Flexible Operating System Integration in Partitioned Aerospace Systems

João Craveiro[1], José Rufino[1], Tobias Schoofs[2], and James Windsor[3]

[1] ⋆ LaSIGE, University of Lisbon, Portugal.
[2] Skysoft Portugal S.A., Lisbon, Portugal.
[3] ESA/ESTEC, Noordwijk, The Netherlands.

Abstract. The ARINC 653-based AIR (ARINC 653 in Space Real-Time Operating System) architecture, developed as a response to the interest of the aerospace industry in adopting the concepts of Integrated Modular Avionics (IMA), proposes a partitioned environment, observing strict temporal and spatial segregation, in which partitions are able to use different (real-time) operating systems and host applications of different criticality levels. This paper centers on recent enhancements to the AIR architecture, like the AIR POS Adaptation Layer (PAL), which aim at optimizing the development and integration processes with the flexible support to new partition operating systems (POS) in mind. We also discuss the current efforts, which already benefit from the properties of the AIR Technology, to integrate Linux as a POS, exploiting the concepts of the paravirtualization interface currently provided by the Linux kernel.

Resumo. A arquitectura AIR (ARINC 653 in Space Real-Time Operating System), baseada na especificação ARINC 653 e desenvolvida como resposta ao interesse da indústria aeroespacial em adoptar os conceitos de Integrated Modular Avionics (IMA), propõe um ambiente compartimentado e com observância estrita de segregação temporal e espacial, onde as partições podem utilizar diferentes sistemas operativos (de tempo-real) e conter aplicações com diferentes níveis de criticalidade. Este artigo centra-se em melhoramentos recentes à arquitectura AIR, como o componente AIR POS Adaptation Layer (PAL), que visam optimizar os processos de desenvolvimento e integração, com a flexibilidade no suporte a novos sistemas operativos em mente. Discutimos também os esforços actuais, que já beneficiam das propriedades da tecnologia AIR, para integrar o Linux como sistema operativo de uma das partições, explorando os conceitos da interface de paravirtualização actualmente fornecida pelo núcleo Linux.

1 Introduction

Traditional federated architectures for avionics systems are based on the distribution of avionics functions along separate collections of dedicated hardware resources. The

⋆ Faculdade de Ciências da Universidade de Lisboa, Bloco C6, Piso III, Campo Grande, 1749–016 Lisboa, Portugal. This work was partially developed within the scope of the ESA (European Space Agency) Innovation Triangular Initiative program, through ESTEC Contract 21217/07/NL/CB — Project AIR-II (ARINC 653 in Space RTOS — Industrial Initiative), URL: http://air.di.fc.ul.pt). This work was partially supported by FCT through the Multiannual Funding and the CMU-Portugal Programs.

demands of modern systems — like reducing system size, weight, and power (SWaP) — require more efficient architectures [1].

As a challenge to federated avionics architectures towards this goal, the Integrated Modular Avionics (IMA) specification defines a partitioned environment, comprising processing, communications and input/output resources, to be shared among avionics functions of different criticalities [2]. Closely related to this architecture is the Avionics Application Software Standard Interface, defined in the ARINC 653 specification [3], and the concepts of temporal and spatial partitioning.

In ARINC 653, temporal partitioning consists of the time-sliced allocation of computing resources to hosted applications, achieved through a fixed, cyclic scheduling of partitions over a major time frame (MTF). This way, strong temporal segregation is achieved, in which activities inside each partition do not affect the timeliness of activities executing inside the remaining partitions in the system. Robust spatial partitioning concerns preventing applications from accessing memory zones outside those belonging to its partition.

Having similar requirements (safety, SWaP, etc.) as avionics platforms, space missions can benefit from adopting similar approaches, which has sparked the interest of the aerospace industry, and the European Space Agency (ESA) in particular, in the concepts of IMA, and time and space partitioning [4,5]. This has led to the development, within the scope of ESA-sponsored initiatives, of the AIR (ARINC 653 in Space RTOS) architecture. The AIR architecture provides time and space partitioning in conformity with the defined in the ARINC 653 specification [3]. AIR preserves the hardware and real-time operating system (RTOS) independence defined within the scope of ARINC 653, while foreseeing the use of different RTOS through the partitions [6,7,8].

