Embedded Linux in a Partitioned Architecture for Aerospace Applications

João Craveiro, José Rufino
LaSIGE–FCUL
Lisboa, Portugal
jcraveiro@lasige.di.fc.ul.pt, ruf@di.fc.ul.pt

Carlos Almeida, Rui Covelo, Pedro Venda
Instituto Superior Técnico
Lisboa, Portugal
cra@comp.ist.utl.pt, ruicovelo@gmail.com, pjvenda@pjvenda.org

***Abstract*—The ARINC 653 specification, defined for aeronautical applications, has the goal of providing a standard interface between a given real-time operating system (RTOS) and the corresponding applications. It also provides robust partitioning, with the final goal of guaranteeing safety and timeliness in mission-critical systems. The interest in ARINC 653 has extended to the aerospace industry, which resulted in the definition of an architecture, compliant with the specification, allowing for operating system heterogeneity. In this paper, we introduce the problem of integrating generic operating systems onto this architecture, and explore the case of GNU/Linux. Adding GNU/Linux allows running existing applications or interpreted scripts without needing to port the application or interpreter to an RTOS. In embedded systems, we have to cope with scarce resources and diverse existent hardware, and a balance between both issues must be reached. For such, we show the genesis of such a solution.**

***Index Terms*—Aerospace industry, computer applications, operating system kernels, operating systems, processor scheduling, real time systems.**

I. Introduction

The ARINC 653 specification [1], defined for aeronautical applications, has the goal of providing a standard interface between a given real-time operating system (RTOS) and the corresponding applications — the APEX (Application Executive) interface. It also presents a concept of temporal and spatial segregation (which consists in the confining of each application — partition, in ARINC 653 terminology — to its memory space and to its temporal window of possession of computing resources). The overall goal is to guarantee safety and timeliness in mission-critical systems. The interest in the concepts of the ARINC 653 specification has extended to the aerospace industry. The AIR (ARINC 653 Interface in RTOS) project — under an industry consortium initiative sponsored by the European Space Agency (ESA) — resulted in the design and definition of a system architecture compliant with ARINC 653 and independent of each specific operating system (OS). Currently, the AIR-II (ARINC 653 in Space RTOS — Industrial Initiative) project is in place, with the goal of consolidating the AIR technology, and evolving towards the definition of an industrial product for aerospace applications. The independence of ARINC 653 towards the operating system was naturally extended to the developed architecture. Thus, the heterogeneity between the RTOS kernels (RTEMS [2], eCos [3], VxWorks [4], etc.) in the various partitions was foreseen.

In this paper, we introduce the effort of consolidating and extending the architectural features to support this heterogeneity towards the operating system kernels to integrate in each partition. These should include RTOS kernels (free/open source, or commercial), and also general-purpose operating systems kernels, like GNU/Linux. The relevance of GNU/Linux specially concerns the availability of a wide set of interfaces together with a wide set of application software. Access of RTOS applications to these facilities can be achieved using AIR inter-partition communication.

The paper is organised as follows. Section II presents the architecture defined in the ARINC 653 specification. Section III exposes the characteristics and properties of the AIR architecture. Section IV introduces the purpose and problematic of integrating generic operating systems, like GNU/Linux, into this kind of architecture. It also includes a review of the Linux state of the art, with emphasis on the features to approach real-time behaviour. Section V describes the process of obtaining a GNU/Linux tailored for embedded and/or real-time systems, using a design-by-reuse approach. Section VI presents the results from the experiences reported in Section V, namely in terms of size and functionality comparison against a standard GNU/Linux distribution. Section VIII closes this paper with concluding remarks.

II. ARINC 653 concepts

The ARINC 653 specification is an important block from the *Integrated Modular Avionics (IMA)* definition [5], where the partitioning concept emerges for protection and functional separation between applications, usually for fault containment and ease of validation, verification, and certification [1], [6].

A. ARINC 653 System Architecture

The architecture of a standard ARINC 653 system is sketched in Figure 1. At the application software layer, each application is executed in a confined context, dubbed partition in ARINC 653 terminology [1]. The application software layer may include system partitions intended to manage interactions with specific hardware devices.

