Hypervisor Board Support Package migration: SPARC and ARM study case

Luis Murta
Instituto Superior Técnico
Lisbon, Portugal
luismurta-at-tecnico.ulisboa.com

***Abstract*—The number of space activities using an ARM target is growing. With both NASA's High Performance Spaceflight Computing program and ESA's DAHLIA board developing ARM-based radiation hardened boards for deep space flight, and the beginning of lower earth orbit nano-satellites using consumer-grade ARM SoCs, a new space market is in its infancy. AIR is a TSP hypervisor implementing the ARINC 653 standard developed for ESA's last generation satellites, running SPARC-based computers. GMV is now seeking new ARM BSPs for its hypervisor. With the objective of reducing the time to market, the method proposed is to migrate the BSP developed for SPARC to ARM. In the end, this thesis accomplishes a well documented side-by-side comparison of the two architectures and it succeeds in broadening the BSP portfolio of AIR with an Arty Z7 board based on Zynq-7000 SoC by Xilinx.**

***Index Terms*—hypervisor, SPARC, ARM, AIR, virtualization, Zynq-7000**

I. Introduction

The European Union (EU) has launched the COMPET-1-2016: Technologies for European non-dependence and competitiveness call [1], under the Horizon 2020 initiative. As a result of this call, the DAHLIA [2] project, a collaboration between STMicroelectronics, Airbus Defence and Space, Integrated Systems Development, NanoXplore and Thales Alenia Space, has been chosen to also create a radiation-hardened ARM-based System on a Chip (SoC), to be used in future space missions by Europe. Expected to perform 20 to 40 times faster than current SoC for space and more than twice as fast as the LEON4 chip, a quad-core processor based on Scalable Processor ARChitecture (SPARC) and the lastest innovation from Gaisler Research, the ARM-based SoC will be able to run Guidance, Navigation and Control (GNC) algorithms and handle Global Navigation Satellite System (GNSS) data and telemetry through integrated peripherals, all on the same chip [3].

Portugal is also eager to enter this new market, recently demonstrated by the creation of the Portuguese Space Agency [4] in March 13, 2019. Led by Chiara Manfletti, the space agency's President, it defined as the agency's priority to implement a shared space strategy with all shareholders and stakeholders in Portugal's space market. It is also responsible for the development of the space port in the isle of *Santa Maria*, in Azores [5]. There is also the Infante Project [6], an 100% Portuguese technology satellite, approved in 2017 by the *Agência Nacional de Inovação* (ANI), and expected to launch in 2020, where GMV takes part in the consortium.

Succeeding the emerging market of nano satellites, with the possibility of new processor architectures appearing in the space segment, and the introduction of a radiation-hardened ARM SoC, has captured the attention of GMV into opening their previously mono-architecture operating system (OS) for additional processor families, such has ARM, PowerPC and RISC-V.

With the possibility of multiple ARM processors being deployed to space on multiple fronts, GMV has a keen interest in broadening its hypervisor to this architecture.

II. Background

Hypervisors are the key enabler of time and space partitioning (TSP) systems. They are relevant in all architectures that require complete isolation between applications and strong fault tolerance. Fundamental in the Integrated Modular Avionics (IMA) concept, their predominance in the aeronautical field has extended to the space domain. But first, the reason for the need of the hypervisor is presented, followed by a review on the branch of computing that enables several applications to run concurrently, with the illusion of complete hardware availability.

A. Virtualization

In a privileged/non-privileged architecture, all instructions are available to software running in privileged (supervisor) mode, whereas only a subset of the instructions are available to non-privileged (user) programs. The non-privileged programs in turn make system calls, typically implemented as supervisor call (SVC) instructions, to the privileged software, usually an OS kernel, that performs the privileged actions on their behalf, possibly checking if the calling non-privileged program has the rights to access the requested functionality. If a non-privileged program attempts to execute privileged instructions, an exception is raised and the program flow is disrupted. The execution jumps to a pre-registered table of exception handlers, normally controlled by the OS. The set of non-privileged instructions plus the system calls offered by the privileged software kernel present an extended view of the system to the user [7], as shown in Fig. 1.