However, porting general-purpose applications to one of the RTOSs one might be using can be a morose task, and certainly not an error-free one [9]. Furthermore, certain hardware interfaces may be necessary that are not supported by the given RTOS. This also applies to the aerospace applications that the AIR architecture targets; an example is a space probe for planetary observation, within which a hardware interface with a camera is needed, and whose pictures need to go through some post-processing by a widely available application that has not been ported to the RTOS. Thus we have preliminarily evaluated the integration of generic operating systems, like (embedded) Linux [10]. The execution of non-real-time Linux processes alongside real-time tasks has been studied and implemented before. Examples include RTLinux [11] and xLuna [12], where the non-real-time Linux processes are only scheduled and dispatched when there is no real-time task ready to execute. The approach on the AIR architecture differs in that the non-real-time Linux partition has a guaranteed execution time window in the cyclic, fixed scheduling of partitions.

In this paper, we describe the recent improvements and current efforts on the AIR architecture, towards the flexible integration of both real-time operating systems and generic non-real-time operating systems in partitions. This paper is structured as follows.

In Section 2, we describe the current state of the art concerning the AIR Technology, so as to provide a solid background. Then, in Section 3, we describe in detail the characteristics and advantages of a recently introduced component, the AIR Partition

OS Adaptation Layer (PAL). This component benefits architectural features and the engineering of AIR components (thus adding flexibility to the process of supporting new partition operating systems) and promotes separation of concerns (to optimize development processes at its various stages). In Section 4, we expose the current efforts on exploiting the flexible integration of partition operating systems, to add support for the Linux operating system. Finally, in Section 5, we draw concluding remarks, and lay the foundations for future developments.

2 AIR Technology overview

The AIR activities span over two projects. The first resulted in the development of a proof of concept and a demonstration of feasibility of use [6,7]. The second, AIR-II, which is still in progress, aims evolving towards an industrial product definition by improving and completing the key ideas identified [8].

2.1 System architecture

The fundamental idea in the definition of the AIR architecture is a simple solution for providing the ARINC 653 functionality missing in off-the-shelf (real-time) operating system kernels, encapsulating those functions in special-purpose additional components with well-defined interfaces, as illustrated in the diagram of Fig. 1. In essence, the AIR architecture preserves the hardware and operating system independence defined in the ARINC 653 specification [3]. Applications may use a strict ARINC 653 service interface or, in the case of system partitions, may bypass this standard interface and use partition operating system kernel specific functions, 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 due increased verification efforts.

The following fundamental components have been defined within the AIR system architecture. We will now explain them with as much detail as needed for the understanding of the issues at hand in this paper. More detailed descriptions can be found in previous publications [7,8].

2.2 AIR Partition Management Kernel (PMK)

The AIR Partition Management kernel (PMK), is a simple micro-kernel that efficiently handles partition scheduling and dispatching, thus playing a lead role in securing robust temporal segregation. A two-level hierarchical scheduling [13] scheme is used: partitions are scheduled deterministically at PMK level by a fixed cyclic scheduler; scheduling of the application processes inside each partition is normally handled by the native Partition Operating System (POS) scheduler. RTOS kernels typically offer a preemptive, priority-based process scheduler. At the AIR PMK level, the Partition Scheduler checks at each system clock tick whether a partition preemption is to occur; if it is so, the AIR PMK Partition Dispatcher has to perform the partitions' context switch. The