Application partitions consist in general of one or more processes and can only use the services provided by a logical

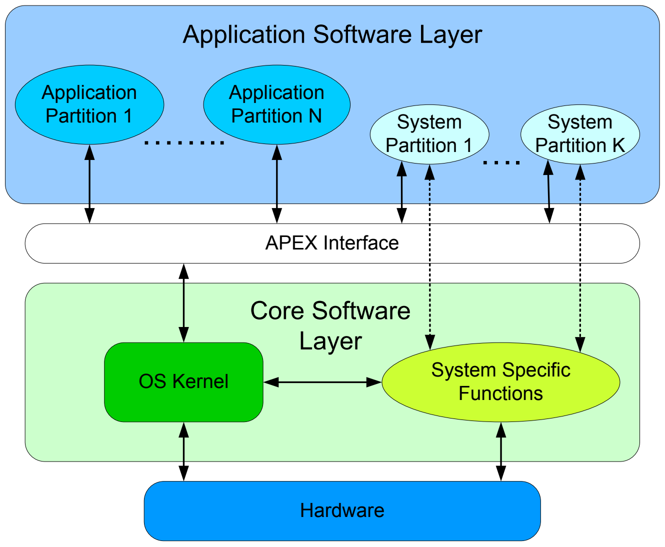

Fig. 1. Standard ARINC 653 architecture

application executive (APEX) interface, as defined in the ARINC 653 specification [1]. System partitions may use also specific functions provided by the core software layer (e.g. hardware interfacing and device drivers), being allowed to bypass the standard APEX interface.

The execution environment provided by the OS kernel module must furnish a relevant set of operating system services, such as process scheduling and management, time and clock management, and inter-process synchronisation and communication.

B. Spatial and Temporal Partitioning

Spatial partitioning ensures that it is not possible for an application to access the memory space (both code and data) of another application running on a different partition.

Temporal partitioning ensures that the activities in one partition do not affect the timing of the activities in any other partition. In ARINC 653, this is supported by a *fixed cycle based scheduling*, where a *major time frame* (MTF) of fixed duration is periodically repeated throughout runtime operation.

C. Health Monitoring

The Health Monitoring (HM) functions consist of a set of mechanisms to monitor system resources and application components. The HM helps to isolate faults and to prevent failures from propagating. Within the scope of the ARINC 653 standard specification the HM functions are defined for process, partition and system levels [1].

D. ARINC 653 Service Interface

The ARINC 653 service requests define the application executive APEX interface layer (Figure 1) provided to the application software developer and the facilities the core executive shall supply. A set of services is mandatory for strict compliance with the ARINC 653 standard [1], grouped into the following major categories: partition and process management, time management, intra and inter-partition communications, and health monitoring.

III. ARINC 653 Interface in Real-Time Operating Systems

The AIR innovation initiative represents a first but significant step toward the usage of off-the-shelf open-source RTOS kernels in the definition and design of ARINC 653 based systems.

This section describes the fundamental ideas on how a RTOS kernel can be integrated in an architecture able to offer the application interface and the functionality required by the ARINC 653 specification [1], [7].

A. AIR System Architecture

A simple solution for providing the ARINC 653 functionality missing in off-the-shelf RTOS kernels, such as the Real-Time Executive for Multiprocessor Systems (RTEMS) [2], implies to encapsulate those functions in components with a well-defined interface and add them to the bare operating system architecture.

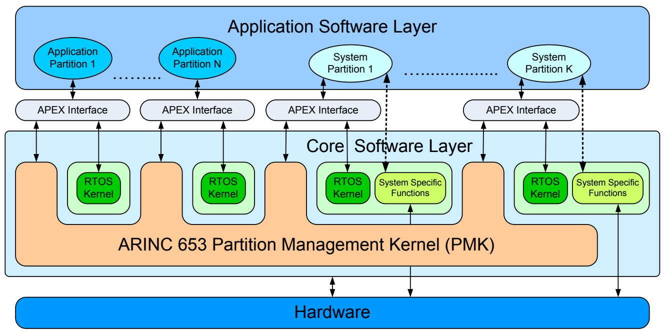

Fig. 2. Overview of the AIR system architecture

The design of the AIR architecture in essence preserves the hardware and RTOS independence defined within the scope of the ARINC 653 specification [1], [8], [7]. A specific module (cf. Figure 2) that needs to be added to the RTOS kernel (e.g. RTEMS) is the *AIR Partition Management Kernel (PMK)*, a simple microkernel that efficiently handles partition scheduling and dispatching, as well as inter-partition communication.

