases is of great importance for a mission's survival [23]. The design of AIR Technology already includes mechanisms of support for adaptation and reconfiguration. Nevertheless, given the high complexity of UAS functions, modern avionic systems may largely benefit from the verification in runtime that the system/mission specification is being fulfilled or that some deviation has just occurred.

This paper addresses how fundamental timeliness adaptation mechanisms can be combined with advanced runtime verification (RV) capabilities in time- and space-partitioned systems. To reduce the temporal overhead of such mechanisms in the operation of onboard systems an innovative non-intrusive design approach is followed.

The paper is organized as follows. Section 2 introduces the AIR architecture for TSP systems. Section 3 details the AIR adaptability and reconfigurability capabilities. Section 4 describes the non-intrusive RV features being introduced in the AIR architecture and Section 5 describes how to integrate them with AIR system adaptability and reconfigurability. Section 6 describes the related work and, finally, Section 7 issues concluding remarks and future research directions.

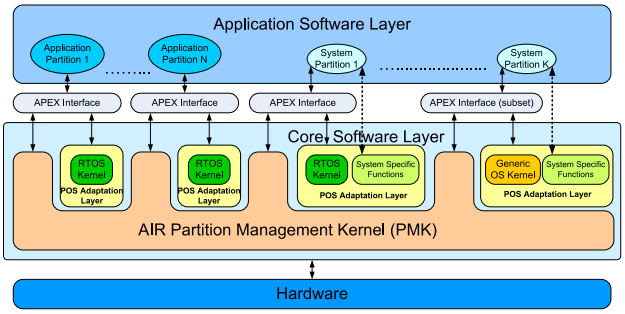

Figure 1: AIR architecture for TSP systems

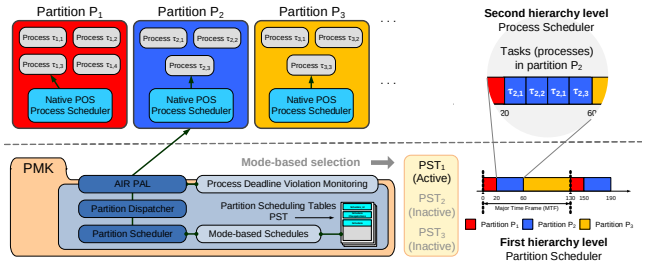

Figure 2: Two-level hierarchical scheduling with partition scheduling featuring mode-based schedules

2. AIR TECHNOLOGY FOR TSP SYSTEMS

The AIR Technology evolved from a proof of feasibility for adding ARINC 653 functional support to the Real-Time Executive for Multiprocessor Systems (RTEMS) to a multi-OS (operating system) TSP architecture [20]. The AIR modular design aims at high levels of flexibility, hardware- and OS-independence, easy integration and independent component verification, validation and certification.

2.1 System architecture

The AIR modular architecture is pictured in Figure 1. The *AIR Partition Management Kernel (PMK)* is the basis of a core software layer, enforcing robust TSP properties and hosting crucial functionality such as partition scheduling and dispatching, low-level interrupt management, and interpartition communication support. Temporal partitioning ensures that the real-time requisites of the different functions executing in each partition are guaranteed. Spatial partitioning relies on having dedicated addressing spaces for the functions executing on different partitions.

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

The *Portable Application Executive* (APEX) *interface* [22] provides a standard programming interface derived from the ARINC 653 specification [1], with the possibility of being subsetted and/or adding specific functional extensions for certain partitions [19].

The organization of vehicle functions in different partitions requires interpartition communication facilities, since a function hosted in a partition may need to exchange information with other partitions. Interpartition communication consists of the authorized transfer of information between partitions without violating neither spatial separation constrains nor information security properties [21, 20, 5].

2.2 Two-level scheduling

The AIR technology employs a two-level scheduling scheme, as illustrated in Figure 2. The first level corresponds to partition scheduling and the second level to process scheduling. Partitions are scheduled on a cyclic basis, through the partition scheduling and dispatching components (Figure 2), according to a partition scheduling table (PST) repeating over a major time frame (MTF). The PST assigns execution time windows to partitions. Inside each partition's time windows, its processes compete for processing resources according to the POS's native process scheduler.

2.3 Health monitoring and error handling

The AIR architecture incorporates *Health Monitor* (HM) functions that aim 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 architectural components.