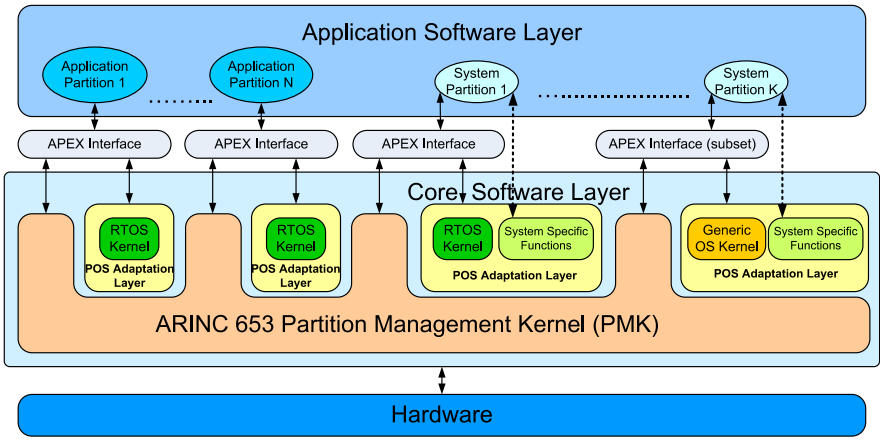


Fig. 1. AIR architecture overview

native POS process scheduler of the heir partition is then notified of the amount of clock ticks elapsed since it was last preempted, thus adjusting the heir partition system time to a common partition-wise time referential.

The design of AIR PMK also incorporates enhanced mechanisms to ensure temporal segregation, like mode-based schedules and process deadline violation monitoring [8].

2.3 Flexible Portable APEX Interface

The APEX Interface implements a set of services defined in the ARINC 653 specification. For generic operating systems (e.g. embedded Linux) only a subset of the APEX standard primitives is needed, primarily for management and monitoring purposes [10]. The APEX design and implementation of the APEX may benefit from the availability of functions related to the recently introduced AIR POS Adaptation Layer (PAL) [8], also detailed in Section 3.

2.4 AIR Health Monitoring (HM)

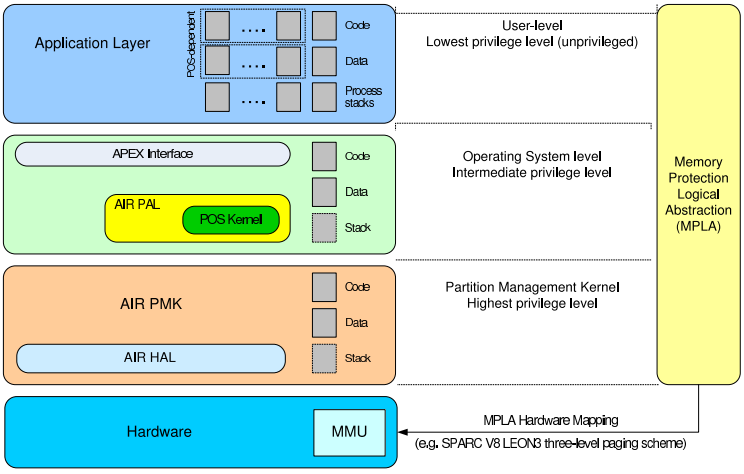
The AIR Health Monitor is responsible for handling hardware and software errors (like deadlines missed, memory protection violations, bounds violation or hardware failures). The aim is to isolate errors within its domain of occurrence: process level errors will cause an application error handler to be invoked, while partition level errors trigger a response action defined in a configuration table. Errors detected at system level may lead the entire system to be stopped or reinitialized [8].

2.5 AIR Space Partitioning and Operating System Integration

The robust partitioning approach defined in the AIR architecture implies the spatial separation of the different (real-time and non real-time) operating systems and its applications in integrity and criticality containers, defined by partitions. Partitions encapsulate

the addressing spaces of the contained POS and applications. No component integrated in a given partition can directly access the addressing space of other partitions, thus guaranteeing that partitions do not interfere with each other [7,8].

