Architecting Robustness and Timeliness in a New Generation of Aerospace Systems*

José Rufino, João Craveiro, and Paulo Verissimo

University of Lisbon, Faculty of Sciences, LaSIGE
Campo Grande, 1749-016 Lisboa, Portugal
ruf@di.fc.ul.pt, jcraveiro@lasige.di.fc.ul.pt, pjv@di.fc.ul.pt

Abstract. Aerospace systems have strict dependability and real-time requirements, as well as a need for flexible resource reallocation and reduced size, weight and power consumption. To cope with these issues, while still maintaining safety and fault containment properties, temporal and spatial partitioning (TSP) principles are employed. In a TSP system, the various onboard functions (avionics, payload) are integrated in a shared computing platform, however being logically separated into partitions. Robust temporal and spatial partitioning means that partitions do not mutually interfere in terms of fulfilment of real-time and addressing space encapsulation requirements. This chapter describes in detail the foundations of an architecture for robust TSP aiming a new generation of spaceborne systems, including advanced dependability and timeliness adaptation/control mechanisms. A formal system model which allows verification of integrator-defined system parameters is defined, and a prototype implementation demonstrating the current state of the art is presented.

1 Introduction

Aerospace systems, namely the onboard computing infrastructure, have strict requirements with respect to dependability and real-time, as well as a need for flexible resource reallocation and reduction of size, weight and power consumption (SWaP). A typical spacecraft onboard computer has to host a set of avionics functions and one or more payload subsystems [13]. Relevant examples of avionics functions are the Attitude and Orbit Control Subsystem (AOCS), Onboard Data Handling (OBDH), Telemetry, Tracking, and Command (TTC) subsystem, and Fault Detection, Isolation and Recovery (FDIR).

Traditionally, dedicated hardware resources have been separately allocated to those functions. However, there has been a recent trend in the aerospace industry towards integrating several functions in the same computing platform. This

* This work was partially developed within the scope of the ESA (European Space Agency) Innovation Triangle Initiative program, through ESTEC Contract 21217/07/NL/CB, Project AIR-II (ARINC 653 in Space RTOS – Industrial Initiative, http://air.di.fc.ul.pt). This work was partially supported by Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology), through the Multiannual Funding and CMU-Portugal Programs and the Individual Doctoral Grant SFRH/BD/60193/2009.

is advantageous in respect to SWaP requirements, but has introduced potential risks, as these functions may have different degrees of criticality and predictability, and originate from multiple providers or development teams [25,30].

In order to mitigate these risks, an architectural principle was proposed whereby onboard applications are functionally separated in logical containers, called partitions. With partitioning we achieve two results: allowing containment of faults in the domain in which they occur; and enabling independent software verification and validation, thus easing the overall certification process. Partitioning in logical containers implies separation of applications' execution in the time domain and usage of dedicated memory and input/output addressing spaces. Robust *temporal and spatial partitioning* (TSP) means that partitions do not interfere with each other in terms of fulfilment of *real-time* and *addressing space encapsulation* requirements.

This chapter describes in detail the foundations and genesis of an *architecture for robust TSP* aiming at a new generation of spaceborne systems. Firstly, the basic architecture is detailed together with the advanced features introduced in its design. A second result described in the chapter is the introduction of a couple of advanced timeliness adaptation and control mechanisms, crucial to the provision of high degrees of dependability in TSP systems: *mode-based partition schedules* (allowing the temporal requirements of the installed functions to vary according to the mission's phase or mode of operation) and *process deadline violation monitoring* (providing fundamental health monitoring services with enhanced diagnostics support). Thirdly, a *formal system model* is defined. This model and associated tools ease the verifiability of systems based on the architecture. It allows for the verification of the integrator-defined system parameters, such as partition scheduling according to the respective temporal requirements, and lays the ground for schedulability analysis and automated aids to the definition of system parameters. Temporal analysis in TSP systems has not been addressed in the literature to the full extent needed to aid design, integration and deployment of modern TSP systems in space [32].

Our research has been motivated by the challenge launched by several space industry partners, such as the National Aeronautics and Space Administration (NASA) [24] and the European Space Agency (ESA) [29], for applying TSP concepts to computing resources onboard spacecrafts, while observing compliance with existing standards such as the ARINC 653 specification [2]. ARINC 653 defines a standard interface for avionics application software to interact with the underlying core software (operating system). This standard is tightly connected to the Integrated Modular Avionics (IMA) concept, which applies the TSP notion to the civil aviation world [1]. The TSP Working Group, comprising space agencies ESA and CNES (the French government space agency) and industry partners Astrium and Thales Alenia Space, has identified the specific requirements for the adoption of IMA concepts in space, and found no technological feasibility impairments to it [32].

In the wake of space agencies' interest, what originally started as a proof of concept for the addition of TSP-capabilities to a free/opensource real-time op-

erating system, the Real-Time Executive for Multiprocessor Systems (RTEMS), evolved into the definition of a more ambitious and innovative architecture in the context of the AIR (ARINC 653 In Space Real-time Operating System) project [11,23,22]. The AIR architecture has been designed to fulfil the requirements for robust TSP, and foresees the use of different operating systems among the partitions, either real-time operating systems (RTOS) or generic non-real-time ones. Temporal partitioning is achieved through the scheduling of partitions in a cyclic sequence of fixed time slices. Inside each partition, processes compete with each other according to the native process scheduler of the partition. In the case of RTOSs, this is usually a dynamic priority-based scheduler.

The chapter is organized as follows. Section 2 describes the AIR system architecture. Then, Sect. 3 contains the formal definition of a generic model for ARINC 653-based systems with regard to their temporal properties and requirements. Sections 4 and 5 describe the principles and implications of introducing, respectively, mode-based schedules and process deadline violation monitoring into AIR. Sect. 6 demonstrates the properties of the system model and the enhancements described in the previous sections by means of a prototype implementation. Section 7 presents related work and Sect. 8 concludes the chapter.

2 AIR System Architecture

The AIR (ARINC 653 in Space RTOS) architecture is currently evolving towards an industrial product definition by improving and completing the key points identified in early proof-of-concept activities [22]. The fundamental idea in the definition of the AIR architecture is a simple solution for providing robust TSP properties, thus guaranteeing the fulfilment of real-time and dedicated memory and input/output addressing space separation requirements. Faults are confined to their domain of occurrence inside each partition. AIR provides the ARINC 653 functionality missing in off-the-shelf (real-time) operating system kernels, as illustrated in the diagram of Fig. 1, encapsulating those functions in special-purpose additional components with well-defined interfaces. In essence, the AIR architecture preserves the hardware and operating system independence defined in the ARINC 653 specification [2]. AIR foresees the possibility that each partition runs a different operating system, henceforth called *Partition Operating System* (POS) [23].

Applications may use a strict ARINC 653 service interface, the *Application Executive* (APEX) interface or, in the case of system partitions, may bypass this standard interface and use additional functions from the POS kernel, as illustrated in Fig. 1. The existence of system partitions with the possibility of bypassing the APEX interface is a requirement of the ARINC 653 specification. It should noted though that these partitions will typically run system administration and management functions, performed by applications which will be subject to increased verification efforts.

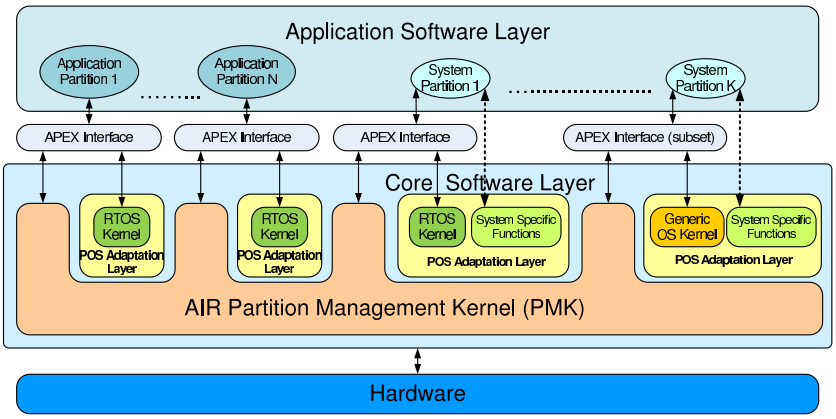

Fig. 1. AIR system architecture

A (system) application, and the given APEX interface, POS and *AIR POS Adaptation Layer* (PAL) instances compose the containment domain of each partition.