Another fundamental component concerns the *ARINC 653 application executive (APEX) interface*, defining for each partition in the system a set of services in strict conformity with the ARINC 653 standard. The APEX interface is designed as much as possible by mapping the ARINC 653 services into the native and/or POSIX primitives of the RTOS [1], [2], [3].

B. AIR Robust Partitioning and Composability

Robust partitioning comprises the protection of each partition's memory addressing space, to be provided by specific memory protection mechanisms usually implemented in a hardware memory management unit (MMU). It requires also a functional protection concerning the management of privilege levels and restrictions to the execution of privileged instructions. A basic set of such mechanisms do exist in the

Intel IA-32 architecture (widely used in everyday applications) and, to a given extent, in the SPARC LEON processor core, crucial to the European space industry.

The ARINC 653 standard specification [1] restricts the processing time assigned to each partition, in conformity with given configuration parameters. The scheduling of partitions defined by the ARINC 653 standard is strictly deterministic over time. Each partition has a fixed temporal window in which it has control over the computational platform. Each partition is scheduled on a fixed, cyclic basis.

This allows the AIR architecture to cope with hard real-time requirements and, in a given sense, opens room for the temporal *composability* of applications.

IV. Integration of Generic Operating Systems

Porting applications to one of the RTOS one might be using (RTEMS [2], eCos [3], VxWorks [4], etc.) can be a complicated task, and definitely not an error-free one [9], [10]. 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.

A. GNU/Linux contribution

To address this portability issue, we are evaluating the approach of having one partition of such a system run *GNU/Linux*, a general-purpose operating system based on the Linux kernel, for which community efforts continuously develop applications and device drivers. In the AIR architecture, such soft real-time and/or non-real-time applications using the standard GNU/Linux interface always receive a guaranteed (albeit shared) execution time window. Such guarantee is not provided by earlier approaches combining real-time and GNU/Linux operation in the same execution platform [11], [12], [13].

The integration of GNU/Linux in the AIR architecture is shown in Figure 3. A subset of the AIR APEX (Application Executive) interface is provided, but only for management and monitoring purposes.

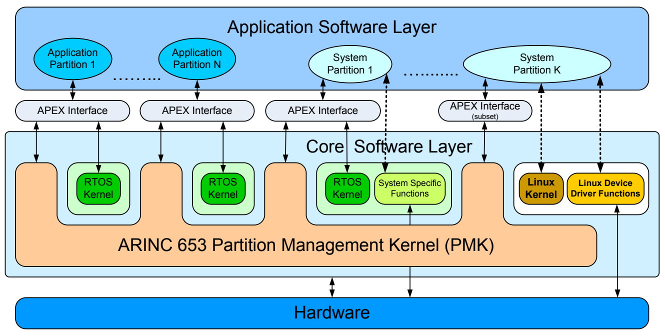

Fig. 3. Overview of the intended incorporation in the AIR system architecture

In this scenario, existent applications for GNU/Linux can be used or created without the further effort of having to port them to a particular RTOS. Another significant advantage is that the benefits of scripting languages widely used in the GNU/Linux world can be brought into scene, something which would otherwise depend on a port of the interpreter to a particular RTOS.

B. Linux state of the art

Linux is an open source operating system kernel available free of charge and maintained by developers from all the world. The source code is accessible for everyone and people are encouraged to contribute with their own code. For this reason, the Linux kernel is extremely portable between computer architectures and supports a massive variety of hardware devices and device drivers. And there's always space for more [14]. An increased modularity allows one to easily select the smallest set of features required for each system, avoiding unnecessary code. For systems with limited resources, or for very specific applications such as those found in aerospace, this may be very important. Additionally, the added support for a wider variety of hardware devices, computer architectures and the improved build tools help enhance the pace of development of the kernel itself. This flexibility makes Linux, and specially Linux 2.6, a good choice for embedded systems design, and for the provision of (soft) real-time guarantees.

Linux kernel 2.6 allows the preemption of kernel tasks, i.e. user applications are no longer locked until the end of all pending system calls before they can continue executing. This significantly reduces the latency of user applications and increases the overall system responsiveness.