A highly modular design approach has been followed in the support of AIR spatial partitioning. Spatial partitioning requirements, specified in ARINC 653 configuration files with the assistance of development tools support, are described in run-time through a high-level processor independent abstraction layer [8]. A set of descriptors is provided per partition, primarily corresponding to the several levels of execution of an activity (e.g. application, POS kernel and AIR PMK) and to its different memory blocks (e.g. code, data and stack), as illustrated in the diagram of Fig. 2.



Memory Management Unit (MMU) Mapping Results				
Processing Platform	***MMU Address Translation Model***	***Primary MMU Descriptors per Partition***	***Number of Partitions***	***Partition Size***
IA-32	segmentation	1	variable	variable
	paging	1	1024	4 MiB
SPARC V8	paging	1	256	16 MiB

Fig. 2. AIR Spatial Partitioning and Operating System Integration

In the AIR architecture the definition of the high-level abstract spatial partitioning takes into account the semantics expected by user-level application programming. At each partition, the application environment inherits the execution model of the corresponding POS and/or its language run-time environment. This is true for system partitions and may be applied also to application partitions, using only the standard APEX interface.

The high-level abstract spatial partitioning description needs to be mapped in run-time to the specific processor memory protection mechanisms, possibly exploiting the

availability of high-level logical address translation schemes, as provided by memory management unit (MMU).

The mapping into MMU specific mechanisms depends on the resources available on each processing platform foreseen for AIR applications. The most versatile mapping assumes the use of a memory segmentation model, such as it exists in the Intel IA-32 architecture, where a one-to-one mapping between the high-level abstract spatial partitioning description and low-level memory management descriptors is possible.

A one-to-one mapping is not possible if a paging translation model is being used in the MMU since a memory descriptor is required by page frame. This also implies some restrictions with respect to the number and size of partitions, as illustrated by the data inscribed in Fig. 2.[1] An optimal design approach is assumed, where the action of changing the status of a partition (active/inactive) requires no more than the update of a single primary MMU descriptor per partition. The data inscribed in Fig. 2, for IA-32 and SPARC V8 RISC processing architectures, is in conformity with the requirements found in typical avionics and aerospace applications.

Mapping of high-level abstract partitioning also includes the management of privilege levels: only the AIR PMK is executed in privileged mode (cf. Fig. 2). The lack of multiple protection rings, such as it exists in the Intel IA-32 processor architecture, may be mitigated in the SPARC V8 architecture by granting access to a given level only during the execution of services belonging to that level (or lower ones). This may be achieved by activating the corresponding memory protection descriptors upon call of a service primitive, and deactivating them when service execution ends.

The provision of these mapping functions is under the scope of overall partition management as provided by the APEX layer and by some specific AIR PMK components. For example, the mapping into processor specific descriptors needs to be updated in run-time when a partition switch occurs. This has to be coordinated by AIR PMK specific components, in this case by the AIR PMK Partition Dispatcher.

3 The AIR POS Adaptation Layer (PAL)

The AIR architecture allows operating systems integration without any fundamental change to a given POS Kernel. In essence, only the OS initialization process and the system clock handler need to be adapted. A generic approach and a uniform methodology have been adopted for the integration of both real-time and generic operating systems [15]. The adopted solution wraps the POS and, if applicable, the system specific functions, through the use of the AIR POS Adaptation Layer (PAL), facilitating the integration at the low- and at high-level domains, i.e. with respect to the AIR PMK and APEX components.

The AIR PAL was added to the original AIR architecture [7] as a means to a truly POS-independent AIR PMK, but its benefits go beyond that. The improvements enabled by the AIR PAL go in three ways: **(i)** architectural properties; **(ii)** engineering of AIR components, and; **(iii)** leaner development processes, stemming from separation of

[1] It is worth mentioning that this paper follows a notation in conformity with the IEC 60027-2 standard in respect to the usage of prefixes for binary multiples [14].

concerns. We will now briefly elaborate on these three direction of benefits, which are more thoroughly described and illustrated in [15].