2.1 AIR Partition Management Kernel (PMK)

The *AIR Partition Management Kernel* (PMK) component, transversal to the whole system (see Fig. 1), could be seen as a hypervisor, playing nevertheless a major role in achieving dependability, by ensuring robust TSP.

Temporal Partitioning

Temporal partitioning concerns partitions not interfering with each other's timeliness. In AIR this is guaranteed by a two-level hierarchical scheduling scheme; partitions are scheduled cyclically under a fixed schedule, and processes are scheduled by the native scheduler of the operating system of the partition in which they are executing, as shown in Fig. 2. The partition schedule is repeated cyclically and covers a time interval denominated *major time frame* (MTF).

The AIR PMK integrates a *Partition Scheduler* and a *Partition Dispatcher* which implement the first level of this hierarchical scheduling scheme. At each clock tick, the Partition Scheduler consults a scheduling table to detect if a partition preemption point has been reached. If that is the case, the Partition Dispatcher is called upon to perform the context switch between the active partition (which currently holds the processing resources) and the heir partition (which will hold the processing resources until the next partition preemption point). The advanced mechanisms represented in Fig. 2 (mode-based schedules and process deadline violation monitoring) are detailed in Sects. 4 and 5.

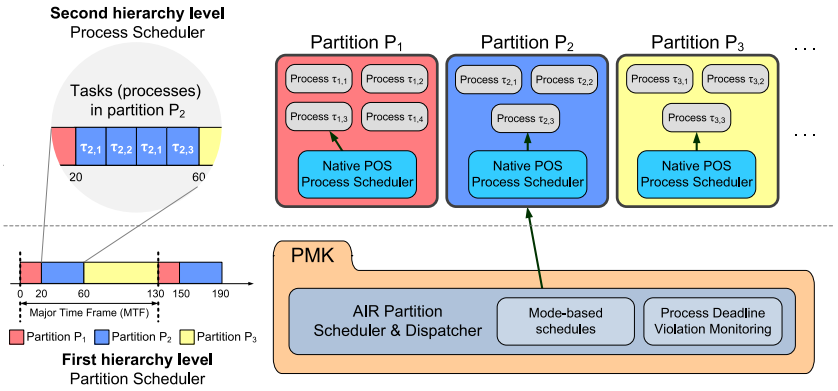

Fig. 2. AIR two-level hierarchical scheduling

Spatial Partitioning

Spatial partitioning means that applications running in one partition cannot access addressing spaces outside those belonging to that partition [24,27]. For the support thereto, AIR follows a highly modular design approach illustrated in Fig. 3. Spatial partitioning requirements (specified in AIR and ARINC 653 configuration files with the assistance of development tools support) are described in runtime through a high-level processor-independent abstraction layer. A set of descriptors is provided per partition, primarily corresponding to the several levels of execution (e.g. application, operating system and AIR PMK) and to its different memory sections (e.g. code, data and stack) [22].

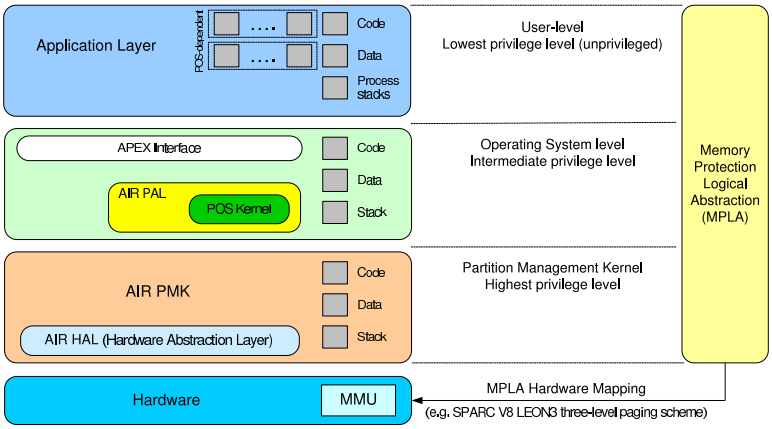

Fig. 3. AIR spatial partitioning mechanisms

The high-level abstract spatial partitioning description needs to be mapped in runtime to the specific processor memory protection mechanisms, exploiting the availability of a hardware Memory Management Unit (MMU), as shown in the lowest layer of Fig. 3. An example of such mapping is the Gaisler SPARC V8 LEON3 three-level page-based MMU core [28].

Interpartition Communication

Notwithstanding spatial partitioning requirements, typical spacecraft partitioned onboard applications need to exchange data. For example, some payload subsystems may need to read AOCS data or transmit data to FDIR. Thus, AIR PMK provides low-level mechanisms for interpartition communication. Applications access the interpartition communication services through the APEX interface (Sect. 2.3), in a way which is agnostic of whether the partitions are local or remote to one another and how they communicate. The AIR PMK deals with these specifics, being obliged to message delivery guarantees. For physically separated partitions, this implies data transmission through a communication infrastructure. In the case of partitions in the same processing platform, interpartition communication is implemented through memory-to-memory copies not violating spatial separation requirements [22].

2.2 AIR POS Adaptation Layer (PAL)

The AIR POS Adaptation Layer (PAL) plays an important role in the AIR architecture, in the sense it wraps each partition's operating system, hiding its particularities from the AIR architecture components. This allows for a more flexible and homogeneous integration of support to new partition operating systems (or new versions thereof) and a better software development process (supported by separation of concerns and stronger validation and certification processes) [22,8].

2.3 Flexible Portable APEX Interface

The APEX interface provides to the applications a set of services, defined in the ARINC 653 specification [2]. AIR employs an innovative implementation of APEX which consists of two components: the APEX Core Layer and the APEX Layer Interface.

The APEX Core Layer implements the advanced notion of *Portable APEX* intended to ensure portability between the different POSs supported by AIR [26]. The AIR APEX fully exploits the the availability of AIR PAL-related functions, and the POSIX application programming interface currently available on most (RT)OSs [15]. An optimized implementation may invoke directly the native (RT)OS service primitives. The AIR APEX also coordinates, when required, the interactions with the AIR Health Monitor, e.g. upon detection of an error [22].

2.4 AIR Health Monitoring (HM)

The AIR Health Monitor is responsible for handling hardware and software errors (like deadlines missed, memory protection violations, or hardware failures). The aim is to isolate errors within its domain of occurrence: process level errors will cause an application error handler to be invoked, while partition level errors trigger a response action defined at system integration time. Errors detected at system level may lead the entire system to be stopped or reinitialized [22].

2.5 Integration of Generic Operating Systems

The foreseen heterogeneity between POSs is also being extended to include generic non-real-time systems, such as Linux, answering to a recent trend in the aerospace industry. The coexistance of real-time and non-real-time POSs is motivated by the lack of relevant functions in most RTOSs, which are commonly provided by generic non-real-time operating systems. Furthermore, porting these functions (e.g. scripting language interpreters) to RTOSs can be a complicated and error-prone task [16]. An embedded variant of Linux has been approached, and yields a fully functional operating system with a minimal size compatible with the coexistence with other POSs and typical space missions requirements [8,9].

To ensure that a non-real-time kernel as Linux cannot undermine the overall time guarantees of the system by disabling or diverting system clock interrupts, the instructions that could allow this must be wrapped by low-level handlers (paravirtualized) [8].

