Paravirtualization of a Real Time Operating System: Development of the AIR hypervisor with RTEMS for ARM

Carolina Serra
carolina.serra@tecnico.ulisboa.pt

Instituto Superior Técnico, Universidade de Lisboa, Portugal

October 2020

Abstract

With the increasing interest in transposing the concept of integrated modular avionics from aeronautics to the space industry, GMV developed AIR, a hypervisor that allows a single computer to run multiple applications and operating systems, maintaining strict temporal and spatial segregation through virtualization. Having originally been developed for the SPARC architecture, recent proposals for the use of ARM in space missions have led to the beginning of the migration of AIR to this architecture. It has been previously developed the support of AIR for the Arty Z7 board based on Zynq-7000 SoC by Xilinx, capable of running applications with a barebones operating system, designed to test the basic functionalities of the hypervisor. This dissertation continues the migration of AIR to ARM through the virtualization of RTEMS, the real time operating system presently adopted by ESA and NASA and elected for their future missions. The virtualization of this operating system for AIR allows the hypervisor to be used for every application developed for RTEMS, introducing Time and Space Partitioning to current and future space missions. It was successfully accomplished an improved version of AIR for ARM capable of supporting RTEMS, enabling the execution of tests of higher complexity than previously possible. 21 validation tests were executed, as well as comparative studies at functional and temporal levels between the original and the virtualized RTEMS, and between AIR for ARM and AIR for SPARC, which demonstrated the correct operation of the hypervisor functionalities.
Keywords: IMA, AIR, RTEMS, ARM, virtualization

1. Introduction

The Integrated Modular Avionics (IMA) architecture was introduced in 1995 to reduce the weight, volume and cost of avionic systems by centralizing the computing capacity into a single processing unit, while providing fault containment and system portability [1]. The success of IMA in aeronautics caught the attention of the space community, and under the efforts to transpose this concept into space avionics, GMV developed AIR, a Real Time Operating System (RTOS) that allows several safety-critical applications to run on the same processor while maintaining strict temporal and memory segregation through Time and Space Partitioning (TSP). AIR was originally developed for SPARC, the architecture adopted by the European Space Agency (ESA) for a large majority of space missions, but with recent proposals for the design of radiation hardened ARM based systems, such as NASA's High Performance Spaceflight Computing (HPSC) Processor Chiplet program solicitation and DAHLIA's project to develop an ARM-based System on a Chip (SoC) expected to achieve performances 20 to 40 times higher than the existing SoC for space, it is expected that ARM will gain relevance in the space industry in the following years, and this prediction prompted GMV to migrate AIR to the ARM architecture.

Prior to this dissertation, AIR for ARM had only been tested with a virtualized barebones OS, created specifically to test the basic functionalities of the hypervisor and with limited capabilities. Providing a virtualized RTOS, such as RTEMS, will not only allow features of higher complexity to be implemented, but also enable further testing to be performed on AIR for ARM, raising the opportunity to fix any issues that might have gone undetected so far.

RTEMS stands for Real Time Executive for Multiprocessor Systems [2]. It is an open source RTOS that supports multiple processor architectures, provides many relevant features and has been subjected to extensive testing that led to a reliable and robust operating system, and the most prevalent RTOS in space for decades.

The virtualization of RTEMS will elevate the potential of AIR for ARM as it allows it to support all of the space applications that have been developed

for RTEMS, making it suitable for integration in a variety of high profile space missions.

2. Background

In the avionics system's federated architecture, each function was designated to one computer. This modular design offers isolation between avionic functions, avoiding fault propagation and easing recovery procedures at the cost of a higher weight, volume and cost. The concept of Integrated Modular Avionics was introduced as an alternative solution, applying virtualization to avionics systems.

2.1. Virtualization

Virtualization refers to the separation of a service request from the underlying hardware that delivers that service. Through virtualization, multiple Virtual Machines (VMs) can operate in a single computer, managed by the Virtual Machine Monitor (VMM). Each virtual machine is an efficient and isolated duplicate of a real machine, capable of running its own OS [3]. This architecture made it possible to overcome the limitations of conventional processors, characterized by two modes of operation: Supervisor, a privileged mode with access to all instructions, and User, a non-privileged mode with access to a subset of the instructions. Although successful in plenty computer systems, conventional processors are limited to only running one OS at a time, and software that requires direct access to privileged instructions is unable to be transposed into this architecture.

With the introduction of virtualization, the VMM (or hypervisor) runs at a higher privilege level than the OS, which would usually run at supervisor mode. The VM should be equivalent to the real machine it emulates, meaning the behavior of an application running on top of the VMM should be identical to the behavior of that same application running with no VMM. The only exceptions to this condition are the timings of execution and the system resources availability. The virtualization method that will be used in this dissertation is paravirtualization, in which the guest OS is modified by replacing the privileged instructions with hypervisor calls (HVCs) that communicate directly with the hypervisor [4]. The HVCs interrupt the VM with a jump to the hypervisor, which performs the service requested on their behalf, allowing information to be passed to and from the hypervisor.