Although the extended system is replicated for each user program, only one OS can be running at a time. This means that it is not only impossible to run more than one OS, but also denies the possibility of running any program that requires

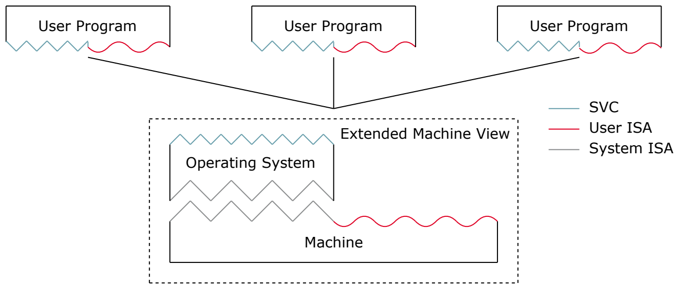

Fig. 1. Extended machine view exposed to non-privileged programs

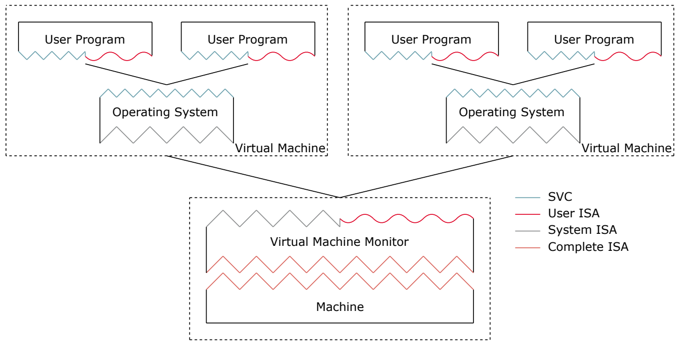

Fig. 2. Virtual machine abstraction to the operating systems

direct access to the privileged instructions. Furthermore, it occurs the inability to support older applications designed for older OSs, to modify and test other OSs and to run test and diagnostic programs that require direct access to privileged instructions, without taking down the machine and disconnecting all the users that were logged in [7].

The crucial innovation of virtualization is the introduction of a virtual machine monitor (VMM) that provides the illusion of multiple hardware infrastructures. Each virtual machine (VM) behaves as a replica of the original machine, including its instruction set architecture (ISA) and system resources, such as the central processing unit (CPU) state and memory and input/output (I/O) accesses. This is the first property of interest when analyzing a VMM, also known as equivalence. Equivalence can be stated as: any program executing in a machine without a VMM should behave identically when running in the same machine virtualized by the VMM. The only exceptions to this principle of equivalence are the CPU timings and total resource availability, which can not be maintained for the second case, since the VMM must split resources across all replicas. Each OS now operates on top of a VM instead of a real one, as illustrated in Fig. 2. The possibility of running more than one OS at once, along with the illusion of the entire hardware availability, offers a very robust multi-programming interface, as long as the VMM manages to run all virtual machines concurrently and keep complete control of the virtualized resources.

B. Critical instructions

For the VMM to work as expected in a privileged/non-privileged architecture, without the OS meddling in its execution, the entire VM must run in user mode, containing both the OS and user applications, letting the supervisor mode entirely as the VMM's playground. The VMM keeps additional information on the privileged level of the instructions carried out by the VMs, if their were executing natively on the machine without the VMM presence, so that it can be used later by the VMM to simulate the behaviour of those instruction depending on their original privilege level [8].

Regardless of being privileged or non-privileged, instructions can be separated into two different sets according to their dependency on the present state of the system. The first is the sensitive instructions set, which groups all instructions that either change or are dependent on the state of the system. Innocuous instructions comprise every other instruction that does not fall into the previous category and represents the second set. To respect the second property of a hypervisor, complete resource control, all sensitive instructions must be run by the hypervisor and any attempt from user level programs to run them must raise an exception to the hypervisor. This effectively prevents the programs running in the underlying VMs to access or modify resources and time shares not allocated to them.

The third and final property of a hypervisor is efficiency. To allow an efficient construction of a hypervisor, the grand majority of instructions need to run natively on the CPU without hypervisor interference. The instructions that are oblivious to the machine state and do not alter the system configurations, the available memory or the I/O resources, can run natively in hardware, while the other ones must either be executed by the hypervisor or with its consent, generally being simulated in supervisor mode taking into account their original privileged level previously saved by the hypervisor.

This second group of instructions that have consequences on the forthcoming state of the machine is the same group as the previously denominated sensitive instruction set. To safeguard that the sensitive instructions are executed by the hypervisor, they must be a subset of the privileged instructions [9]. Since the OS inside a VM is running in user mode, whenever it executes a privileged instruction, the execution is trapped into the hypervisor. If the instruction is sensitive, then the hypervisor simulates it in software, updating the respective VM CPU registers and memory boundaries saved in software structures, returning back to the OS after handling the exception. This last property of a hypervisor is the hardest to achieve, as not all architectures are virtualization friendly, having both sensitive and non-privileged instructions. The instructions in the intersection between the sensitive and non-privileged sets are often called critical instructions. If a sensitive instruction is not trapped into the hypervisor, then unpredictable behaviour will ensue, and the second property, complete resource control, will be void, with a high likelihood of leading the other VMs to experience faults.