The original Linux 2.6 $\mathcal{O}(1)$ scheduler [15] was further improved into the *Completely Fair Scheduler (CFS)*, which wields $\mathcal{O}(1)$ complexity for choosing a task and $\mathcal{O}(\log n)$ for rescheduling a task after it has executed and becomes ready again. This is accomplished by the substitution of runqueues for a red–black tree where the task to choose is always the root. The CFS also uses nanosecond granularity accounting (provided by the introduction of *high-resolution timers*), abandoning the notion of timeslices and the need for specific process interactivity heuristics, while still allowing to tune the scheduler to cope with different workload patterns [16].

C. Safety issues

In the scope of the integration of GNU/Linux, it is required that the GNU/Linux partition components will not contaminate the robust partitioning properties of the whole system. One important issue is guaranteeing safe preemption of the GNU/Linux partition, namely keeping the context of the running process. Another crucial point concerns the execution of privileged instructions by the GNU/Linux kernel. This would be preferably achieved through para-virtualisation techniques. Otherwise, it is required that some of the GNU/Linux kernel source code be properly changed.

V. Bullet Linux

There are a few GNU/Linux distributions available for embedded systems. However, some are commercial or targeted at a specific type of device, and others simply have too much unnecessary features or already have their own kernel modifications. Having a Linux kernel built from a standard, unpatched source tree, exactly as distributed by the developers, is extremely important. The absence of customised patches ensures easier upgradability and less compatibility issues between different versions.

Therefore, we analyse how to build a specific version of a Linux-based operating system targeted for embedded systems and applications. The main vectors for achieving an effective balance between functionality and available resources are: 1) configuration of the Linux kernel; 2) inclusion of functional features; 3) use of a smaller system library; and 4) provision of the standard Unix command interface in a more resource-efficient way. We call this design approach the *Bullet Linux*. However, instead of designing a solution totally from scratch, we follow a design-by-reuse approach, and use as much available tools as possible.

The foundations of such process have been addressed in the literature [17], [18]. Next, we describe its application, assess its effectiveness, and discuss its relevance to embedded and aerospace systems and applications.

A. Configuring the Linux kernel

The first step to produce a small kernel image for Bullet Linux exploits the configurability of the Linux kernel. The configuration of the Linux kernel is performed via a menu-driven graphical interface. In a first approach, superfluous features and device drivers were removed.

The diagram in Figure 4 illustrates the overall size difference between the Linux kernel image included in a generic distribution, and the image produced for Bullet Linux.

Generic	Bullet
2150 KiB	830 KiB

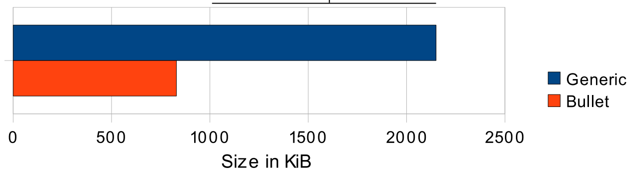

Fig. 4. Size comparison between a kernel in a generic GNU/Linux distribution and the Bullet Linux kernel

Two specific issues are worth mentioning. One is that, besides the kernel image, a standard GNU/Linux distribution ships a set of loadable kernel modules that can amount to 50 MiB[1], which were not accounted for in Figure 4, to make the comparison fairer. The other one is that the size gain illustrated in Figure 4 is both a combination of feature selection and using only built-in features.

[1]This corresponds to the prefixes for binary multiples defined in the IEC 60027-2 standard specification [19].

B. Building in functional features

The reason why no items were included as modules is that in an embedded solution they must be always present in memory. Therefore, the design choice was to build in such functionalities into the kernel.

Figure 5 highlights the exact gain obtainable by removing the loadable module support and building features in into the kernel, instead of providing them through modules (with no condition differences otherwise). The data presented were obtained by adding, to the Bullet Linux kernel, some extra functionalities (USB, Ethernet, WLAN, TCP/IP networking, PCMCIA, and Ext3 filesystem support) and the device drivers thereto associated. The Ext3 filesystem support was added on the grounds of the possibility of using an external solid-state disk through the USB and/or PCMCIA interfaces.