2.2. Integrated Modular Avionics

The concept of IMA consists on centralizing the computing capacity into a single processing unit, using virtualization to separate the software functions from each other as well as from the hardware [5]. The AEEC has published a series of specifications in an effort to standardize the access to shared resources in this architecture, among which is the ARINC 653 – Avionics Application Software Standard Interface, which describes the required and recommended properties of an IMA OS [6] through the definition of the application executive (APEX), an added software layer that separates the application from the operating system.

IMA systems are typically characterized by a high fault tolerance, allowing a defective module to be replaced without harming the rest of the system; technology transparency due to the standardization efforts, which in turn supports application portability; scalability, enabling its application to systems of various sizes and supporting future growth; and system reconfigurability. These characteristics are also highly desired in space avionics, and in 2010 the IMA for Space (IMA-SP) program was started. Under this initiative, GMV developed the AIR hypervisor.

Partitioning

One of the main purposes of the hypervisor in an IMA system is to support multiple avionic applications, independent from each other. To ensure this separation of the applications, the ARINC 653 specification describes the concept of Time and Space Partitioning (TSP). Each partition is a program in a single application environment, which encompasses their individual data, context, configuration attributes and other partition specific information. The hypervisor allocates a fixed time slot for each partition (**time partitioning**) as well as a predetermined area of memory (**space partitioning**), and the partition is prohibited to access resources that are not assigned to it. The communication between partitions is established through ARINC 653 services.

Fault Detection and Response

ARINC 653 defines the Health Monitor (HM) to monitor and report faults in hardware, application and OS, providing isolation and adequate recovery. The HM uses configuration tables to decide the adequate response to a certain fault. Faults can occur at process, partition or module level, and the HM response should be performed within the level it occurred so as to not violate partitioning.

3. Technologies

3.1. RTEMS

Real Time Operating Systems are characterized by the concept of **predictability**, meaning the degree to which the times of execution are guaranteed within minimal variations [7]. The predictability of RTOS is crucial for safety-critical, real time space applications, in which a failure or delay may have catastrophic consequences. Some of these applications include the software that monitors and controls the spacecraft's health, position and response

to harsh environment conditions.

RTEMS has been the chosen RTOS for several space missions, such as the NASA Solar Dynamic Observatory, and the Electra UHF antenna of the Mars Reconnaissance Orbiter. It is an open source RTOS that supports multiple processor architectures, including all the processors currently used by ESA and NASA, and provides many relevant features such as multitasking capabilities, intertask communication and synchronization protocols, priority-based preemptive scheduling and interrupt management.

Internal Architecture

RTEMS is organized as a set of layered components that provide services to an application. These services are accessible to the application through resource managers, which are executive interfaces formed by grouping directives into logical sets. The RTEMS core provides functions that are used by multiple of these components, such as scheduling, dispatching and object management.

Tasks

As defined by RTEMS, a task is the smallest thread of execution which can compete on its own for system resources. RTEMS supports both periodic and sporadic tasks. Each task has a priority level, ranging from 1 to 255, which is used by the scheduler to determine which ready task will be executed.

Scheduling

The scheduler is the component responsible to allocate the processor time to the various competing tasks. The allocation of the processor for each task is done by the dispatcher, which retrieves the control of the processor from the current task and passes it to the next by performing a context switch. The context switch consists on saving the context of the current task and restoring the context of the task that has been allocated to the processor. The context includes all of the necessary information to ensure that the task is capable of returning to execution without suffering any change by the interruption. The most common scheduling algorithm is Priority Scheduling, which allocates the processor to the ready task with the highest priority.

System Configuration and Build

RTEMS provides tools to assist the build process by automatically generating all the necessary makefiles. RTEMS is configured for each application through a set of macros to automate the generation of the data structures used at the system initialization. These macros encompass information such as the length of each clock tick, the application initialization tasks, the task scheduling algorithm and the device drivers needed by the application. The configure program generates a variety of files based on these macros, including the necessary Makefiles, to build and install RTEMS with the correct configurations.

3.2. AIR

AIR stands for ARINC 653 Interface for RTEMS, and as the name implies, it was developed as an adaptation of RTEMS to implement the ARINC 653 standard. It is a real time TSP hypervisor, allowing multiple VMs to run on a single processor while mantaining temporal and spacial isolation between applications.

Architecture

AIR consists of four development independent modules, depicted in Figure 1:

- The **Partition Management Kernel (PMK)** holds the main functionalities of the hypervisor. It implements the temporal and spacial segregation, the scheduling initialization, the interrupt handling and the context switch between partitions.
- Each partition has its own **Partition Operating System (POS)**, a paravirtualized RTOS capable of features such as scheduling tasks and managing virtual interrupts.
- The **LIBS** module consists of a set of libraries that implement functions to assist the connection between the modules, such as the IMASPEX library, which implements the APEX interface in conformity with the ARINC 653 standard.
- The **AIR toolchain (TOOLS)** is a parallel module coded in python to aid the build process.