The HM plays an important role in achieving system safety given it prevents/mitigates ill-effects of process and/or partition level errors in the remaining partitions. The action to be performed in the event of an error is defined by the application programmer through an appropriate error handler. This may comprise adaptability features such as the redefinition of timing and control parameters or the issue of a different schedule request. If no handler is provided, a response action defined by the partition's HM ARINC 653 configuration table is executed, as shown in Figure 3.

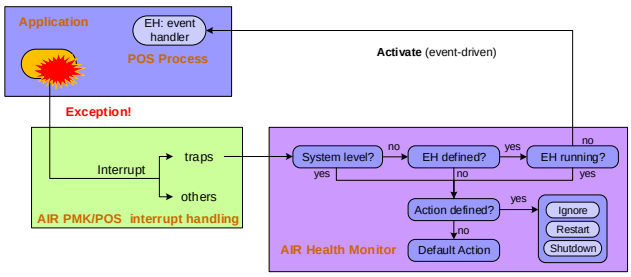

Figure 3: Health monitoring

The design of AIR allows HM handlers to simply replace existing exception handlers or to be added to existing ones, in pre- and/or post-processing modes.

3. ADAPTABILITY

The adaptation to changing environmental or operating conditions is crucial for unmanned space and aerial missions survivability, which can be significantly improved through software reconfigurability, as studied in [23].

The design of AIR integrates special-purpose mechanisms to address specific adaptation requirements, thoroughly described in [7] and summarised next.

3.1 Mode-based schedules

Timing requirements may change according to a mission's phase since certain functions should only execute during certain phases. The original ARINC 653 notion of a single fixed PST [1], defined offline, is limited in terms of timeliness adaptability, as well as safety and fault-tolerance control, and surely contributes to some degree of resource utilization waste. To address this primary limitation, the AIR design incorporates the notion of *mode-based partition schedules*, inspired by the optional service defined within the scope of ARINC 653 Part 2 specification [2].

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, as shown in Figure 2. The system can then switch between these PSTs; selection of the active PST 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.

Hosting multiple PSTs aboard autonomous vehicles opens room for the (self-)adaptability of unmanned missions, in function of passage of time and of changing environmental and operational conditions. Pre-generation of different partition schedules can be aided by a tool that applies rules and formulas to the temporal requirements of processes/partitions, taking into account the functions' needs in different anticipated conditions [20, 8]. Unforeseeable conditions can be handled thorough the mechanisms for remote update of PSTs and onboard software described in [19].

3.2 Process deadline violation monitoring

During runtime execution, it may be the case that a process exceeds its deadline. In the AIR architecture, the PAL component monitors, at each POS clock tick, if some process in the active partition has violated its deadline (Figure 2). In addition, it is also possible that a process exceeds its deadline while the partition in which it executes is inactive. This violation will only be detected when the partition is being dispatched, just before the PAL component invokes the POS process scheduler, as shown in the diagram of Figure 2.

The use of offline tools that verify the fulfilment of timing requirements [20, 8], should rule out deadline violations due to faulty system planning (e.g., time windows not satisfying the partitions' timing requirements). However, such tools cannot cope with process deadline violations caused by a runtime malfunction, by transient overload (e. g., due to abnormally high event occurrence rates), or by the underestimation of a process's worst case execution time (WCET) at system configuration and integration time.

4. MECHANISMS FOR NON-INTRUSIVE RUNTIME VERIFICATION

Runtime verification (RV) obtains and analyses data from the execution of a system to detect and possibly react to behaviours, either satisfying or violating the system specification. RV implies that small components, which are not part of the functional system, acting as *observers*, are added to monitor and assess the state of the system in runtime.

The usage of reconfigurable logic supporting versatile platform designs (e.g., soft-processors) enables innovative approaches to RV [17], herein explored in the context of TSP systems. An enhanced AIR architecture makes use of an *AIR Observer* (AO) featuring: *non-intrusiveness*, meaning system operation is not adversely affected and code instrumentation with RV probes is not required; *configurable*, being able to accommodate different event observations.