	Image			Modules	Total
	Bullet kernel	*Module support*	*Built-in features*		
Modular	830 KiB	225 KiB	—	1568 KiB	2623 KiB
Built-in	830 KiB	—	644 KiB	—	1474 KiB

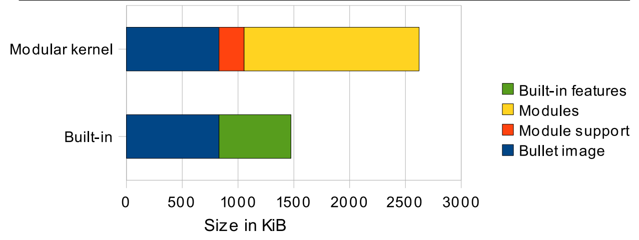

Fig. 5. Size comparison between the Bullet Linux kernel modular and built-in approaches to the inclusion of the same features

While adding features and device drivers through a built-in approach only results on an enlarged kernel image, the modular approach aggravates the total size in two fronts. On the one hand, the kernel image is enlarged to include the built-in support for loadable kernel modules. On the other hand, the kernel image must be accompanied by a set of several modules, which bear a noticeable overhead over the alternative of building those modules' features in into the kernel image.

C. Small system library

One of the most important components of a Unix-like system is the *system library*. The system library provides application programmers a comprehensive set of services.

The most used system library is the *GNU C library (glibc)* [20]. This library is targeted for generic systems, exhibiting excessive functionality (for embedded systems), a non-optimised implementation, and a large object size.

An alternative design option is *uClibc* [21], a C library specially developed for embedded systems. It features almost all GNU libc functionality, while exhibiting a small object size appropriate for systems with low memory resources. The uClibc developers have accomplished this by reimplementing it with size optimisations in mind, and by modularising some functionalities, allowing the configuration of the uClibc library and its adaptation to the requirements of the target system.

There are alternative small footprint C libraries available, such as *newlib* [22] and *diet libc* [23]. uClibc was chosen for its maturity and for how well other tools used (BusyBox, Buildroot) integrate with it.

The use of uClibc allowed Bullet Linux to keep a small size, when comparing against the use of a standard GNU libc. Figure 6 illustrates the immediate advantages brought by uClibc, showing the sizes taken up by both the GNU C library and uClibc.

glibc	uClibc
2474 KiB	368 KiB

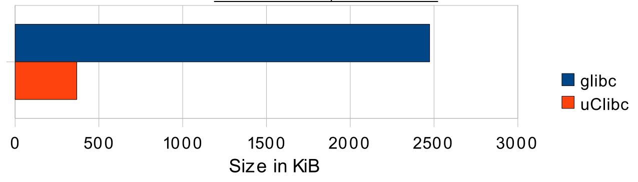

Fig. 6. Size comparison between the GNU C Library (glibc) and the uClibc

D. GNU/Linux utilities and tools

A complete Linux-based operating system needs some well-known command-line utilities and tools. Even using shared libraries, standard GNU tools can use a lot of space, which is a real problem when dealing with resources shortage. To efficiently provide this functionality, we use *BusyBox* [24], which is a set of those utilities and tools bundled together with a shell in a single executable. This approach alone reduces memory size requirements. Furthermore, the developers of BusyBox have rewritten these tools to be smaller than their original counterparts. This was accomplished by code optimisations and by the absence of some of the features, although maintaining the most important functions. Discarding unneeded sections from intermediate object files before generating the BusyBox executable allows a slight additional size gain.

BusyBox was chosen also because it is highly modular and configurable. It provides a wide array of commands (e.g. core utilities like dd, network utilities like ifconfig, or editors like sed) that can be included at this stage, some of which can be fine tuned as to only include a part of the available features.

Figure 7 shows the difference in size between a set of tools chosen for Bullet Linux (consisting, mainly, of core utilities), provided as both standalone executables and as only one BusyBox executable. The technical difference between the *BusyBox (unstripped)* and *BusyBox* executables is that the latter (which is the one included in Bullet Linux) was produced by discarding unneeded sections from the former; this process is automatically performed when compiling BusyBox.

E. Shell

BusyBox provides a few shell options, the most traditional of which is the Almquist Shell (*ash*). Although compatible with the Bourne shell and suitable for low memory systems, *ash* lacks some extras provided by other shells like the ubiquitous Bourne Again Shell (*bash*).