3 System Model

To allow for formal verification of properties and requirements, the AIR architecture can be modeled as follows. The model presented here focuses on the temporal aspects of the system, which are the most relevant for the contributions of this chapter. This system model is generic enough that it can possibly apply to other TSP systems, especially those based on the ARINC 653 specification [2]. A simplified version of the system model can also apply to hypervisor-based systems in general.

The notation used in this chapter has been chosen so as to follow recent efforts towards a common notation among the research community [10]. To that purpose, symbols to denote the notions of the system model coincide with those used in previous works in the area [10,18]. Symbols for new concepts try, as much as possible, not to conflict with those already widely used in the literature for different concepts. A reference table for this notation, as applicable to the system model resulting from the work presented in this chapter, is presented in the Appendix.

3.1 Partitions

An AIR-based system, or actually a generic ARINC 653-based system, is composed by a set of partitions, P:

$$P = \{P_1, P_2, \ldots, P_{n(P)}\} \ . \tag{1}$$

Each partition P_m is defined as:

$$P_m = \langle \eta_m, d_m, \tau_m, M_m(t) \rangle \tag{2}$$

where η_m is the partition's activation cycle, d_m is its assigned duration (the amount of time to be given to the partition per cycle), and τ_m is the set of processes running inside the partition (these will be covered in detail in Sect. 3.3). $M_m(t)$ is the operating mode of the partition P_m at the instant t, such that:

$$M_m(t) \in \{\mathsf{normal}, \mathsf{idle}, \mathsf{coldStart}, \mathsf{warmStart}\} \ . \tag{3}$$

In the normal mode, the partition is effectively operational, with its process scheduler active, while the idle mode corresponds to a shut-down partition not executing any processes. The coldStart and warmStart modes both indicate that the partition is initializing (with process scheduling disabled), differing from one another regarding the initial context [2].

This model is very flexible, supporting partitions with either an inherently periodic or aperiodic behaviour. As seen in Sect. 2.1, the partition schedule repeats over a major time frame (MTF); thus, a partition which is not by itself periodic can be modeled as having a partition cycle equal to the duration of the MTF. Partitions which do not have strict time requirements, such as those running non-real time operating systems, are also covered, having $d_m = 0$.

3.2 Partition Scheduling

Partitions are scheduled on a fixed cyclic basis in the first of the two levels of the hierarchical scheduling scheme, as illustrated in Fig. 2. The time interval covered by a partition schedule, and over which it repeats, is called the major time frame (MTF). The partition scheduling table (PST) for a system, χ, can be defined as:

$$\chi = \langle MTF, \omega = \{\omega_1, \omega_2, \ldots, \omega_{n(\omega)}\} \rangle \ . \tag{4}$$

ω is a set of time windows, each one defined as:

$$\omega_j = \langle P_j^\omega, O_j, c_j \rangle \qquad P_j^\omega \in P \tag{5}$$

where P_j^ω is the partition scheduled to be active during the jth time window, O_j is the window's offset (relative to the beginning of a major time frame) and c_j is its duration. At this stage, we assume that every partition in P has, at least, one time window, thus $\bigcup_{\omega_j \in \omega} P_j^\omega = P$. Time windows do not intersect and are fully contained within one MTF, so:

$$\begin{cases} O_j + c_j \leq O_{j+1} & \forall j < n(\omega) \\ O_{n(\omega)} + c_{n(\omega)} \leq MTF \ . \end{cases} \tag{6}$$

As a necessary but not sufficient condition for system-wide schedulability, the MTF should be a multiple of the least common multiple (lcm) of all the partitions' cycles:

$$MTF = k \times \operatorname*{lcm}_{\forall P_m \in P} (\eta_m) \qquad k \in \mathbb{N} \ . \tag{7}$$

and the sum of each partition's time windows should account for the duration defined for that partition:

$$\sum_{\{\, \omega_j \in \omega \mid P_j^\omega = P_m \,\}} c_j \geq d_m \frac{MTF}{\eta_m} \qquad \forall P_m \in P \ . \tag{8}$$

This is nevertheless not a sufficient condition for compliance with the partitions' temporal requirements. Besides respecting (8), the time windows must guarantee that, if a partition completes more than one cycle inside the MTF, it executes the assigned duration within each of these cycles:

$$\sum_{\substack{\{\, \omega_j \in \omega \mid P_j^\omega = P_m \wedge \\ O_j \in [k\,\eta_m ; (k+1)\eta_m [\ \}}} c_j \geq d_m \qquad \forall P_m \in P \,,\ \forall k \in \left[0..\frac{MTF}{\eta_m} - 1 \right] \ . \tag{9}$$

If the condition expressed in (9) holds, each partition P_m has at least d_m time units assigned in each of the $\frac{MTF}{\eta_m}$ cycles completed inside one MTF. Thus (8) will also hold — the sum of all the time windows inside one MTF will be no less than $d_m \frac{MTF}{\eta_m}$.

3.3 Processes

In AIR (and ARINC 653), the scope of process management is restricted to its partition. As defined in (2), each partition $P_m \in P$ contains a set of processes:

$$\tau_m = \{\tau_{m,1}, \tau_{m,2}, \ldots, \tau_{m,n(\tau_m)}\} \ . \tag{10}$$

Each process $\tau_{m,q}$ can be defined as:

$$\tau_{m,q} = \langle T_{m,q}, D_{m,q}, p_{m,q}, C_{m,q}, S_{m,q}(t) \rangle \tag{11}$$

where $T_{m,q}$ is the process's period, $D_{m,q}$ its relative deadline, $p_{m,q}$ its *base* priority, and $S_{m,q}(t)$ represents the status of the process at instant t. If the process $\tau_{m,q}$ is aperiodic or sporadic, $T_{m,q}$ represents the lower bound for the time between consecutive activations. If $D_{m,q} = \infty$, then $\tau_{m,q}$ has no deadlines. The worst case execution time (WCET), $C_{m,q}$, is not originally a process attribute in the ARINC 653 specification. It is though added to the system model, since it is essential for further scheduling analyses.

The status of the process:

$$S_{m,q}(t) = \langle D'_{m,q}(t), p'_{m,q}(t), St_{m,q}(t) \rangle \tag{12}$$

includes the process's absolute deadline time, $D'_{m,q}(t)$, *current* priority, $p'_{m,q}(t)$, and state:

$$St_{m,q}(t) \in \{\mathsf{dormant}, \mathsf{ready}, \mathsf{running}, \mathsf{waiting}\} \ . \tag{13}$$

A dormant process is ineligible to receive resources because it either has not been started or has been stopped. A ready process is one which is able to be executed, while a running process (only one at any time) is the one currently executing. A waiting process is not eligible to be scheduled until a certain event for which it is waiting has occurred — a delay, a semaphore, a period, etc. — or another process resumes it (if it has been suspended).

Processes inside each partition compete for processing time during the partition's time windows. In RTOSs, this is usually done according to a preemptive priority-driven scheduling algorithm. The convention here is that lower numerical values represent greater priorities. In normal operation with preemption enabled, the heir process in a given partition at a given moment, $heir_m(t)$, is wielded by:

$$\begin{aligned} heir_m(t) = \tau_{m,h} \in Ready_m(t) \mid (p'_{m,h}(t) < p'_{m,q}(t)) \vee \\ (p'_{m,h}(t) = p'_{m,q}(t) \wedge h < q) \qquad \forall \tau_{m,q} \in Ready_m(t), q \neq h \end{aligned} \tag{14}$$

where:

$$Ready_m(t) = \{\tau_{m,q} \in \tau_m \mid St_{m,q}(t) \in \{\mathsf{ready}, \mathsf{running}\}\} \ . \tag{15}$$

Processes are assumed to be sorted in decreasing order of antiquity in the ready state. This reads that the process selected to be executed is the highest priority ready (or already running) process in the partition; if more than one process has the highest priority, the oldest one is selected.