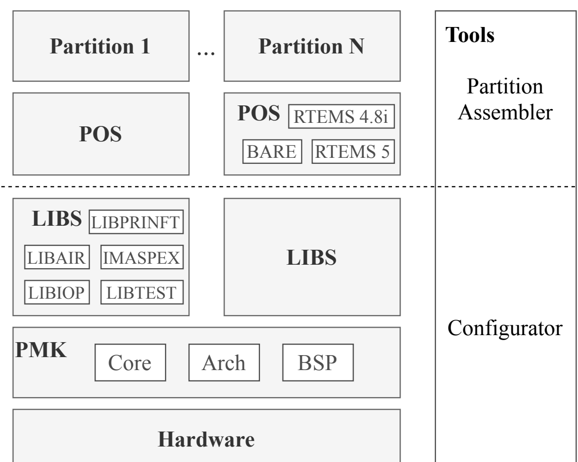

Figure 1: AIR architecture.

Health Monitor

As defined by the ARINC 653 standard, a fault in the execution should be handled by a Health Monitor (HM). Figure 2 depicts the HM procedure. The HM handler starts by searching the error id and the operational state in the system HM table to check whether the fault occurred at the module or partition level. It then searches for the error ID in the table corresponding to that error level to obtain the action that should be performed. These tables are defined by the user in the application configuration XML file.

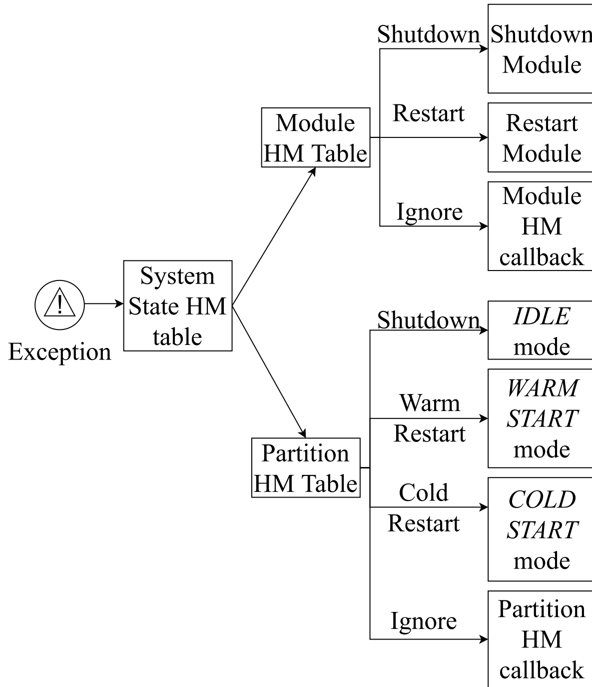

Figure 2: AIR HM flow chart.

3.3. ARM

ARM is a family of Reduced Instruction Set Computing (RISC) processor architectures that started gaining traction in the 90's and is currently leading the SoC industry, holding a share of 96% of the mobile market [8]. This dissertation is targeted at the Cortex-A9 MPCore processor equipped in the Arty Z7 board provided by GMV, which implements the ARMv7-A architecture.

Program Status Register

The Program Status Register is a 32-bit processor register that holds information on the processor status and control, such as the processor mode, the instruction set flags, the interrupt flags and condition flags.

The current state of the processor is stored in the Current Processor State Register (CPSR), and each mode (except for User and System modes) has its own Saved Processor State Register (SPSR) which holds the previous CPSR from before changing to the current mode, so that the previous context can be restored.

Operating Modes

The ARMv7 basic model operates in two privilege levels: PL0, the lowest privilege level, in which an exception is raised if there is an attempt to access privileged resources, and PL1, in which an operating system is expected to run. There are seven basic operating modes, each with its own stack pointer (SP) and link register (LR):

USER is the mode most applications are expected to run at, and is the only mode executing at PL0;

SUPERVISOR is the standard entry mode of operation, and can also be entered through Supervisor Calls (SVCs);

SYSTEM mode is used to access the USER registers without sacrificing the access to privileged resources;

ABORT mode is entered through Data Abort and Prefetch Abort exceptions, raised when the CPU attempts to access data or fetch an instruction from illegal memory locations, respectively;

UNDEFINED mode is entered through undefined exceptions, raised when an instruction is not recognized by the processor;

IRQ is the Interrupt Request Mode;

FIQ is the Fast Interrupt Request Mode.

Exception Handling

Exceptions, also referred to as traps, are anomalous or exceptional events that interrupt the normal execution and require special processing. When an exception is raised, the execution jumps to a predefined table of handlers.