3.1 Architectural properties

By wrapping each partition's operating system kernel inside an adaptation layer (the AIR PAL), the AIR PMK can act upon the POS in a way that is agnostic of the latter, when necessary. Upon the need to add support to a different POS, the AIR PMK remains unaltered, with support being coded by developing an adequate PAL. This way, previous or ongoing verification, validation and/or certification efforts on the AIR PMK are not hindered. The AIR PAL also benefits the design of other AIR components, such as the Portable APEX and the AIR Health Monitoring (HM) [15].

3.2 Component engineering

Besides consolidating the properties of the AIR architecture and its components, the AIR PAL can also make up for some non-optimal or inappropriate behavior of the native POS implementation of some function. Also, by providing these surrogate functions — intended to be called in spite of the native ones — instead of creating patches to be applied to the POS's source code, we extend the lifetime of the support to a given POS through more subsequent versions of the latter. The reason for this is that the patches' mapping onto their target relies on source code file names and line numbers, whereas the AIR PAL relies on function prototypes and behaviors.

The improvements on architectural properties and component engineering are closely related, and constitute the basis of a flexible integration of partition operating systems.

3.3 Separation of concerns

Another reason against patching the POS so as to obtain the integration and intended behavior for running on the AIR architecture is that it would break the desired separation of concerns, thus undermining an otherwise streamlined development process. Application developers should not be concerned with how the underlying POS is adapted to the AIR architecture, and neither should support for new POSs divert the AIR PMK maintainers' focus from what should be their main concerns — the robust temporal and spatial partitioning properties of AIR.

By consolidating the separation of concerns in the AIR architecture, the development workflow can rely much more on reusable components. This leads to leaner software [10] development processes, with less overhead spent on interactions between different stakeholders (partition application developers, system integrators, etc.).

4 Integration of generic non-real-time operating systems

In [10], we presented the problem of integrating generic operating systems onto the AIR architecture, thus tackling the issue of functionally porting general-purpose applications to an environment provided by real-time operating systems [9].

4.1 Usefulness of Linux-based partitions

One solution for avoiding the effort of porting general-purpose applications to real-time partitions is to have them running on their native operating system.

We have looked into Linux as a candidate for a generic non-real-time partition OS in AIR and shown the development and evaluation of a fully functional operating system, based on the Linux kernel. The integration of Linux makes available to AIR applications a wide range of utilities, tools, language interpreters (Python, Perl, Ruby, tcl, etc.), and device drivers. This specific facilities no longer need to be ported to the RTOS to construct AIR applications. Should it be a requirement, the access to those tools can be supported by AIR inter-partition communication facilities [10].

4.2 Embedded Linux

Given the coexistence of the multiple POSs in the system, which in the absence of persistent storage (e.g. hard disk drive) will be resident in memory during the entire platform execution time, it is particularly important to keep the POSs to a minimum size. Thus, it makes perfect sense using techniques and methodologies aimed at systems with scarce resources — embedded systems.

The embedded variant of Linux which was described and evaluated in [10] was developed around three pillars of optimization: kernel configuration, system library, and utilities/tools. Regarding kernel configuration, size optimizations consisted of selecting only a relevant set of features, and providing those as built-in in the kernel, rather than as loadable modules. The system library used in most typical Linux distributions, the GNU C library (glibc) was replaced by uClibc, more appropriate for systems with scarce resources; uClibc developers accomplished this by reimplementing functionalities with size optimizations in mind, and by modularizing some of them, (allowing the configuration of the uClibc library and its adaptation to the requirements of the target system). Finally, common utilities and tools, usually provided as standalone executables, are provided through a utility called BusyBox, which also provides optimized implementations and allows both selection and fine-tuning of the utilities to include in one single executable file.

To aid building the cross compilation toolchain and the final target system image, we used Buildroot, which also allows (through its simple configuration tool) including extra functionalities as standalone executables, such as a different system shell, or support for interpreted/scripting languages. Figure 3 compares the obtained embedded variant of Linux with a typical desktop distribution.