4 Mode-Based Schedules

The original AIR Partition Scheduler, integrated in the AIR PMK component (Sect. 2), defines a static scheduling of partitions, cyclically obeying to a Partition Scheduling Table (PST) defined offline, at system integration time (see (4)). The AIR Partition Scheduler verifies whether a partition preemption point has been reached and, in that case, selects the heir partition.

This static scheme is very restricting in terms of configuration flexibility and fault tolerance. The AIR advanced design addresses this issue by introducing support for multiple mode-based partition schedules. Examples of the usefulness of mode-based schedules include the adaptation of partition scheduling to different modes/phases (initialization, operation, etc.) and the accommodation of component failures (e.g., assigning a critical program running in a failed processor to another one). Such a notion is also conveyed in Part 2 of the ARINC 653 specification [3].

The basic mandatory scheduling scheme is extended to allow multiple schedules to be defined at system integration time. At execution time, authorized partitions may request switching between the different PSTs, represented in the rightmost part of Fig. 4.

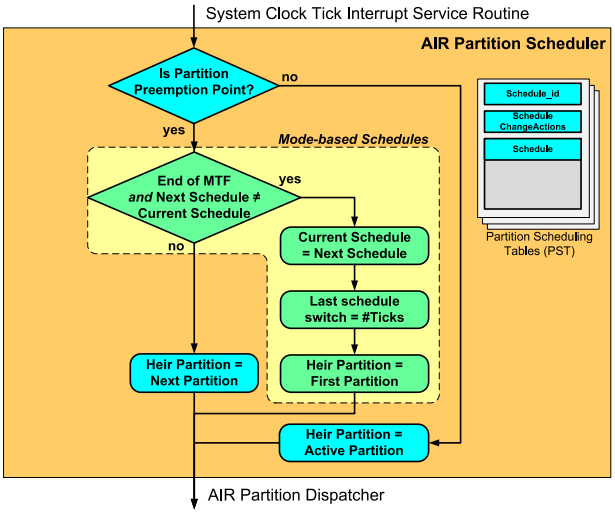

Fig. 4. AIR Partition Scheduler with support for mode-based schedules

To this purpose, the system configuration and integration process is extended in two ways:

1. definition of multiple schedules, with different major time frames, partitions, and respective periods and execution time windows;
2. inclusion of restart actions (ScheduleChangeAction) to be performed, on a per-partition and per-schedule basis, when the schedule is changed.

The AIR Partition Scheduler has to be modified with the functions highlighted (dotted line) in Fig. 4. These modifications concern the verification, at the end of each MTF, of whether a schedule switch is pending and, if that is the case, make the schedule switch effective.

4.1 Implications on the System Model

The introduction of mode-based schedules imposes a reformulation of the system model presented in Sect. 3. Besides the system now having *a set of* partition scheduling tables, the partitions' timing requirements (period and duration) are no longer attributes of the partition, but rather attributes of the partition *in a given schedule*. Schedules may change on account of the mission's mode/phase changing, and this in turn implies that most likely some processes inside partitions need not be always active. Thus, each partition's timing requirements may change from schedule to schedule; the alternative approach of keeping each partition's time requirements constant throughout the schedules by targeting an extremely pessimistic case would lead to a poorly efficient resource utilization through time.

The system is still composed of a set P of partitions (1), which will now be deprived of timing requirements on their own:

$$P_m = \langle \tau_m, M_m(t) \rangle \ . \tag{16}$$

The system also holds, as mentioned, a set of partition scheduling tables:

$$\chi = \{\chi_1, \chi_2, \ldots, \chi_{n(\chi)}\} \tag{17}$$

which definition should be adjusted from (4):

$$\chi_i = \langle MTF_i, Q_i = \{Q_{i,1}, \ldots, Q_{i,n(Q_i)}\}, \omega_i = \{\omega_{i,1}, \ldots, \omega_{i,n(\omega_i)}\} \rangle \tag{18}$$

where:

$$Q_{i,m} = \langle P^{\chi}_{i,m}, \eta_{i,m}, d_{i,m} \rangle \qquad P^{\chi}_{i,m} \in P \ . \tag{19}$$

Since now it may be the case that not all partitions will be present in every schedule, the requirements expressed in (7), (8) and (9) are too strong, as they express requirements for each PST in terms of all partitions in the system (regardless of which partitions are present in each PST).

To reflect the changes expressed in (17), (18) and (19), and the concern over the mentioned requirements being too strong, the system model must be enhanced by replacing (5) to (7) with (20) to (22):

$$\omega_{i,j} = \langle P^{\omega}_{i,j}, O_{i,j}, c_{i,j} \rangle \qquad P^{\omega}_{i,j} \in Q_i \tag{20}$$

$$\begin{cases} O_{i,j} + c_{i,j} \leq O_{i,j+1} & \forall j < n(\omega_i) \\ O_{i,n(\omega_i)} + c_{i,n(\omega_i)} \leq MTF_i \end{cases} \tag{21}$$

$$MTF_i = k_i \times \operatorname*{lcm}_{\forall Q_m \in Q_i} (\eta_m) \qquad k_i \in \mathbb{N} \ . \tag{22}$$

The fundamental timing requirement fulfilment condition expressed in (9) is accordingly updated as can be seen in (23):

$$\sum_{\substack{\{\, \omega_{i,j} \in \omega_i \mid P^{\omega}_{i,j} = P_m \wedge \\ O_{i,j} \in [k\,\eta_m ; (k+1)\eta_m [\ \}}} c_{i,j} \geq d_m \quad \forall i \leq n(\chi) \,, \forall P_m \in Q_i \,, \forall k \in \left[0..\frac{MTF_i}{\eta_m} - 1\right] \ . \tag{23}$$

Since (9) implies (8), a replacement for the latter is not provided. The kind of system initially described, with only one statically defined partition scheduling table, can still be modeled, as a special case of a system with $n(\chi) = 1$.

4.2 Implications on the APEX Interface

Support for mode-based schedules requires the provision of additional APEX services. These should allow setting and obtaining the current partition scheduling parameters.

First and foremost, a service which sets the schedule that will start executing at the top of the next MTF must be provided. It must be invoked by an authorized partition, and have the identifier of an existing schedule as its only parameter. The immediate result is only that of storing the identifier of the next schedule.

The effective schedule switch occurs at the start of the next MTF, by having the AIR Partition Scheduler (see Algorithm 1) perform the following steps:

Line 4: *currentSchedule* is set to *nextSchedule*, which is the identifier stored in the latest previous call to the service.

Line 5: *lastScheduleSwitch* is set to the current time.

Algorithm 1 AIR Partition Scheduler featuring mode-based schedules

```
1: ticks ← ticks + 1                          ▷ ticks is the global system clock tick counter
2: if schedules_currentSchedule.table_tableIterator.tick =
   (ticks − lastScheduleSwitch) mod schedules_currentSchedule.mtf then
3:     if currentSchedule ≠ nextSchedule ∧
       (ticks − lastScheduleSwitch) mod schedules_currentSchedule.mtf = 0 then
4:         currentSchedule ← nextSchedule
5:         lastScheduleSwitch ← ticks
6:         tableIterator ← 0
7:     end if
8:     heirPartition ← schedules_currentSchedule.table_tableIterator.partition
9:     tableIterator ← (tableIterator + 1) mod
       schedules_currentSchedule.numberPartitionPreemptionPoints
10: end if
```

Also, each partition P_m in the new schedule running in normal mode, i.e. $M_m(t) = \mathsf{normal}$, will have to be restarted according to the value defined for its ScheduleChangeAction (which can indicate that no restart should occur). This action takes place the first time each partition is scheduled/dispatched after the schedule switch (not represented in Algorithm 1). Support for the schedule switch makes up for virtually the whole of the changes made to the original AIR Partition Scheduler.