While in other architectures the exceptions are handled through a single trap table (such as SPARC, with a 256 entries trap table), ARM handles the exceptions through multiple redirection levels: a first main trap table contains the 7 main exceptions that can be raised, presented in Table 1, and redirects the execution to a **low level handler**, coded in assembly. Within the 7 main exceptions, there are several possible interrupt sources, such as timers, device drivers and different faults. The low level handler calls a C subroutine, or a **high level handler**, to identify the interrupt source and perform the adequate actions.

The information needed to manage the interrupts is mapped in the General Interrupt Controller (GIC) registers [9]. In a virtualized environment, the virtualized OS running in a non privileged level should not access the real GIC blocks, so the necessary registers must be copied into the VM.

4. Implementation

Prior to this dissertation, the migration of AIR to ARM had already accomplished some of the

Exception	Mode
Reset	Supervisor
Undefined Instruction	Undefined
Supervisor Call	Supervisor
Prefetch Abort	Abort
Data Abort	Abort
Interrupt Request (IRQ)	IRQ
Fast Interrupt Request (FIQ)	FIQ

Table 1: ARM exceptions.

PMK functionalities, such as the memory translation between virtual and real memory, the hypervisor scheduler, partition switching, AIR exception handling, and the implementation of several system calls. However, due to the simplicity of a barebones OS, some important functionalities were not implemented or lacked testing. For instance, AIR did not have a mechanism to run the guest OS's virtualized exception handlers, and consequently the guest OS's scheduler and multitasking were not supported. Thus, in order to run RTEMS on AIR for ARM, it is necessary to both virtualize RTEMS and apply modifications to the PMK, and this will be achieved through a sequential and iterative workflow.

4.1. System Configuration

The first step to run RTEMS on AIR for ARM is to add it as a supported POS in the AIR toolchain, so that the AIR configure script can generate the necessary files to compile the system and run the application while incorporating the RTEMS configuration and build process. Since RTEMS is already a supported POS for SPARC, at this point the implementation consists only on migrating the RTEMS configuration and makefile template from SPARC and applying it to ARM by changing the name of the target BSP.

Before building the system it is also necessary to provide the partition linkcmds file, which is a script that specifies the memory regions in which the code and data of the application will be written to. The memory assigned to a partition was contained between the addresses 0x10000000 and 0x10200000, providing each partition 2MB. The memory regions attributed to the partitions correspond to virtual memory, which will be translated into physical memory by the MMU.

4.2. System Initialization

The initialization of the AIR hypervisor is responsible for starting the BSP and initializing the hypervisor features such as the scheduler, spacial segregation, HM and, if available, multicore. Once the PMK is ready to run the applications, the execution enters the entry point of the partition assigned to the first minor time frame by the scheduler. If the OS assigned to that partition is RTEMS, then this entry point corresponds to the start of the RTEMS initialization.

Low Level Initialization

The RTEMS initialization is performed at two levels, low and high. At the lower level, it consists on the start code present in the start file. This file is typically written in assembly and is specific for each architecture, performing the minimum actions possible that enable the processor to run C code correctly. As in a virtualized environment the board is already initialized when entering a partition, most of the low level initialization required to run RTEMS directly on the HW is redundant and unnecessary when running on top of AIR. Therefore, most of the low level initialization can be removed, greatly simplifying the code. Only two functions were kept: the routine to clean the .bss, to ensure that the statically allocated variables are initialized with the value 0; and a call to the boot_card() function to complete the initialization at a high level. In the original RTEMS, all this function does is prepare to pass the control to the Initialization Manager and call the initialization directive rtems_initialize_executive(), responsible for passing the control of the processor from the OS to the user application. In a virtualized environment, RTEMS still needs to set the virtual trap table.

Virtual Trap Table

When running RTEMS on top of AIR, the exceptions should be handled by the hypervisor. When an exception is raised, it will be caught by the AIR trap table and handled by the AIR handler. If AIR is not informed of a virtual trap table address, then after handling the exception, it will resume the partition from the same point where the exception was raised. The RTEMS handlers will not be executed, which means that the POS will not be aware that an exception occurred. This affects several functionalities. In the case of timer interrupts, not performing the RTEMS clock handler means that its scheduler will not be activated at regular intervals, and the task management will be severely limited. It is therefore imperative to set a virtual trap table using the AIR_SYSCALL_SET_TBR, to follow the procedure presented in Figure 3.

The original RTEMS for ARM defines a main trap table at at the start.S file, so it is merely necessary to use the AIR system call to set this table as the virtual trap table.

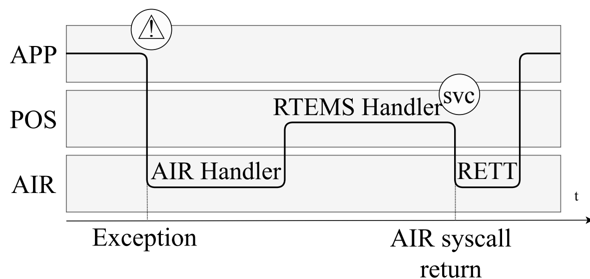

Figure 3: AIR exception handling procedure with a set virtual trap table.