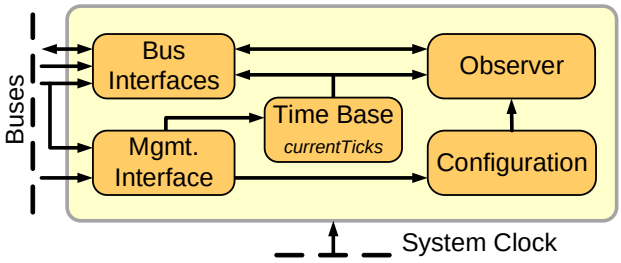

Figure 4: AIR Observer architecture

The AO is plugged to the platform where the AIR software components execute, and comprises the modules depicted in Figure 4: Bus Interfaces, capturing all physical bus activity, such as bus transfers or interrupts; Management Interface, enabling AO configuration; Configuration, storing the patterns of the events to be detected; Observer, detecting events of interest based on the registered configurations.

Though RV concepts can be applied to both time and space partitioning, this paper is restricted to temporal issues. Thus, it is assumed that a **robust time base**[1] accounts for, in the AO hardware (Figure 4), the number of POS-level clock ticks elapsed so far, to which AIR components have access, through the read only *currentTicks* variable/register.

5. INTEGRATING ADAPTABILITY AND NON-INTRUSIVE RUNTIME VERIFICATION

The integration of RV features in the AIR architecture uses a dual approach, as follows:

- operation enforced in hardware, either totally or with some degree of assistance from software components, being the runtime verification actions performed in software;
- operation achieved through the execution of software components, with runtime verification actions enforced in hardware.

5.1 Mode-based schedules

In the generic and highly flexible AIR architecture design presented in [20], the handling of mode-based schedules is entirely integrated within a software-based AIR Partition Scheduler, as illustrated in the diagram of Figure 2.

In a hardware-assisted approach, partition scheduling switch decisions from the AO hardware are complemented with software RV and partition switch actions: when a partition

[1]The design and engineering AIR robust timers is out of the scope of this paper. It will be addressed in a future work.

Algorithm 1 AIR Partition Scheduler with Runtime Verification featuring adaptation through mode-based schedules

```
1: ▷ Entered upon exception: partition preemption point detected
2: ▷ Runtime verification actions
3: if schedules_currentSchedule.table_tableIterator.tick ≠
       (currentTicks − lastScheduleSwitch) mod
       schedules_currentSchedule.mtf then
4:     HealthMonitor(activePartition)
5: else  ▷ Partition switch actions
6:     if currentSchedule ≠ nextSchedule ∧
           (currentTicks − lastScheduleSwitch) mod
           schedules_currentSchedule.mtf = 0 then
7:         currentSchedule ← nextSchedule
8:         lastScheduleSwitch ← currentTicks
9:         tableIterator ← 0
10:    end if
11:    heirPartition ←
       schedules_currentSchedule.table_tableIterator.partition
12:    tableIterator ← (tableIterator + 1) mod
       schedules_currentSchedule.numberPartitionPreemptionPoints
13: end if
```

is dispatched, the absolute value (in POS-level clock ticks) of its *partition preemption point* is inserted in the AO configuration; when this instant is reached, an AO's hardware exception triggers the execution of Algorithm 1.

The RV actions of **Algorithm 1** check, from the active PST, if the current instant is a partition preemption point (line 3). If that is not the case, a severe system level error has occurred and the HM is notified (line 4) to handle the situation. The remaining lines (6-12) implement the partition switch actions of [20], checking (line 6) if there is a pending scheduling switch to be applied and the current instant is the end of the MTF. If these conditions apply, a different PST will be used henceforth (line 7). The processing resources are assigned to the heir partition, obtained (line 11) from the PST in use, until the next partition preemption point. The AIR Partition Scheduler is set (line 12) to access the heir partition parameters.

This hardware/software co-design allows to maintain some degree of AIR architectural flexibility with advantages in terms of improved safety and timeliness. This is particularly useful for running AIR in platforms integrating processor cores (e.g., dual-core ARM) and FPGA logic [10].

The partition switch actions are followed by the execution of the AIR Partition Dispatcher specified in **Algorithm 2**. Two significant differences from [20] do exist: suppression of specific elapsed clock ticks setting, which are no longer required because the partition dispatcher is always invoked after a partition switch; insertion of the next partition preemption point in the AO configuration (line 6). The remaining actions in Algorithm 2 are related to saving and restoring the execution context (lines 2 and 7) and evaluation of the elapsed clock ticks (line 4). Line 8 enforces the execution of pending actions the first time after a PST change the partition is executed [20].