Standard	BusyBox (unstripped)	BusyBox
1932 KiB	440 KiB	363 KiB

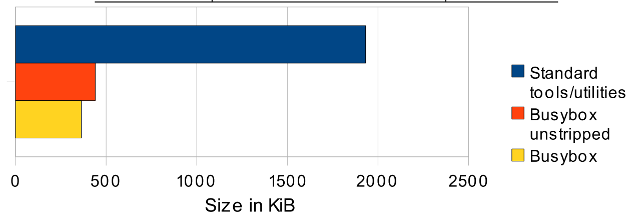

Fig. 7. Size comparison between a set of GNU utilities and tools provided both as separate executables and as a single BusyBox executable (both stripped and unstripped of unnecessary symbols)

When the use of scripting is needing, one has to evaluate if the functionality provided by *ash* is appropriate. Otherwise, a more appropriate shell can be included, as a standalone executable. This can be automated during the building process (cf. Section V-G, ahead).

F. Interpreted/scripting languages

The previous design steps of Bullet Linux leave out the support for interpreted/scripting languages. The support for interpreted/scripting languages is extremely interesting for a wide set of applications, including some of those in the aerospace domain.

BusyBox does not support any of these interpreters (save for the aforementioned shells), so support must be added as standalone executables. Once again, this can be automated during the building process (Section V-G). Currently, the available packages are: lua, microperl (Perl without OS-specific functions), python, ruby, tcl, and php.

G. Building process

Buildroot [25] is a tool suite that makes it easy to generate a cross-compilation toolchain and other resources for the target Linux system using the uClibc C library. Buildroot is specially appropriate for embedded systems engineering, being used to facilitate the configuration and build process of the uClibc system library and the BusyBox toolset. It configures builds, and prepares the cross-compiler environment for the later build of the system library and toolset. This cross-compiling environment is necessary because the target architecture for Bullet Linux may be different from the architecture of the build system.

In the specific case of Bullet Linux, the kernel was compiled from unpatched sources with a specific configuration for the existing devices and interfaces of the prototype systems (Intel IA-32-based, Ethernet network, and usually no hard-disk drive). The system library and toolset were also configured to be as small as possible, while maintaining all the important functionalities.

Bullet Linux was built with Linux kernel 2.6.26, uClibc 0.9.29 and BusyBox 1.11.13. It is extremely customisable, inheriting its main components' flexibility and modularity.

By putting together a specially configured Linux kernel 2.6 for embedded system prototyping, a restricted uClibc system library (e.g. excluding large file support), and a selected set of system tools (including the Almquist Shell, several core utilities, a few archival utilities, and no network utilities or Ext2 filesystem-related programs), it became possible to build an entire GNU/Linux operating system that can fit in as little as 2 MiB. This does not include any additional shell or language interpreter options as standalone executables.

VI. Overall results analysis

The size gain of Bullet Linux can be analysed by comparing each of its components individually with the equivalent in a desktop distribution. Typically, a desktop distribution is built with standard or lightly patched kernel compiled with a modular approach; a modular Linux Kernel is composed of an image plus a set of files that correspond to different modules. Modules are loaded into memory only if considered necessary by the system or the user. A typical modular kernel has an image size slightly above 2 MiB and a set of modules with about 50 MiB. The system library is a fully featured GNU libc 2.X (libc 6) along with many other smaller less generic libraries. The system tools in a desktop distribution are compiled against its glibc and occupy a big slice of storage space. Globally, Figure 8 summarises the analysis of size in two distinct situations: a generic Linux distribution and Bullet Linux. For comparison sake, we only present the size occupied, in a standard Linux distribution, by the same system utilities/tools and libraries included in Bullet Linux.

Kernel		**System library**		**System tools**	
Generic	***Bullet***	***glibc***	***uClibc***	***Generic***	***BusyBox***
2150 KiB	830 KiB	2474 KiB	368 KiB	1932 KiB	363 KiB

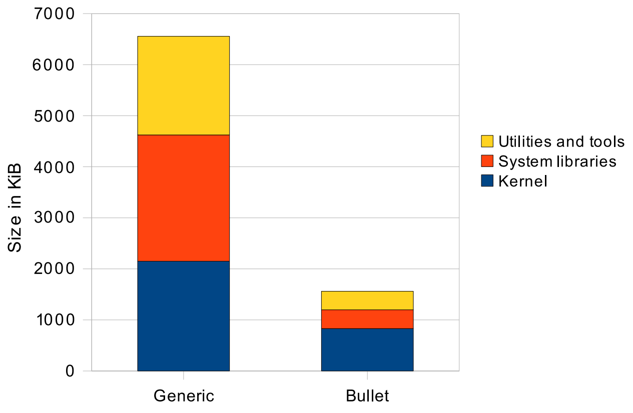

Fig. 8. Overall size comparison

The results obtained with Bullet Linux open room for its integration in the AIR architecture. The embedded solution provided by Bullet Linux allows for an implementation of Linux-based systems and applications which will always be entirely present in physical memory. No virtual memory mechanisms are required, meaning no particular memory protection scheme is needed for compliance to the ARINC 653 specification and integration in the AIR architecture.