Initialization Manager

The rtems_initialize_executive() directive is responsible for preparing the system to pass control to the user application, and will initialize the RTEMS features that are configured for that particular program through a system initialization linker set. Most of these features do not require virtualization. The two services that do require modifications are the routine to start the BSP, which initializes the interrupts, and the routine to initialize all the device drivers.

The interrupts are initialized in bsp_interrupt_facility_initialize(), a function that is specific for each BSP. In ARM, this function accesses the GIC registers to enable and set the priority of the interrupts. The GIC should not be accessed in user mode, and since the interrupts should be managed by the hypervisor, these accesses can be removed.

At this point, RTEMS also replaces the default IRQ exception handler by an alternative low level handler. These default handlers are simple routines coded in assembly that assume that the exception was an error and therefore terminate the program. However, it might be desired that the exception handler calls a high level handler, adequate for the particular interrupt that occurred. For instance, the IRQ handler should distinguish a timer interrupt and redirect the execution to the clock handler. In those cases, RTEMS provides alternative low level exception handlers, prepared to store the context, redirect to a high level handler and finally restore the context to return to the normal execution. The function that installs the alternative handlers was virtualized by removing the privileged accesses.

As for the initialization of device drivers, the only driver that should be configured is the clock. All the remaining I/O drivers should be managed by AIR through the IOP, and the header file containing the RTEMS configuration macros that was generated by the AIR configurator does not include any other device.

The clock initialization consists on, first, installing the Clock_isr() routine as the exception handler for the Global Timer (GT) interrupt. Then, the GT registers are accessed in order to initialize the RTEMS timecounter. With AIR, instead of accessing the real processor registers, an AIR system call is used to get the time elapsed since the beginning of the partition and set the AIR timecounter with the appropriate data.

4.3. User Application

The application's init.c is generated by the configurator through init.c.mako template. This file is part of the user application, designed to run in an unprivileged mode, so it does not require virtualization. However, since it was intended for SPARC, some modifications need to be applied so that it can run on ARM as well.

The Init() function, before redirecting the execution to the entry_point defined in the configuration, installs a HM handler so that the application programmer can change the HM Callback as desired. When AIR only supported the SPARC architecture, the HM handler was installed through a function that is not defined for ARM. This function was replaced by a new routine that, according to the target architecture that was configured, performs the right functions to install the HM handler. In ARM, this requires installing low level handlers for the data abort, exception abort and undefined instruction exceptions, followed by installing the desired HM Callback as a high level handler.

4.4. Exception Handling

When an exception is raised, the execution jumps to the trap table in AIR, which branches to the AIR Exception Handler. This handler starts by storing the processor context, or the Interrupt Stack Frame (ISF), in the IRQ stack. Then, AIR performs the adequate actions to handle the exception.

After the AIR handler, if the virtual trap table is not initialized, AIR does not recognize a virtual interrupt. The context will be restored from the stack, moving the IRQ SP back to the top of the ISF so that, when the next exception occurs, its context will replace the one that has already been restored. The execution returns to the application.

If AIR recognizes a virtual exception, then before branching to the virtual trap table address to perform the RTEMS exception handlers, as shown in Figure 3, the processor registers take the values that are stored in the IRQ, but the SP remains at the bottom of the context. This way, if another exception occurs inside the virtual handler (for instance, one of the several SVCs that are raised inside the virtualized RTEMS IRQ handler), it will be stored below the previous exception, not corrupting the context that has yet to be recovered.

RTEMS Exception Handlers

After the AIR exception handler sends the execution to the virtual trap table, the RTEMS exception handler will be performed. Currently, this applies to IRQs, aborts and undefined instructions, and the handlers for these exceptions required virtualization. The virtualization of the handler functions is similar, and mostly consists on preventing the processor to access the PSR registers, either by replacing these accesses with supervisor calls or, if these accesses are unnecessary when running on top of AIR, by removing them entirely. The virtualization of the IRQ handler presents more challenges, as it uses both the SVC stack and the IRQ stack to store data. When running on AIR, the whole function will run on user mode, therefore requiring adjusting the stack pointer to ensure that the data is not corrupted.

After storing the context in the low level handler, the high level handler is called, being either the HM handler for the aborts and undefined exceptions or, for IRQs, the handler that was installed for the interrupt that occurred. The original RTEMS accesses the GIC registers to get the interrupt ID from the Interrupt Acknowledge Register (IAR), and the AIR_SYSCALL_ACKNOWLEDGE_INT was implemented in AIR for ARM to replace this access to the GIC and get the interrupt ID from the virtual IAR instead. Currently, the only IRQ that is being passed from AIR to the RTEMS handler is the Global Timer (ID=27), taking the execution to the clock handler. After completing this handler, RTEMS then verifies if a context switch is necessary by checking the thread dispatch state. If so, the control is passed to the dispatcher to perform the context switch.