Sizes illustrated for the system library and utilities/tools account for identical sets of features on both sides. Regarding the Linux kernel, a typical Linux distribution ships a set of loadable kernel modules that can amount to 50 MiB, which were not accounted for in the chart to allow for a fairer comparison.

Since the AIR architecture, when running on a SPARC architecture, will allow for 16 MiB per partition, the obtained size of about 1.5 MiB is very promising, and is also closer to the typical size of space applications.

Kernel		System library		System tools		Total	
Generic	*Embedded*	*glibc*	*uClibc*	*Generic*	*BusyBox*	*Generic*	*Embedded*
2150 KiB	830 KiB	2474 KiB	368 KiB	1932 KiB	363 KiB	6556 KiB[a]	1561 KiB

[a] Plus a set of modules amounting to 50 MiB

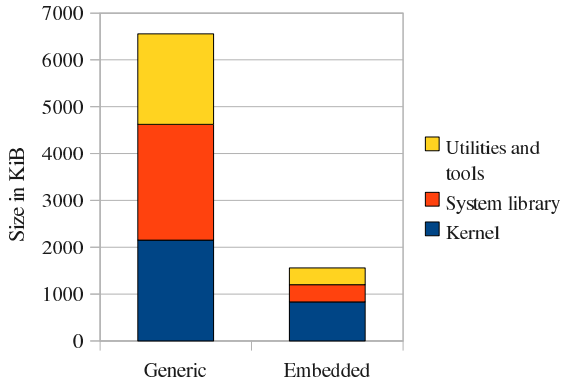


Fig. 3. Overall size comparison between an Embedded Linux and a typical Linux distribution

4.3 Integration issues

Issues being currently researched regarding the integration of Linux as partition OS focus on guaranteeing that it does not contaminate the robust temporal and spatial partitioning of the AIR architecture. Temporal partitioning is ensured, as standard, by the cyclic fixed scheduling of partitions, provided that the Linux partition can not disable or divert interrupts at the hardware (processor) level. We will want the Linux kernel to be notified of clock ticks, like other partition operating systems, only when its partition is active. Thus, interrupts will be totally controlled and handled by the AIR PMK, bypassing the Linux interrupt infrastructure [16].

To guarantee this, and since most processor architectures are not fully virtualizable (i.e., not all sensitive instructions are also privileged instructions), we can not merely run Linux in an unprivileged mode (usermode) and rely on having sensitive instructions generate a trap [17,18]. A good candidate to solve this issue is the employment of paravirtualization [19].

4.4 Paravirtualization in the Linux kernel

The paravirt-ops paravirtualization interface, which enables multiple hypervisors to hook directly into the Linux kernel, has been merged into the main Linux kernel starting with version 2.6.21, along with the support for VMWare's Virtual Machine Interface (VMI). VMI is the open specification of an interface for the paravirtualized guest OS kernel to communicate with the hypervisor [20], which takes advantage of hooks onto the paravirt-ops interface. Many popular GNU/Linux distributions shipping with Linux 2.6.21 have the paravirt-ops and VMI configuration options enabled; this means that the

same kernel will run both on native hardware and on top of a VMI-enabled hypervisor without requiring recompilation (with negligible performance overhead when running on native hardware [21]).

Figure 4 illustrates the process in which a VMI-enabled Linux kernel is booted, and either runs natively or on top of a hypervisor. Early during the boot process, the VMI initialization code probes for a ROM module through which the hypervisor's VMI layer is to be published to the paravirtualized operating system. If such a module is found, the VMI initialization code dynamically patches the kernel, so as to inject the necessary calls to the hypervisor's VMI layer; if not, the kernel continues to run as normal, natively on top of the hardware [20].