Another service provided in AIR allows obtaining the full current schedule status information, which (in compliance with ARINC 653 Part 2 [3]) comprises:

- the time of the last schedule switch (0 if none ever occurred);
- the identifier of the current schedule;
- the identifier of the next schedule, which will be the same as the current schedule if no schedule change is pending for the end of the present major time frame.

4.3 Design and Engineering Issues

Since the AIR Partition Scheduler code is invoked at every system clock tick, its code needs to be as efficient as possible. In the AIR implementation presented in Algorithm 1, in the best and most frequent case, only two computations are performed:

Line 1: Increment the number of clock ticks by one.
Line 2: Verify if a partition preemption point has been reached (this best case is also the most frequent one, since this verification will turn out false far more often than true).

To incorporate the mode-based schedules functionality, the AIR Partition Scheduler computations had to be extended (see Algorithm 1); verifications of the presence of a partition preemption point (line 2) or the end of a MTF (line 3) need to rely on the number of clock ticks elapsed since the last schedule switch, and not solely the number of clock ticks since system initialization.

The AIR Partition Dispatcher is executed after the Partition Scheduler. Its only modification regarding mode-based schedules is the invocation of pending schedule change actions. The mechanism of the AIR Partition Dispatcher, with this modification highlighted (dotted line), is detailed in Fig. 5.

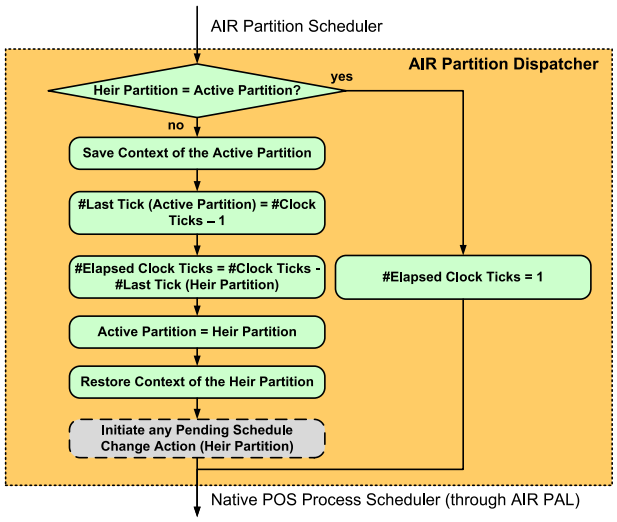

Fig. 5. AIR Partition Dispatcher with support for mode-based schedules

Part 2 of the ARINC 653 specification [3] does not clearly state whether schedule change actions should be performed immediately after effectively changing schedule (i.e., at the beginning of the first MTF under the new schedule, for

all partitions) or performed for each partition as it is dispatched for the first time after the schedule switch. It is nevertheless our understanding that the latter approach is more compliant with the fulfilment of temporal separation requirements, since these will only affect its own execution time window. This is specified in Algorithm 2 (line 9). The remaining actions in Algorithm 2 are related to saving and restoring the execution context (lines 4 and 8), and evaluation of the elapsed clock ticks (lines 2 and 6).

Algorithm 2 AIR Partition Dispatcher featuring mode-based schedules

```
 1: if heirPartition = activePartition then
 2:     elapsedTicks ← 1
 3: else
 4:     SaveContext(activePartition.context)
 5:     activePartition.lastTick ← ticks − 1
 6:     elapsedTicks ← ticks − heirPartition.lastTick
 7:     activePartition ← heirPartition
 8:     RestoreContext(heirPartition.context)
 9:     PendingScheduleChangeAction(heirPartition)
10: end if
```

5 Process Deadline Violation Monitoring

During the execution of the system, it may be the case that a process exceeds its deadline; this can be caused by a malfunction or because that process's WCET was underestimated at system configuration and integration time. Other 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 the timing requirements as expressed in (23).

In addition, it is also possible that a process exceeds a deadline while the partition in which it executes is inactive, and that will only be detected when the partition is being dispatched, just before invoking the process scheduler. The earliest deadline is checked; following deadlines may subsequently be verified until one has not been missed. This can be computationally optimized with the help of an appropriate data structure with the deadlines in ascending order, allowing for $\mathcal{O}(1)$ retrieval of the earliest deadline. This is extremely relevant given deadline verification is performed inside the system clock interrupt service routine (ISR). Furthermore, this methodology is optimal with respect to deadline violation detection latency.

In the context of Health Monitoring (HM), ARINC 653 classifies process deadline violation as a process level error (an error that impacts one or more processes in the partition, or the entire partition) [2,22]. Possible recovery actions in the event of such an error are:

– ignoring the error (logging it, but taking no action);
– logging the error a certain number of times before acting upon it;
– stopping the faulty process, and reinitializing it from the entry address or starting another process;
– stopping the faulty process, assuming that the partition will detect this and recover;
– restarting or stopping the partition.

The actual action to be performed is defined by the application programmer, through an appropriate error handler [22].

5.1 Implications on the System Model

At instant t, the set of processes having violated their deadlines is given by:

$$V(t) = \bigcup_{m=1}^{n(P)} \left\{ \tau_{m,q} \in \tau_m \mid D_{m,q} \neq \infty \wedge D'_{m,q}(t) < t \right\} \tag{24}$$

The $D_{m,q} \neq \infty$ condition translates the fact that the notion of deadline violation does not apply to non-real-time processes.

5.2 Implications on the APEX Interface

The information about processes statuses and deadlines is maintained in such a way that it is conveniently kept updated by the relevant APEX primitives which:

– start a process, making it become able to be executed by initializing all its attributes, setting the runtime stack, and placing it in the ready state;
– start a process with a given delay, by placing it in the waiting state until the requested delay is expired;
– suspend the execution of a (periodic) process until the next release point[1];
– postpone a process's deadline time (replenishment);
– perform a partition shutdown or restart.

Each of these primitives will need to insert or update the due processes' deadlines, while the services which stop a process (putting it in the dormant state, by its own request or from another process) need to remove the due processes' deadline information from the control data structures.

The AIR PAL component provides private interfaces for these APEX services to register/update and unregister deadlines, and keeps the appropriate data structures containing this information. This is the most reasonable implementation, from the engineering, integrity and spatial separation points of view. An example of how the APEX and the AIR PAL for one given partition integrate to provide this functionality is shown in Fig. 6.

[1] A release point of a process is defined in general as the instant the process becomes ready for execution. For a periodic process the consecutive release points will be separated by the respective period.

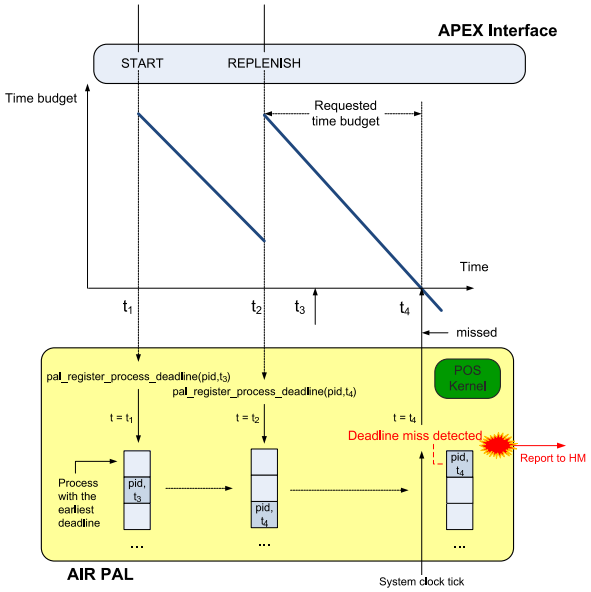

Fig. 6. Integration of the APEX Interface and the AIR PAL to provide process deadline violation detection and reporting

When a process is started, via the START APEX service, its deadline time is set to instant t_3 (obtained by adding the process's time capacity to the current instant), and this value is registered via the AIR PAL-provided interface. Upon a replenishment request (REPLENISH service), a new deadline time, t_4, is calculated (by adding the requested budget time to the current instant). The interface provided by AIR PAL to register a process deadline is again called, to update the information for this process; if necessary, this information will be moved to keep the deadlines sorted by ascending deadline time order. When instant t_4 is reached without the process having finished its execution, a deadline miss has occurred, which is detected and should be reported to partition-wise health monitoring and error handling mechanisms through appropriate primitives.