Compared with the equivalent specification in [20], the design of **Algorithm 3** was greatly simplified since all the actions concerning process deadline violation monitoring were removed. The needed actions are now restricted to the signalling of the elapsed clock ticks (line 2) and to the instantiation of the POS native process scheduler (line 4).

Algorithm 2 AIR Partition Dispatcher

```
1: ▷ Entered from the AIR Partition Scheduler after partition switch
   actions
2: SaveContext(activePartition.context)
3: activePartition.lastTick ← currentTicks − 1
4: elapsedTicks ← currentTicks − heirPartition.lastTick
5: activePartition ← heirPartition
6: ReplacePreemptionPoint(heirPartition.tick)
7: RestoreContext(heirPartition.context)
8: PendingScheduleChangeAction(heirPartition)
```

Algorithm 3 AIR PAL – pre POS process scheduler

```
1: ▷ Executed immediately after AIR Partition Dispatcher and
   at every POS-level clock tick
2: PAL_ClockTickAnnounce(elapsedTicks)
3: elapsedTicks = 1
4: POS_ProcessScheduler()
```

5.2 Process deadline violation monitoring

Process deadline violation monitoring, a RV action, was made non-intrusive in the AIR hardware-assisted design. Each process issues, when required, system calls through the APEX interface. For those listed in Table 1, AIR PAL encapsulation provides the registering of the process' deadline in the AO (updating the process' entry, or creating a new one, if not configured yet) or its unregistering (removing the process' entry from the AO configuration). If a process' deadline instant is reached, the AO detects the timeliness violation and issues a hardware exception that once caught activates the process level event handler defined by the application programmer, as illustrated in Figure 3. The APEX primitive RAISE_APPLICATION_ERROR is used for that purpose, with PAL encapsulating a software-based RV action confirming the process deadline violation.

Table 1: APEX primitives needing access to the AO to support process deadline violation monitoring

Primitive	Short description
Need to register/update deadline in the AO	
[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 from the AO	
STOP[_SELF]	Stop a process [itself]

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. A system-wide reconfigurability logic should be included in FDIR [7, 20].

5.3 Analysis and discussion

Critical software, namely that developed to go aboard an aerial or space vehicle, goes through a strict process of verification, validation and certification.

Code complexity affects the effort required for that process, being one relevant metric for code complexity its size, in lines of source code. Towards the usage of standardized accounting methods one employ the *logical source lines of code (logical SLOC)* metric of the Unified CodeCount tool [15].

The C implementation of fundamental AIR software components is assessed in Table 2, which shows its logical SLOC count along with the entity instantiating the component, and implicitly, the instantiation frequency.

Most AIR software components have linear complexity, $\mathcal{O}(1)$: accesses to multielement structures are made by index, being independent of the number and position of the elements. The exception concern process deadline verification in the software-based approach, which in the worst case wields $\mathcal{O}(n)$, being n the number of processes in the partition.

Similar considerations apply to timing issues. The AIR observer and the non-intrusive hardware-assisted approach has reduced the number and code complexity of software components in the path of POS-level clock tick processing. This is specially true for Algorithm 3 and its software-based counterpart code complexity and worst case timing, though in the normal and most frequent case where no process deadlines occur, both components exhibit similar execution times.

Comparing the normalised processing time overheads of AIR Partition Scheduler and Dispatcher ($\mathcal{T}_{SD}$), in the software-based and hardware-assisted approaches, along a full normalised MTF period ($\mathcal{T}_{MTF}$):

$$\upsilon = \frac{\mathcal{T}_{SD\_Soft} - \mathcal{T}_{SD\_Hard}}{\mathcal{T}_{MTF}} \quad (1)$$

$$\approx \frac{\mathcal{T}_{SD\_Soft}}{\mathcal{T}_{tick}} - \frac{\mathcal{T}_{SD\_Hard}}{\mathcal{T}_{MTF}} \cdot n_{ppp} \quad (2)$$

where, n_{ppp} is the number of partition preemption points in the MTF and $\mathcal{T}_{tick}$ is the normalised POS-level clock tick. The normalisation of timing parameters in Figure 5 take the experimental values $\mathcal{T}_{SD\_Soft} = 150\ ns$ and $\mathcal{T}_{tick} = 1\ ms$ as references, making $\mathcal{T}_{SD\_Hard} \approx \mathcal{T}_{SD\_Soft}$ for hardware-assisted and $\mathcal{T}_{SD\_Hard} = 0$ for a full hardware implementation of the AIR Partition Scheduler/Dispatcher.