Finally, the low level handler is concluded by restoring the previous context and returning from the exception, which is done through AIR by calling AIR_SYSCALL_RETT.

Clock

The precise timing responses that characterize RTOS are achieved both in AIR and RTEMS through periodic timer interrupts. When running on top of AIR, since the interrupts are managed by the hypervisor, the time counting in RTEMS must be done through an AIR service rather than by accessing the real processor timer registers. AIR provides the AIR_SYSCALL_GET_ELAPSED_TICKS routine, which gets the number of ticks elapsed since the partition's initialization.

The RTEMS clock handler is Clock_isr(), which is hardware independent, making its virtualization for the SPARC architecture applicable to ARM as well. It uses macros to call BSP specific functions, which were virtualized to use the AIR system call to update the RTEMS timecounter, so that the RTEMS scheduler correctly allocates the execution time to the right task.

Return from Exception

The virtualized exception handlers end by calling AIR_SYSCALL_RETT in order to perform a return to normal execution. While other exceptions need to store the context in order to return to the previous one after the exception is handled, AIR_SYSCALL_RETT does not need to return to the context before the SVC was called, but rather the context stored before that, corresponding to the exception from which the return is intended. The code that stores the context can therefore be skipped, and once the execution reaches the context restore, it will recover the context that was stored before the SVC, as intended unless RTEMS performed a context switch.

Context Switch

In an application with multitasking, RTEMS performs context switches in the cases in which the scheduler deems it necessary to change the task that is allocated to the processor. This context switch is performed inside the RTEMS IRQ handler, and does not require virtualization. However, without accounting for this feature, AIR is not aware that a context switch occurred, and in the return system call, it will always restore the last context regardless of the context that RTEMS is trying to recover. This will lead to a different behaviour than the desired.

Since RTEMS is running exclusively in user mode, the contexts inside the OS are stored in the user stack. In context switches inside the OS, RTEMS changes the user SP to point to the context of the next task. This means that it is possible to verify if RTEMS performed a context switch in AIR_SYSCALL_RETT by comparing the current user SP with the user SP stored in the previous context. If different, then AIR knows that the OS performed a context switch and is trying to recover a different context. AIR can then go through all the stored contexts to find the one with the user SP that RTEMS means to restore.

The problem with this solution is that after the context switch, the next exception will be stored in the middle of the stack instead of the bottom. If the handler of this exception calls SVCs, then the context of the SVC will replace data that might have yet to be recovered. This issue was solved by leaving enough space for another context between two consecutive IRQs. This way, when there is an SVC inside the IRQ handler, its context will be stored in this space and will not corrupt the contexts of the IRQs below that have yet to be restored.

4.5. End of Test
When the test ends, RTEMS shuts down the hardware, which must not happen in a virtualized environment. AIR provides a shutdown SVC that should be called at the end of the user applications to terminate AIR and shutdown the machine.

RTEMS may also attempt to shut down the machine when a process level error occurs. In this case, the shutdown system call should not be used, as a process level error should not affect the other partitions or terminate AIR. Instead, the error should be caught by the HM, using the AIR_SYSCALL_RAISE_HM_EVENT.

5. Evaluation

The tests performed to validade the hypervisor can be divided into three categories: RTEMS Testsuites, AIR examples and AIR validation tests. The ARM tests were executed in the QEMU emulator and the Arty Z7 board.

5.1. RTEMS Test Suites
There are over 600 tests comprised in the RTEMS test suites, and without an automatized facility to migrate these programs to AIR and execute the tests, performing all of them would be a time consuming and exhaustive job, so only a six were selected: 4 sample tests to start the virtualization (hello, nsecs, ticker and paranoia), and 2 tests recommended by the RTEMS organization (pthread test, designed to test the thread creation in conformity with the POSIX standard, and spcontext, further explained ahead). From a functional level, all the sample tests and the pthread test performed as expected, showing the same behavior in the original and the virtualized RTEMS.

The **ticker test** will be further discussed as it was used to perform a basic timing analysis. This test consists of 3 tasks that periodically print the time:

- **TA1**, printing the time every 5 seconds;
- **TA2**, printing the time every 10 seconds;
- **TA3**, printing the time every 15 seconds.

The test was migrated into AIR partitions and configured with different frequencies. Table 5.1 presents the results printed with the ticker test running in 2 partitions (P1 and P2), alternating at every second, with a frequency of 100 ticks per second (T/s).

These results demonstrate that the partitioning did not affect the timings of execution, each partition presenting the same timings as the original RTEMS.

To analyze the influence of the frequency in the results, the test was repeated in a single partition configured with varying differences. With the same frequency as the original RTEMS, at 100 T/s, it can be observed that the timings correspond exactly to the expected. At different frequencies, however, the results oscillated, presenting errors of up to 18ms that require further investigation. It should be noted that these results were obtained from console prints, which is not a reliable method for time analysis as the prints themselves may significantly affect the timings. It is therefore imperative to acquire tools to perform a more accurate time analysis before implementing this software in safety critical real time systems.