5.3 Design and Engineering Issues

Figure 7 illustrates the modification to the surrogate clock tick announcement routine provided by the AIR PAL, so as to verify the earliest deadline(s) and report any violations to the health monitoring. Process scheduling and dispatching is ensured by the corresponding native POS mechanisms.

The implementation of process deadline violation monitoring in AIR is shown in Algorithm 3. To keep the computational complexity of the process deadline violation monitoring to a minimum, the information concerning process deadlines is kept at each partition's AIR PAL component, ordered by deadline, and only

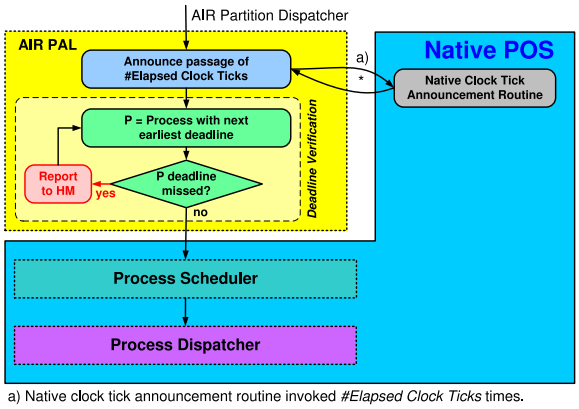

Fig. 7. Modifications on the surrogate clock tick announcement routine to accommodate deadline verification features

the earliest deadline is verified by default; this verification (line 3) happens after announcing the passage of the elapsed clock ticks (line 1). The information on the earliest deadline is retrieved in constant time ($\mathcal{O}(1)$). Only in the presence of deadline violations will more deadlines be checked, in ascending order until reaching one that has not been violated.

Algorithm 3 Deadline verification at the AIR PAL level

```
1: *POS_ClockTickAnnounce(elapsedTicks)
2: for all d ∈ PAL_deadlines do
3:     if d.deadlineTime ≥ PAL_GetCurrentTime( ) then
4:         break
5:     end if
6:     HM_DeadlineViolated(d.pid)              ▷ pid: process identifier
7:     PAL_RemoveProcessDeadline(d)
8: end for
```

Currently, the AIR PAL uses a linked list to keep the process deadline information. In the deadline verification process, a violation may be detected (Algorithm 3, line 3), and after reporting its occurrence to Health Monitoring (line 6) the deadline is removed from the control structure (line 7). Since we already have a pointer to the node to be removed, the complexity of the deadline removal from the linked list will effectively be $\mathcal{O}(1)$, as opposed to the generic $\mathcal{O}(n)$ complexity yielded by linked lists.

A point where the use of a self-balancing binary search tree would theoretically outperform a linked list concern the act of inserting, removing or updating nodes, materialized in the register/unregister deadline interfaces provided to the

APEX — $\mathcal{O}(\log n)$ vs. $\mathcal{O}(n)$. Nevertheless, since these operations do not happen inside the system clock tick ISR, but rather on a partition's execution time window, and also the number of processes accounted for deadline verification will be typically small, such asymptotic advantage will not correlate to effective and/or significant profit, and certainly not compensate for the more critical downside to operations running during an ISR.

6 Prototype Implementation

To demonstrate the advanced timeliness control features, we developed a prototype implementation of an AIR-based system comprising four partitions. Each partition executes an RTEMS-based [21] mockup application representative of typical functions present in a satellite system. The demonstration was implemented for an Intel IA-32 target, and ran on the QEMU emulator.

This sample system is configured with two PSTs, between which it is possible to alternate through the mode-based schedules service described in Sect. 4. These partition scheduling tables are shown in Fig. 8. Both tables repeat over a MTF of 1300 time units, but this is not a strict requirement — it stems from the partitions' timing requirements as per (22). Each partition contains one to three mockup processes, which period is a multiple of the respective partition's cycle duration.

$$
\begin{aligned}
P &= \{P_1, P_2, P_3, P_4\} \\
Q_1 = Q_2 &= \{\langle P_1, 1300, 200\rangle, \langle P_2, 650, 100\rangle, \langle P_3, 650, 100\rangle, \langle P_4, 1300, 100\rangle\} \\
\chi_1 &= \langle MTF_1 = 1300, \omega_1 = \{\langle Q_{1,1}, 0, 200\rangle, \langle Q_{1,2}, 200, 100\rangle, \langle Q_{1,3}, 300, 100\rangle, \langle Q_{1,4}, 400, 600\rangle, \\
&\qquad \langle Q_{1,2}, 1000, 100\rangle, \langle Q_{1,3}, 1100, 100\rangle, \langle Q_{1,4}, 1200, 100\rangle\}\rangle \\
\chi_2 &= \langle MTF_2 = 1300, \omega_2 = \{\langle Q_{2,1}, 0, 200\rangle, \langle Q_{2,4}, 200, 100\rangle, \langle Q_{2,3}, 300, 100\rangle, \langle Q_{2,2}, 400, 600\rangle, \\
&\qquad \langle Q_{2,4}, 1000, 100\rangle, \langle Q_{2,3}, 1100, 100\rangle, \langle Q_{2,2}, 1200, 100\rangle\}\rangle
\end{aligned}
$$

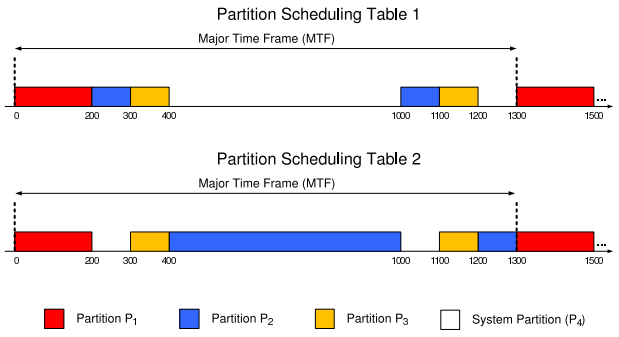

Fig. 8. Partition scheduling tables for the prototype implementation

We have the possibility to inject a faulty process on P_1, so that a deadline miss occurs even though both PSTs comply with P_1's timing requirements (cf. (23) and (25) for schedule χ_1).

$$\begin{aligned}
\sum_{\{\,\omega_{i,j}\in\omega_i \mid P^{\omega}_{i,j}=P_m \wedge O_{i,j}\in[k\,\eta_m;(k+1)\eta_m[\,\}} c_{i,j} &\geq d_m \qquad i=1, P_m=Q_{1,1}, k=0\\
\sum_{\{\,\omega_{1,j}\in\omega_1 \mid P^{\omega}_{1,j}=Q_{1,1} \wedge O_{1,j}\in[0;1300[\,\}} c_{1,j} &\geq 200\\
\sum_{\{\,\langle Q_{1,1},0,200\rangle\,\}} c_{1,j} &\geq 200\\
200 &\geq 200 \qquad \square
\end{aligned} \tag{25}$$

When the faulty process is active, its deadline violation is detected and reported every time (except the first) that P_1 is scheduled and dispatched to execute.