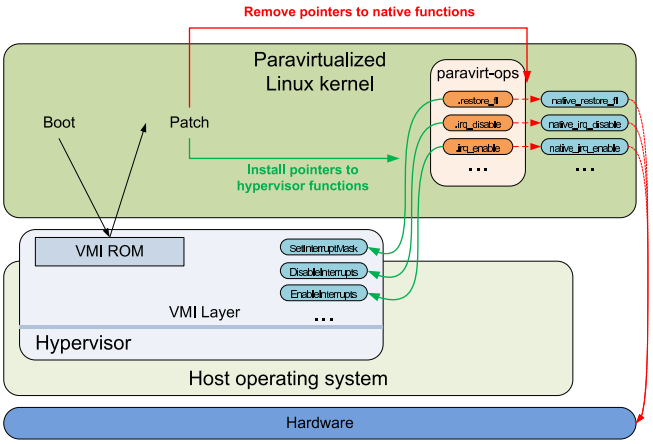


Fig. 4. Boot process of a paravirtualized Linux kernel on top of a VMI-compliant hypervisor

4.5 AIR Linux partition: AIR PAL design and integration

When transposing this to the reality of the AIR architecture, the AIR PAL will provide the relevant functions of the VMI layer to the partition operating system, interacting with AIR PMK when required. Examples of the VMI functions to be provided by the AIR PAL include virtualization of: **(i)** interrupt management; **(ii)** input/output (I/O) calls; **(iii)** memory and I/O space protection mechanisms; **(iv)** privilege level management. This integration is illustrated in Fig. 5.

4.6 AIR application platforms

The space applications to which the AIR technology is to be applied typically employ SPARC-V8 RISC processors, like LEON 2 and LEON 3, so the concepts of paravirt-ops and VMI, which are Intel IA-32 and Intel 64-centric by design, have to be transposed to the reality of this architecture. The current state of the art is nevertheless interesting for

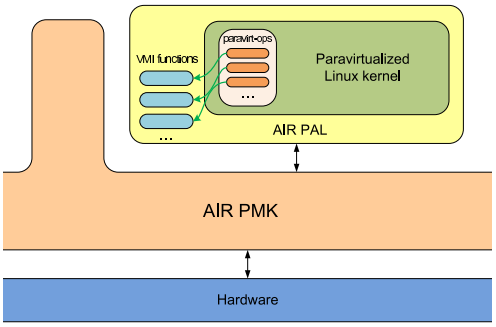


Fig. 5. Concepts of paravirtualization in the AIR architecture

proof of concept prototyping purposes, and to apply to ground-segment applications, where the Intel architectures are present. As of Linux kernel 2.6.30, paravirt-ops and VMI support is implemented for both Intel IA-32 and Intel 64 architectures.

5 Conclusions and future work

In this paper, we have focused on recent and current developments performed on the ARINC 653-based AIR Technology, in order to make the process of adding support to new operating systems, including generic non-real-time operating systems, more flexible. Having described the essentials of the AIR architectural components, we further detailed the latest space segregation results and the recently introduced AIR Partition OS Adaptation Layer (PAL), crucial for the provision of a homogeneous and flexible operating system integration process, which brings stronger architectural properties, benefits the engineering of the nuclear components of the AIR architecture, and improves the development process in its various stages, by promoting separation of concerns.

We also describe the current efforts of integrating Linux as a partition operating systems, focusing on the concepts associated with the paravirtualization interface currently provided by the Linux kernel (paravirt-ops).

There are still challenges open to future developments, both at architectural level and in the provisioning of adequate tools to build ARINC 653-based systems and applications. At the architectural level, future work includes consolidating the application of the concepts of paravirt-ops to the integration of Linux. A first approach will be based on the Intel IA-32 architecture, for which the paravirt-ops interface is already implemented; subsequently, the idea should be ported to the SPARC architecture, namely the LEON processors, employed in space missions. Work concerning the provisioning of adequate tools will include tools for developers and integrators, combining the analysis of the mutual impact between partition- and process-level scheduling with the automated generation of partition scheduling tables.