VII. Related work

The initial approaches for introducing hard real-time processes in Linux-based operating systems were given by [11], [12] in the RTLinux and RTAI design approaches. Both solutions secure real-time behaviour through a low-level specific-purpose microkernel inserted between the hardware infrastructure and the Linux operating system kernel. The Linux kernel and applicatons run as the idle task of the real-time microkernel.

A similar approach is followed in the xLuna [13] operating system, replacing the RTAI/RTLinux microkernel with the RTEMS kernel. A xLuna microkernel mediates the interactions between the hardware and the operating system components (RTEMS and Linux). Linux runs as the lowest priority task. The xLuna has been targeted to run on the SPARC LEON processor.

The system described in [26] heavily integrates the Linux and μITRON kernels, executing Linux as one of the threads of μITRON. The proposed architecture employs a microkernel to provide protected execution environments for the embedded kernels. The microkernel provides the scheduling of embedded kernel instances. Two scheduling policies in the microkernel are used for different purposes.

In [27], the suitability of the real-time Linux variant RTAI/LXRT [12] as an operating system solution observing spatial and temporal partitioning is evaluated. Temporal segregation of real-time tasks is ensured through a time-triggered scheduler built on top of the native RTAI real-time kernel. Though such solution guarantees a given amount of processing time for each real-time task, Linux kernel and applications are only scheduled for execution when no other task is ready to run. A restrictive Application Programming Interface (API) excludes all operations that affect partitioning.

In the AIR architecture, a guaranteed execution window is provided for Linux partitions. This allows progress of Linux applications even under heavy load conditions. An evolution of the AIR technology foresees advanced improvements, with dynamic transfer of the processing budget not used by real-time partitions to non-real-time Linux partitions.

VIII. Concluding remarks

In this work, we presented the problem of integrating generic operating systems onto the AIR architecture, an architecture for aerospace applications featuring temporal and spatial segregation, based on the ARINC 653 specification [1]. This approach tackles the issue of having to port specific applications to a given RTOS.

We look into GNU/Linux in the perspective of such an integration (as a partition OS), and show the development of a fully functional operating system for embedded systems with scarce storage resources, based on the Linux kernel. The resulting solution is analysed in terms of how better it combines with the scarce resources of embedded systems compared to a standard GNU/Linux distribution.

We call this solution Bullet Linux. It sets the ground for a consolidated architecture based on the AIR technology, addressing safety issues, and integration of different partition operating systems. This architecture inherits the robust partitioning properties introduced in the design of the AIR technology [7]. The integration of Bullet Linux makes available to AIR applications a wide range of utilities, tools, language interpreters (Python, Perl, 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.

The analysis of the impact of the two-level hierarchical scheduler approach on partition and process schedulability is an already identified and approached research issue [28], [29], [30] which will be further dealt with in the next steps of the present work. This includes particular attention to the impact on the performance of the Linux applications, which will be adequately assessed during the development and tests of a prototype of the hereby presented integration.

ACKNOWLEDGMENT

This work was partially supported by FCT through the Multiannual Funding Programme. 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 — Industrial Initiative), URL: http://air.di.fc.ul.pt).

The authors would like to thank James Windsor at ESA/ESTEC for the valuable comments and interest on the present work.

The authors would also like to thank the staff at Skysoft Portugal with whom the collaboration within the AIR activities was most valuable for the pursuit of this line of work: Tobias Schoofs, Edgar Pascoal, Sérgio Santos, Cássia Tatibana, and José Neves.

REFERENCES

[1] "Airlines Electronic Engineering Committee (AEEC), Avionics application software standard interface (ARINC specification 653-2)," ARINC, Inc., 2006.

[2] *RTEMS C Users Guide*, On-Line Applications Research Corporation (OAR), Feb. 2008, edition 4.8, for RTEMS 4.8.

[3] A. Massa, *Embedded Software Development with eCos*. Prentice-Hall, 2002.