The **spcontext test** creates three tasks of different priorities and different FPU contexts, and activates a timer that switches the task priorities in periodic time intervals. The scheduler will therefore perform context switches, and the application

Task	RTEMS	Virtualized RTEMS	
		P1	P2
TA1	0,00	0,00	0,00
TA2	0,00	0,00	0,00
TA3	0,00	0,00	0,00
TA1	5,00	5,00	5,00
TA2	10,00	10,00	10,00
TA1	10,00	10,00	10,00
TA1	15,00	15,00	15,00
TA3	15,00	15,00	15,00
TA2	20,00	20,00	20,00
TA1	20,00	20,00	20,00
TA1	25,00	25,00	25,00
TA2	30,00	30,00	30,00
TA1	30,00	30,00	30,00
TA3	30,00	30,00	30,00

Table 2: Times printed (in seconds) in the ticker test, with a frequency of 100 T/s and two partitions.

Task	50 T/s	100 T/s	200 T/s	500 T/s
TA1	0,00	0,00	0,000	0,000
TA2	0,00	0,00	0,000	0,000
TA3	0,00	0,00	0,000	0,000
TA1	5,02	5,00	5,005	5,002
TA2	10,02	10,00	10,000	10,002
TA1	10,02	10,00	10,005	10,006
TA1	15,02	15,00	15,005	15,010
TA3	15,02	15,00	15,000	15,004
TA2	20,02	20,00	20,000	20,004
TA1	20,02	20,00	20,005	20,014
TA1	25,02	25,00	25,005	25,016
TA2	30,02	30,00	30,000	30,006
TA1	30,02	30,00	30,005	30,018
TA3	30,02	30,00	30,000	30,006

Table 3: Times printed (in seconds) in the ticker test running on AIR for ARM, with varying frequencies and a single partition.

verifies whether the context was correctly restored. This test was used to develop and validate the context switch solution in the AIR exception return. Through this test it was possible to verify that this solution performs correctly, saving and restoring the correct contexts without corrupting data.

5.2. AIR Examples

The examples provided by AIR allow the testing of basic hypervisor functionalities such as the inter-partition communication (ports example), the shared memory (shm example) and the creation of periodic tasks (periodic example). All of these tests performed as expected, demonstrating that the modifications applied to AIR did not negatively affect these functionalities. The **HM example** was used to test the possible responses to various faults, and both the PMK and the partition HM behaved as expected.

The **time example** was used to compare the speed of the barebones OS and the virtualized RTEMS running on AIR. The results are presented in Table 5.2, in which t_{BARE} and t_{RTEMS} represent the times printed in the test running on a barebones OS and on RTEMS, respectively. Δt corresponds to the difference between the current and the last time measurements.

t_{BARE}	t_{RTEMS}	$\frac{\Delta t_{RTEMS} - \Delta t_{BARE}}{\Delta t_{BARE}}$
3	9	2,0000
5136	5709	0,1105
15403	17103	0,1098
41059	45585	0,1101
76976	85451	0,1099
77303	85814	0,1101
92700	102902	0,1098
108096	119994	0,1102

Table 4: Times printed (in milliseconds) in the time example of the ARM unit tests.

From these results it is possible to observe that RTEMS is consistently slower than the barebones OS, which is expected, considering that both the initialization and the exception handling in RTEMS are significantly more complex than the barebones OS's. After the initialization, RTEMS consistently takes added 0,11ms for each ms of the execution of the barebones OS. This consistency is a good indicator of the predictability of RTEMS.

The **hello_world example** was used to compare the timings of execution of AIR for ARM (in the Arty Z7 board) and AIR for SPARC (in the GR740 board). In this test 3 partitions print the time at intervals of 0,1 seconds. The results are presented in Table 5.2.

There is a considerable difference of up to 50ms between the results obtained in SPARC and in ARM, which might be due to the fact that AIR for SPARC is compiled using optimization flags to improve its performance, while AIR for ARM is currently being compiled without optimization options to ease the debugging. Once AIR for ARM achieves a higher maturity, these options should be explored for a more accurate performance comparison. In ARM, the timings of the three partitions were exactly the same, showing that, despite the lower speeds than SPARC, the system seems to behave in predictable timings, which is the main concern in real time applications.

GR740 (SPARC)			Arty Z7 (ARM)		
P1	P2	P3	P1	P2	P3
0,004	0,006	0,006	0,010	0,010	0,010
0,104	0,104	0,104	0,116	0,116	0,116
0,204	0,208	0,204	0,222	0,222	0,222
1,008	1,008	1,008	1,044	1,044	1,044
1,108	1,108	1,108	1,150	1,150	1,150
1,208	1,208	1,214	1,256	1,256	1,256
2,008	2,008	2,008	2,044	2,044	2,044
2,108	2,108	2,108	2,150	2,150	2,150
2,208	2,208	2,208	2,256	2,256	2,256
3,008	3,008	3,008	3,044	3,044	3,044

Table 5: Times printed (in seconds) in the Hello World example provided by AIR.