The difference to software-based processing overheads (υ) in function of MTF duration is represented in Figure 5. For the full hardware implementation, that difference is independent from the MTF value being, in any case, upper bounded by the $\mathcal{T}_{SD\_Soft}/\mathcal{T}_{tick}$ ratio, which typically have quite small values due to the efficient coding of AIR Partition Scheduler and Dispatcher components. For the hardware-assisted approach the processing overhead difference is smaller, dependent on the number/frequency of partition preemption points and only asymptotically approaches the $\mathcal{T}_{SD\_Soft}/\mathcal{T}_{tick}$ limit.

Though POS-level hardware-assisted mechanisms are also deemed to benefit partition/process scheduling timeliness and jitter, those issues have not been addressed so far.

6. RELATED WORK

To the best of our knowledge, contemporary approaches to flexible scheduling in TSP systems are restricted to the mode-based scheduling feature of the commercial Wind River VxWorks 653 product [27]. Previous research on other TSP solutions [9] and works on scheduling analysis for avionic systems [14, 11] do not foresee mechanisms for timeliness adaptation. Alternatives to TSP/IMA are compared in [12],

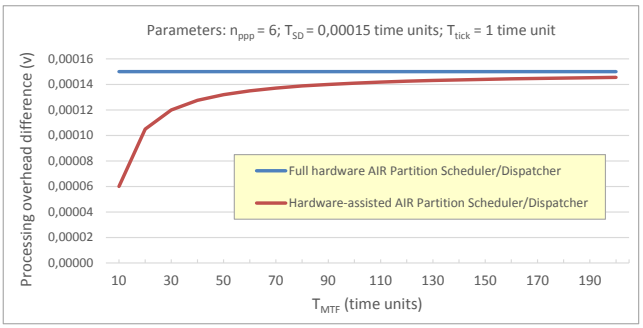

Figure 5: Analysis of AIR Partition Scheduler and Dispatcher processing overheads

which includes recommendations for adaptation of IMA-like architectures. Some results on reconfigurable IMA [4] and UAS adaptive and reconfigurable control [25] do exist.

A hardware/software co-design approach and the concept of system observer is present in the Simplex Architecture [3]. Emergence of non-intrusive runtime verification techniques for embedded systems in general is addressed in [26, 18], while its applicability to complex safety-critical systems is presented in [13]. However, no previous work have applied such techniques to the realm of TSP systems.

7. CONCLUSION

This paper addressed fundamental mechanisms providing support for adaptive and self-adaptive behaviour to applications based on the AIR architecture for time- and space-partitioned systems. The usage of hybrid platforms combining processor cores and programmable logic makes advantageous the use of a hardware-assisted design complemented with some simple software-based components. The computational cost of such components decreases and the non-intrusive runtime verification of the system enables improvements in both safety and timeliness properties.

Non-intrusive runtime verification is a relevant contribution with respect to verification, validation and certification efforts of TSP systems that will be extended in future research. Additional works aim to taking advantage of multicore platforms in AIR [6], which include adaptation/reconfiguration features and, in the near future, RV capabilities.

8. REFERENCES

[1] AEEC (Airlines Electronic Engineering Committee). *Avionics Application Software Standard Interface, Part 1 - Required Services*, Mar. 2006.

[2] AEEC (Airlines Electronic Engineering Committee). *Avionics Application Software Standard Interface, Part 2 - Extended Services*, Dec. 2008.

[3] S. Bak, D. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. The System-level Simplex Architecture for improved real-time embedded system safety. In *15th IEEE Real-Time and Embedded Tech. and Applications Symposium*, pages 99–107, Apr. 2009.