[4] *VxWorks Kernel Programmer's Guide, 6.2*, Wind River Systems, Inc., 2005.

[5] "Airlines Electronic Engineering Committee (AEEC), design guidance for integrated modular avionics (ARINC specification 651)," ARINC, Inc., 1991.

[6] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms and assurance," SRI International, California, USA, NASA Contractor Report CR-1999-209347, Jun. 1999.

[7] J. Rufino, S. Filipe, M. Coutinho, S. Santos, and J. Windsor, "ARINC 653 interface in RTEMS," in *Proceedings of Data Systems in Aerospace (DASIA'07)*, Naples, Italy, Jun. 2007.

[8] N. Diniz and J. Rufino, "ARINC 653 in space," in *Proceedings of Data Systems in Aerospace (DASIA'05)*, Edinburgh, Scotland, Jun. 2005.

[9] L. Kinnan, J. Wlad, and P. Rogers, "Porting applications to an ARINC 653 compliant IMA platform using VxWorks as an example," in *Proceedings of The 23rd Digital Avionics Systems Conference (DASC 04)*, vol. 2, Oct. 2004.

[10] L. Kinnan, "Application migration from Linux prototype to deployable IMA platform using ARINC 653 and Open GL," in *Proceedings of The 26th Digital Avionics Systems Conference (DASC 07)*, Oct. 2007.

[11] V. Yodaiken and M. Barabanov, "A real-time Linux," in *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Jan. 1997.

[12] G. Racciu and P. Mantegazza, *RTAI 3.4 User Manual, rev. 0.3*, Oct. 2006.

[13] P. Braga, L. Henriques, B. Carvalho, P. Chevalley, and M. Zulianello, "xLuna - demonstrator on ESA Mars Rover," in *Data Systems in Aerospace (DASIA'08)*, Palma de Mallorca, Spain, May 2008.

[14] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," in *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. Washington, DC, USA: IEEE Computer Society, 1999, p. 111.

[15] J. Aas, "Understanding the Linux 2.6.8.1 CPU scheduler," Silicon Graphics International (SGI), Feb. 2005.

[16] D. Hart, J. Stultz, and T. Ts'o, "Real-time Linux in real time," *IBM Systems Journal*, vol. 47, no. 2, pp. 207–220, Apr.–Jun. 2008.

[17] D. Abbott, *Linux for Embedded and Real-time Applications, Second Edition (Embedded Technology)*. Newnes, 2006.

[18] C. Hallinan, *Embedded Linux Primer: A Practical Real-World Approach*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

[19] *IEC 60027-2: Letter symbols to be used in electrical technology – Part 2: telecommunications and electronics*, IEC Std., Aug. 2005.

[20] GNU C library. [Online]. Available: http://www.gnu.org/software/libc/

[21] uClibc. [Online]. Available: http://www.uclibc.org/

[22] The newlib homepage. [Online]. Available: http://sourceware.org/newlib/

[23] diet libc - a libc optimized for small size. [Online]. Available: http://www.fefe.de/dietlibc/

[24] BusyBox. [Online]. Available: http://www.busybox.net/

[25] Buildroot. [Online]. Available: http://buildroot.uclibc.org/

[26] T. Nakajima, M. Sugaya, and S. Oikawa, "Operating Systems For Building Robust Embedded Systems," in *Proceedings of The 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS05)*. IEEE, Feb. 2005.

[27] R. Obermaisser and B. Leiner, "Temporal and spatial partitioning of a time-triggered operating system based on real-time Linux," *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pp. 429–435, May 2008.

[28] M. Coutinho, "Integração modular de dispositivos de entrada/saída em plataformas de controlo distribuído," M.Sc. thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, Dec. 2007, in Portuguese.

[29] M. Coutinho, J. Rufino, and C. Almeida, "Response time analysis of asynchronous periodic and sporadic tasks scheduled by a fixed-priority preemptive algorithm," in *Proceedings of The 20th Euromicro Conference on Real-Time Systems (ECRTS 08)*, Prague, Czech Republic, Jul. 2008.

[30] J. Rufino and J. Craveiro, "Robust partitioning and composability in ARINC 653 conformant real-time operating systems," Presented at the 1st INTERAC Research Network Plenary Workshop, Braga, Portugal, Jul. 2008, Fast Abstract.