Successive requests to change schedule are correctly handled at the end of the current MTF and do not introduce deadline violations other than the one injected in a process in P_1. This is caused, not by the schedule switch mechanism itself, but by ensuring that the different PSTs comply with the temporal requirements of the partitions therein contained. This is in consonance with the overall tone of the ARINC 653 specification, which emphasizes that in many cases the system can only *support* certain properties, and cannot *guarantee* them without proper and careful integration and configuration [2].

To allow for proof of concept visualization and interaction, the prototype includes VITRAL, a text-mode windows manager for RTEMS [7], whose graphical aspect can be seen in Fig. 9. There is one window for each partition, where its output can be seen, and also two more windows which allow observation of the behaviour of AIR components. VITRAL also supports keyboard interaction, which is used, for demonstration purposes, to allow switching to a given partition scheduling table at the end of the present major time frame and activating the faulty process on P_1.

7 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 [31]. Previous academic research on TSP solutions [19] and works on scheduling analysis for TSP systems [4,18,12] do not include or foresee mechanisms for timing parameters adaptation.

Mode-based scheduling principles are also employed to communication protocols. In the Time-Triggered Protocol (TTP) [17], the controller state includes an operational mode, repeated at every mode cycle, which controls the sequence, attributes and schedule for nodes to send messages. If a node intends to change mode, it signals the remaining nodes through a frame's control field.

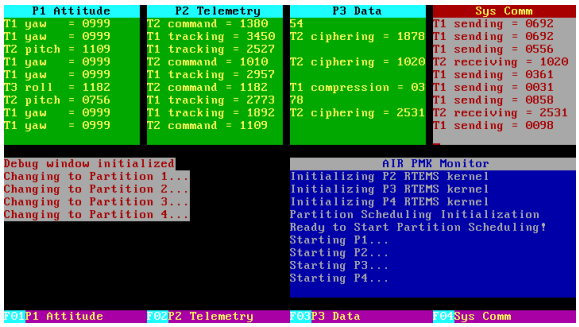

Fig. 9. Prototype implementation demonstration, featuring the VITRAL text-mode windows manager for RTEMS

The overall concept of a timing watchdog to detect timing failures in the context of IMA-based systems is mentioned in [5,6]. In order to process deadline violation monitoring, the ARINC 653 specification defines deadline miss as a process level error, but makes no considerations on how or when the error should be detected [2]. In AIR, on the other hand, we propose an efficient implementation of such a mechanism. XtratuM, in its documentation, does not mention any provision of any similar deadline supervision [19]. VxWorks 653 is said to fully implement the ARINC 653 APEX specification, but it is not clear if deadline violation monitoring is addressed [31].

Temporal analysis in TSP systems such as IMA/ARINC 653 as been addressed in some instances in the literature, albeit not to the full extent needed to aid design, integration and deployment of TSP systems in space. In [4] the response time analysis leads to the proposal of abandoning two-level scheduling in favour of a single-level priority preemptive scheduling, and [14] also makes the case for abandoning cyclic partition scheduling, but in favour of reservation-based scheduling.

A theorem for partition schedulability is presented in [18], assuming that each partition is assigned a single continuous execution time window within each iteration of its cycle; this is much of a simplification of the scheduling mechanisms for TSP systems. This fact is also pointed out in [20], which addresses the task and partition scheduling problems with assumptions that differ from those possible when using the IMA and ARINC 653 specifications as a basis. For instance, the authors analyze the schedulability of processes (within a partition) by Earliest Deadline First policies, whereas ARINC 653 mandates a preemptive priority-based algorithm [2].

Finally, [12] models ARINC 653 with two-level scheduling and apply timing analysis techniques to generate partition schedules. This analysis relies on a model with some limiting (and, in some cases, unjustified) assumptions; for instance, the authors ignore aperiodic processes on the grounds that they are scheduled as background workload.

8 Conclusion and Future Work

The strict requirements of modern aerospace systems has brought us to integrating several onboard functions (avionics, payload), traditionally separated in dedicated resources, in the same computing platform. Robust temporal and spatial partitioning (TSP) is introduced to address dependability challenges resulting from this integration. TSP involves onboard applications being separated in logical containers (partitions), implying fault containment. Partitions do not interfere with one another regarding real-time and addressing space separation requirements.

In this chapter we presented the design of the TSP-based AIR architecture, which is compliant with the ARINC 653 specification. Then, we formally modeled AIR, with emphasis on the temporal properties and requirements. The innovative features introduced in the AIR architecture to enforce its dependability with respect to timeliness guarantees (mode-based schedules and process deadline violation monitoring) are then detailed regarding their definition, implementation and implications on the definition of an extended system model. Finally, we presented a prototype implementation, demonstrating the AIR architecture with the newly introduced timeliness adaptation and control features.

Mode-based schedules and process deadline violation monitoring do not, actively and/or by themselves, improve the timeliness of an AIR system. What they do is provide valuable means for system developers, integrators, maintainers and mission controllers to have a much greater control on whether and how this timeliness is achieved. Process deadline violation monitoring can give an almost immediate insight on possible underdimensioning of the execution time given to one or more partitions, which, coupled with mode-based schedules and a system integrated and configured to cope with this kind of event, can allow the problem to be solved in execution time.

As future work, the system model resulting from this chapter, composed of equations:

- (1), (3) and (16)–(23) (partitions, and partition mode-based scheduling);
- (10)–(15) (processes), and;
- (24) (process deadline violations);

will be consolidated and much extended, namely so as to include: *(i)* necessary conditions for process scheduling and deadline fulfilment; *(ii)* spatial separation characteristics, addressing space protection attributes, and fault detection requirements; *(iii)* the implications of unforeseen events on the time model (aperiodic/sporadic processes, event overload, etc.), and; *(iv)* parallelism between partition time windows on a multicore platform [8]. Formal definition of the characteristics and requirements of an ARINC 653-based system, such as those built on the AIR architecture, is of paramount importance for future space missions, since it opens room for system verification and development of timing analysis tools to aid system integration. This also implies deeper studies on schedulability analysis for TSP systems.

Acknowledgments

The authors would like to thank Tobias Schoofs, Sérgio Santos, Cássia Tatibana, Edgar Pascoal, José Neves (GMV Portugal) and James Windsor (ESA–ESTEC) for the joint efforts in the scope of the AIR activities, and Manuel Coutinho, for the extensive work on the VITRAL window manager for RTEMS and on earlier AIR prototyping.

References

1. AEEC: Design guidance for Integrated Modular Avionics. ARINC Report 651-1 (Nov 1997)
2. AEEC: Avionics application software standard interface, part 1 - required services. ARINC Specification 653P1-2 (Mar 2006)
3. AEEC: Avionics application software standard interface, part 2 - extended services. ARINC Specification 653P2-1 (Dec 2008)
4. Audsley, N., Wellings, A.: Analysing APEX applications. In: Proc. 17th IEEE Real-Time Systems Symp. pp. 39–44. Washington, DC, USA (Dec 1996)
5. Bate, I., Burns, A.: A dependable distributed architecture for a safety critical hard real-time system. In: IEE Half-Day Colloquium on Hardware Systems for Dependable Applications (Digest No: 1997/335). pp. 1/1–1/6 (1997)
6. Conmy, P., McDermid, J.: High level failure analysis for Integrated Modular Avionics. In: Proc. 6th Australian Workshop on Safety critical systems and software. vol. 3, pp. 13–21. Australian Computer Society, Inc., Brisbane, Australia (2001)
7. Coutinho, M., Almeida, C., Rufino, J.: VITRAL - a text mode window manager for real-time embedded kernels. In: Proc. 11th IEEE Int. Conf. on Emerging Technologies and Factory Automation. Prague, Czech Republic (Sep 2006)
8. Craveiro, J.: Integration of generic operating systems in partitioned architectures. MSc thesis, Faculty of Sciences, University of Lisbon (Jul 2009)
9. Craveiro, J., Rufino, J., Almeida, C., Covelo, R., Venda, P.: Embedded Linux in a partitioned architecture for aerospace applications. In: Proc. 7th ACS/IEEE Int. Conf. on Computer Systems and Applications. pp. 132–138. Rabat, Morocco (May 2009)
10. Davis, R., Burns, A.: A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. Tech. Rep. YCS-2009-443, University of York, Department of Computer Science (2009)
11. Diniz, N., Rufino, J.: ARINC 653 in space. In: Proc. DASIA 2005 "DAta Systems In Aerospace" Conf. Edinburgh, Scotland (Jun 2005)
12. Easwaran, A., Lee, I., Sokolsky, O., Vestal, S.: A compositional scheduling framework for digital avionics systems. In: Proc. 15th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications. Beijing, China (Aug 2009)
13. Fortescue, P.W., Stark, J.P.W., Swinerd, G. (eds.): Spacecraft Systems Engineering, 3rd edition. Wiley (2003)
14. Grigg, A., Audsley, N.: Towards a scheduling and timing analysis solution for integrated modular avionic systems. Microprocessors and Microsystems Journal 22(8), 423–431 (1999)
15. IEEE: 1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application: Program Interface (API) [C Language]. IEEE, New York, USA (1996)