5.3. AIR Validation Tests

The qualification of AIR for space applications by a recognized entity such as ESA would bring significant value to the hypervisor. To achieve this in the future, extensive testing is required, and the AIR validation and qualification project was started to implement a set of tests with a nearly total code coverage with the goal of validating the required functionalities of a TSP hypervisor. Currently the AIR validation tests comprise 38 tests designed for the SPARC architecture. 21 of these tests have been successfully employed on AIR for ARM, presenting the same results in both supported architectures.

The remaining validation tests that still need to be investigated and adapted to the ARM architecture include tests that fail due to timing constraints, since AIR for ARM presents different timings than AIR for SPARC as previously discussed; tests that rely on hardware specific language to induce faults in order to test the HM; and tests that use cache register handling system calls that have not yet been implemented in AIR for ARM.

6. Conclusions

Prior to this dissertation, AIR for ARM only supported a simple barebones guest OS that lacked important functionalities for the deployment on high

profile space missions. The goal of this thesis was to further develop the ARM BSP for AIR by allowing it to run RTEMS, the RTOS currently adopted by ESA and NASA in a variety of space missions. This objective was successfully achieved through the virtualization of RTEMS, and through applying the necessary modifications to the AIR hypervisor to ensure that it does not interfere with the behavior of the guest OS.

From a functional standpoint, the virtualization of RTEMS was successfully accomplished. The changes on RTEMS achieved a virtualized OS that behaves as the original RTOS running in a non virtualized environment. This virtualization made AIR for ARM capable of running single core applications designed for this OS, including a wide range of tests that were previously not available in this target architecture due to the simplicity of the barebones OS and that allow the further validation of the hypervisor. These tests include RTEMS Testsuites, AIR examples and the validation tests developed to test AIR for SPARC with a nearly total code coverage. Of the 38 validation tests currently executing in AIR for SPARC, 21 were successfully deployed in AIR for ARM. A comparison between the two architectures currently supported by AIR showed that AIR for ARM achieved the same results regarding the hypervisor functionalities as AIR for SPARC.

From a temporal perspective, the virtualized RTOS allowed for a more detailed analysis of the timings of execution, showing the limitations that the ARM BSP still presents and that require further investigation through the usage of profiling and evaluation tools capable of more accurate timing analysis.

Additionally to the software development, the iterative process used to achieve the stated goals of this dissertation was documented step-by-step, providing a resource that can be used for guiding the virtualization of RTOS in future projects.

6.1. Future Work

While this dissertation further developed the ARM BSP for the AIR hypervisor, there is still work to be done in order to achieve the maturity that the SPARC BSP presents. The three next steps, currently in progress, are virtualizing the RTEMS support for multi-core applications, expanding the number of device drivers supported by AIR for ARM, and executing the 17 remaining validation tests that have yet to run successfully on AIR for ARM, either by adapting them to the ARM architecture or implementing the missing system calls in AIR for ARM.

Although from a functional perspective the behavior of the software corresponds to the expected, the timings of execution still require further study, and in that prospect, it is crucial to acquire tools for a more accurate timing analysis, such as RapiTime or VectorCAST, and apply them to AIR to ensure its timely behavior.

In the long term, the qualification of AIR for space applications by ESA would bring value and recognition to the hypervisor. To achieve this, extensive testing is needed, and therefore tools to automatically test, validate and verify AIR should be implemented. A possible tool that is being studied is the Continuous Integration and Continuous Development (CI/CD) tools built into GitLab Runner, that can be used to perform scripts and send the results back to GitLab. The possibility of taking advantage of GitLab Runner along with tools to generate validation tests with the highest code coverage achievable would provide a fast, easy and reliable platform for effortlessly weaving the testing into the development process, allowing the immediate detection of issues and assisting their solution.

References

[1] H. Butz. Open integrated modular avionic (IMA): State of the art and future development road map at airbus deutschland". *1st International Workshop on Aircraft System Technologies*, 2010.

[2] RTEMS Project and contributors. RTEMS documentation project, 2018. [Online] https://docs.rtems.org/. Accessed: 24 February 2020.

[3] R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 1970.

[4] VMWare. Understanding full virtualization, paravirtualization, and hardware assist. White paper, March 2008.

[5] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to Integrated Modular Avionics. *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, 2007.

[6] Airlines Electronic Engineering Committee. Avionics Application Software Standard Interface Part 1 - Required Services, December 2005.

[7] M. Cheng. A Predictable Real Time Operating System. University of Victoria, October 2003.

[8] I. Thornton. ARM: Investing for future growth, ARM limited Q1 2019.

[9] ARM Limited. ARM ® Architecture Reference Manual ARMv7-A and ARMv7-R edition ARM Architecture Reference Manual, 2018.