References

1. Watkins, C., Walter, R.: Transitioning from federated avionics architectures to Integrated Modular Avionics. In: Digital Avionics Systems Conference, 2007. DASC '07. IEEE/AIAA 26th. (October 2007) 2.A.1–1–2.A.1–10
2. Airlines Electronic Engineering Committee (AEEC): Design Guidance for Integrated Modular Avionics, ARINC Specification 651 (1991)
3. Airlines Electronic Engineering Committee (AEEC): Avionics Application Software Standard Interface, ARINC Specification 653-2 Part 1 (Required Services) (March 2006)
4. Terraillon, J.L., Hjortnaes, K.: Technical note on on-board software. European Space Technology Harmonisation, Technical Dossier on Mapping, TOSE-2-DOS-1, ESA (February 2003)
5. TSP Working Group: Time and space partitioning for space application (presentation). In: ESA Workshop on Avionics Data, Control and Software Systems (ADCSS). (October 2008)
6. Diniz, N., Rufino, J.: ARINC 653 in space. In: Proceedings of the DASIA 2005 "DAta Systems In Aerospace" Conference, EUROSPACE (June 2005)
7. Rufino, J., Filipe, S., Coutinho, M., Santos, S., Windsor, J.: ARINC 653 interface in RTEMS. In: Proceedings of Data Systems in Aerospace (DASIA'07). (June 2007)
8. Rufino, J., Craveiro, J., Schoofs, T., Tatibana, C., Windsor, J.: AIR Technology: a step towards ARINC 653 in space. In: Proceedings of the DASIA 2009 "DAta Systems In Aerospace" Conference, EUROSPACE (May 2009)
9. Kinnan, L., Wlad, J., Rogers, P.: Porting applications to an ARINC 653 compliant IMA platform using VxWorks as an example. In: Digital Avionics Systems Conference, 2004. DASC 04. The 23rd. Volume 2. (October 2004) 10.B.1–10.1–8 Vol.2
10. Craveiro, J., Rufino, J., Almeida, C., Covelo, R., Venda, P.: Embedded Linux in a partitioned architecture for aerospace applications. In: Proceedings of the 7th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'09). (May 2009) 132–138
11. Yodaiken, V., Barabanov, M.: A real-time Linux. In: Proc. Linux Applications Development and Deployment Conference (USELINUX). (January 1997)
12. Braga, P., Henriques, L., Carvalho, B., Chevalley, P., Zulianello, M.: xLuna - demonstrator on ESA Mars Rover. In: Proc. of DASIA 2008 — DAta Systems In Aerospace. (May 2008)
13. Audsley, N., Wellings, A.: Analysing APEX applications. In: 17th IEEE Real-Time Systems Symposium. (Dec 1996) 39–44
14. International Electrotechnical Commission (IEC): IEC 60027-2: Letter symbols to be used in electrical technology – Part 2: telecommunications and electronics (August 2005)
15. Craveiro, J., Rufino, J., Schoofs, T., Windsor, J.: Robustness, flexibility and separation of concerns in ARINC 653-based aerospace systems. AIR-II Technical Report RT-09-02 (2009) Submission to IEEE Embedded Systems Letters (ESL).
16. Bovet, D., Cesati, M.: Understanding the Linux Kernel. 3rd edn. O'Reilly Media, Inc. (August 2008)
17. Popek, G.J., Goldberg, R.P.: Formal requirements for virtualizable third generation architectures. Commun. ACM **17**(7) (1974) 412–421
18. Smith, J.E., Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann Publishers (2005)
19. Whitaker, A., Shaw, M., Gribble, S.D.: Denali: Lightweight virtual machines for distributed and networked applications. In: Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation. (June 2002) 195–209
20. Amsden, Z., Arai, D., Hecht, D., Holler, A., Subrahmanyam, P.: VMI: An interface for paravirtualization. In: Proceedings of the Linux Symposium. (July 2006) 363–378
21. VMware, Inc.: Native performance: paravirt vs non-paravirt kernel (May 2009) http://www.vmware.com/interfaces/paravirtualization/performance.html.