[4] P. Bieber, E. Noulard, C. Pagetti, T. Planche, and F. Vialard. Preliminary design of future reconfigurable IMA platforms. In *Second Int. Workshop on Adaptive*

Table 2: Logical SLOC metrics and instantiation entities for fundamental AIR software components

	Software-based approach		Hardware-assisted approach	
	Logical SLOC	**Instantiation**	**Logical SLOC**	**Instantiation**
AIR Partition Scheduler [a]	13	POS-level clock tick	-	-
AIR RV Partition Scheduler [b]	-	-	12	partition preemption point
AIR Dispatcher [c]	10	POS-level clock tick	8	partition preemption point
AIR PAL – register deadline	34	APEX call	4	APEX call
AIR PAL – unregister deadline	12	APEX call	6	APEX call
AIR PAL – Pre POS Process Scheduler [d]	16	POS-level clock tick	4	POS-level clock tick
POS-level clock tick ISR	>190 [e]	POS-level clock tick	>190	POS-level clock tick

[a] Specified and analysed in [20, 7]
[b] Specified in Algorithm 1
[c] Specified in Algorithm 2
[d] Specified in Algorithm 3; software-based approach specified and analysed in [20, 7]
[e] RTEMS 4.9 [16] C code only; plus >182 assembly instructions in the POS-level clock interrupt service routine (ISR)

and Reconfigurable Embedded Systems, pages 21–24, Grenoble, France, Oct. 2009.

[5] J. Carraca, R. C. Pinto, J. P. Craveiro, and J. Rufino. Information security in time- and space-partitioned architectures for aerospace systems. In *Proc. 6th Simpósio de Informática (INForum 2014)*, pages 457–472, Porto, Portugal, Sept. 2014.

[6] J. P. Craveiro. *Real-Time Scheduling in Multicore Time- and Space-Partitioned Architectures*. PhD thesis, Universidade de Lisboa, Portugal, Aug. 2013.

[7] J. P. Craveiro and J. Rufino. Adaptability support in time- and space-partitioned aerospace systems. In *Proc. 2nd Int. Conf. on Adaptive and Self-adaptive Systems and Applic.*, Lisbon, Portugal, Nov. 2010.

[8] J. P. 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, Sept. 2010.

[9] A. Crespo, I. Ripoll, and M. Masmano. Partitioned embedded architecture based on hypervisor: the XtratuM approach. In *Proc. 8th European Dependable Computing Conf.*, Valencia, Spain, Apr. 2010.

[10] DILIGENT. *ZYBO Reference Manual*, Feb. 2014.

[11] 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.

[12] 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, UK, Apr. 2009.

[13] A. Kane. *Runtime Monitoring for Safety-Critical Embedded Systems*. PhD thesis, Carnegie Mellon University, USA, Feb. 2015.

[14] 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*, pages 103–109, Hiroshima, Japan, 1998.

[15] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm. A SLOC counting standard. In *The 22nd Int. Ann. Forum on COCOMO and Systems/Software Cost Modelling*, Los Angeles, USA, 2007.

[16] On-Line Applications Research Corporation. *RTEMS C User's Guide*, 4.9.4 edition, 2010.

[17] R. C. Pinto and J. Rufino. Towards non-invasive run-time verification of real-time systems. In *26th Euromicro Conf. on Real-Time Systems - WIP Session*, pages 25–28, Madrid, Spain, July 2014.

[18] T. Reinbacher, M. Fugger, and J. Brauer. Runtime verification of embedded real-time systems. *Formal Methods in System Design*, 24(3):203–239, 2014.

[19] J. Rosa, J. P. Craveiro, and J. Rufino. Safe online reconfiguration of time- and space-partitioned systems. In *Proc. 9th IEEE Int. Conf. on Industrial Informatics (INDIN 2011)*, Caparica, Lisbon, Portugal, July 2011.

[20] J. Rufino, J. Craveiro, and P. Verissimo. Architecting robustness and timeliness in a new generation of aerospace systems. In A. Casimiro, R. de Lemos, and C. Gacek, editors, *Architecting Dependable Systems VII*, volume 6420 of *LNCS*. Springer, 2010.

[21] J. Rushby. Partitioning in avionics architectures: Requirements, mechanisms and assurance. Technical Report NASA CR-1999-209347, SRI International, California, USA, June 1999.

[22] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor. A portable ARINC 653 standard interface. In *Proc. 27th Digital Avionics Systems Conf.*, St. Paul, MN, USA, Oct. 2008.

[23] M. Tafazoli. A study of on-orbit spacecraft failures. *Acta Astronautica*, 64(2-3):195–205, 2009.

[24] TSP Working Group. Avionics time and space partitioning user needs. Technical Note TEC-SW/09-247/JW, ESA, Aug. 2009.

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

[26] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *Software, IET*, 1(5):172–179, October 2007.

[27] Wind River. Wind River VxWorks 653 Platform 2.4 and 2.5, 2015. Retrieved Jun 29, 2015.