16. Kinnan, L.: Application migration from Linux prototype to deployable IMA platform using ARINC 653 and Open GL. In: Proc. 26th IEEE/AIAA Digital Avionics Systems Conference. pp. 6.C.2–1–6.C.2–5. Dallas, TX, USA (Oct 2007)
17. Kopetz, H., Grünsteidl, G.: TTP — a time-triggered protocol for fault-tolerant real-time systems. In: Proc. 23rd Int. Symp. on Fault-Tolerant Computing (1993)
18. Lee, Y., Kim, D., Younis, M., Zhou, J.: Partition scheduling in APEX runtime environment for embedded avionics software. In: Proc. 5th Int. Conf. on Real-Time Computing Systems and Applications. pp. 103–109. Hiroshima, Japan (1998)
19. Masmano, M., Ripoll, I., Crespo, A.: XtratuM Hypervisor for LEON2: design and implementation overview. Tech. rep., I. U. de Automática e Informática Industrial, Universidad Politécnica de Valencia (Jan 2009)
20. Mok, A.K., Feng, A.X.: Real-time virtual resource: A timely abstraction for embedded systems. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT '02. Lecture Notes In Computer Science, vol. 2491, pp. 182–196. Springer-Verlag, London, UK (2002)
21. OAR — On-Line Applications Research Corporation: RTEMS C Users Guide, 4.8 edn. (Feb 2008)
22. Rufino, J., Craveiro, J., Schoofs, T., Tatibana, C., Windsor, J.: AIR Technology: a step towards ARINC 653 in space. In: Proc. DASIA 2009 "DAta Systems In Aerospace" Conf. Istanbul, Turkey (May 2009)
23. Rufino, J., Filipe, S., Coutinho, M., Santos, S., Windsor, J.: ARINC 653 interface in RTEMS. In: Proc. DASIA 2007 "DAta Systems In Aerospace" Conf. Naples, Italy (Jun 2007)
24. Rushby, J.: Partitioning in avionics architectures: Requirements, mechanisms and assurance. NASA Contractor Report CR-1999-209347, SRI International, California, USA (Jun 1999)
25. Sánchez-Puebla, M.A., Carretero, J.: A new approach for distributed computing in avionics systems. In: Proc. 1st Int. Symp. on Information and Communication Technologies. pp. 579–584. Trinity College Dublin, Dublin, Ireland (2003)
26. Santos, S., Rufino, J., Schoofs, T., Tatibana, C., Windsor, J.: A portable ARINC 653 standard interface. In: Proc. IEEE/AIAA 27th Digital Avionics Systems Conf. St. Paul, MN, USA (Oct 2008)
27. Seyer, R., Siemers, C., Falsett, R., Ecker, K., Richter, H.: Robust partitioning for reliable real-time systems. In: Proc. 18th Int. Parallel and Distributed Processing Symp. pp. 117–122 (Apr 2004)
28. SPARC International, Inc., Menlo Park, CA, USA: The SPARC Architecture Manual, Version 8 (1992)
29. Terraillon, J.L., Hjortnaes, K.: Technical note on on-board software. European Space Technology Harmonisation, Technical Dossier on Mapping, TOSE-2-DOS-1, ESA (Feb 2003)
30. Watkins, C., Walter, R.: Transitioning from federated avionics architectures to Integrated Modular Avionics. In: Proc. 26th IEEE/AIAA Digital Avionics Systems Conf. Dallas, TX, USA (Oct 2007)
31. Wind River: Wind River VxWorks 653 Platform, `http://www.windriver.com/products/platforms/safety_critical_arinc_653/`, retrieved on Jun 17, 2010
32. Windsor, J., Hjortnaes, K.: Time and space partitioning in spacecraft avionics. In: Proc. 3rd IEEE Int. Conf. on Space Mission Challenges for Information Technology. pp. 13–20. Pasadena, CA, USA (Jul 2009)

Appendix: Notation

Symbol	Description
	Convention used
$\mathbb{N}$	Set of natural numbers ($0 \notin \mathbb{N}$)
$n(S)$	(Where S is a set) Equivalent to $\#S$
$a \bmod b$	Modulo operation (remainder of the division of a by b)
	Partitions
P	Set of partitions in the system
$n(P)$	Number of partitions in the system ($n(P) \equiv \#P$)
P_m	Partition m
$M_m(t)$	Operating mode of partition P_m at instant t (normal, idle, coldStart, or warmStart)
	Partition scheduling
χ	Set of partition schedules available in the system
$n(\chi)$	Number of partition schedules available ($n(\chi) \equiv \#\chi$)
χ_i	Partition schedule i
MTF_i	Major time frame of schedule χ_i
Q_i	Set of partition time requirements for χ_i
$P^{\chi}_{i,m}$	Each partition with at least one time window in χ_i
$\eta_{i,m}$	Activation cycle of partition $P^{\chi}_{i,m}$ under χ_i
$d_{i,m}$	Duration of partition $P^{\chi}_{i,m}$ under χ_i
ω_i	Set of time windows in schedule χ_i
$n(\omega_i)$	Number of time windows in schedule χ_i ($n(\omega_i) \equiv \#\omega_i$)
$\omega_{i,j}$	Time window j in schedule χ_i
$P^{\omega}_{i,j}$	Partition active during time window $\omega_{i,j}$
$O_{i,j}$	Offset of time window $\omega_{i,j}$, relative to the beginning of MTF_i
$c_{i,j}$	Duration of time window $\omega_{i,j}$
	Tasks/processes
τ_m	Taskset of partition P_m
$n(\tau_m)$	Number of tasks (processes) in partition P_m ($n(\tau_m) \equiv \#\tau_m$)
$\tau_{m,q}$	Task q of partition P_m
$T_{m,q}$	Period of task $\tau_{m,q}$
$D_{m,q}$	Relative deadline of task $\tau_{m,q}$
$p_{m,q}$	(Base) priority of task $\tau_{m,q}$
$C_{m,q}$	Worst case execution time (WCET) of task $\tau_{m,q}$
$S_{m,q}(t)$	Status of task $\tau_{m,q}$ at instant t
$D'_{m,q}(t)$	(Absolute) deadline time of task $\tau_{m,q}$ at instant t
$p'_{m,q}(t)$	(Current) priority of task $\tau_{m,q}$ at instant t
$St_{m,q}(t)$	State of task $\tau_{m,q}$ at instant t (dormant, ready, running, or waiting)
$Ready_m(t)$	Set of schedulable tasks (ready or running) in partition P_m at instant t
$heir_m(t)$	Heir task in partition P_m at instant t
$V(t)$	Set of tasks which, at instant t, have violated a deadline

Note: This notation, as described in this table, applies to the system model where multiple partition schedules are supported.