These properties offer an insightful view on what is a virtualization-friendly architecture, where the last two properties entail an architecture without critical instructions.

There are two possible solutions to the critical instructions problem, full virtualization and paravirtualization, either with its advantages and disadvantages. These do not offer a virtualization as efficient as in the case of Popek and Goldberg's efficient hypervisor, where there are no critical instructions, but since there are plenty of typical privileged/non-privileged architectures that feature such instructions, alternatives had to be found to host a hypervisor and achieve correct virtualization.

The first is not available in the target board support package (BSP) since it requires hardware modification, so the second was used, with the added benefit of being the optimal virtualization techniques, despite the added maintenance cost. In this virtualization method, the guest OS is aware that it is running inside a VM and is modified (or created from scratch) to interact with the hypervisor [10], [11]. It accomplishes the communication with the hypervisor through hypercalls (HVCs), the equivalent of a SVC when user applications request the OS to perform an action on their behalf, but in this case the OS requests something from the hypervisor. This method eliminates the critical instructions by replacing them with HVCs and removes the overhead imposed in the hypervisor from discerning the current context of the sensitive instruction attempted in the VM, whenever an exception is raised. The guest OS now has access to an application programming interface (API) of HVCs offered by the hypervisor to help in the virtualization effort.

Replacing sensitive code segments with HVCs removes the otherwise occurring exceptions with near function behaviour, with the ability to pass arguments and receive values from the hypervisor. However, as extended kernel modifications are warranted and have to be maintained after updates, greater maintenance costs are associated with it. It also requires the OS availability for customization and since some OSs are proprietary they cannot be freely altered.

C. Hypervisor

As a consequence of the success of the IMA architecture in the aeronautical field, a similar approach has transitioned into the space domain. The hypervisor in an IMA architecture can be boiled down into two fundamental notions, TSP and fault detection, isolation and recovery (FDIR). By taking advantage of the existing regulations such as the ARINC 653 avionics standard, a framework for the onboard computer (OBC) OS can be designed using well established and tested principles [12]. Incorporating these two notions into a type-1 hypervisor creates the foundations for a space-graded hypervisor.

1) Time and Space Partitioning: As described in ARINC 653 (Avionics Application Software Standard Interface) - Part 1 - Required Services [13], the central philosophy of an IMA system is partitioning, where applications are segregated with respect to space (memory partitioning) and time (temporal partitioning). This is useful both for fault containment and for ease of verification, validation and certification. The base unit of partitioning is a partition. Partitions can range from a Real Time Operating System (RTOS) plus applications to a single bare-metal OS and a single-threaded program, each having their own data, context, permissions, etc. This notion of partition behaves very much alike that of a VM in the context of virtualization. Partitions are limited to using only the system calls defined in the published application executive (APEX). To circumvent the restricted operations permitted by the ARINC APEX, the standard allows an optional partition type, the system partition, that can use additional system calls to the hypervisor, for instance, to manage I/O device drivers. The system partition still needs to conform by the robust temporal and spatial partitioning. The standard also recognizes the concept of multiple partitions belonging to the same component, called modules.

The hypervisor is responsible for enforcing the partitioning, making sure that different partitions are completely contained from each other spatially and temporally. It is also responsible for handling any errors that may arise from the partitions without interfering with other partitions in the system, i.e., errors resulting from a partition are handled during that partition's time slice.

Temporal partitioning is attained by following a fixed and cyclic schedule, determined before deployment, which ensures deterministic behaviour. The schedule uses as the base unit of time a major time frame. These have a fixed duration, and are repeated periodically. The major time frame is composed of smaller frames equal in size, which are allocated for the partitions. At the end of each minor time frame, the hypervisor retakes control of the execution and determines whether the following minor time frame is alloted for the same partition as the previous. If it is, then is resumes the partition, and if not, it saves the previous application context and restores the following one, passing the control to the succeeding application. The period between each minor time frame also determines the precision of the wall clock offered by the hypervisor to the partitions. Each partition has a number of tasks or processes running during its execution, but each partition's processes are managed by that partition scheduler and are outside of the scope of the hypervisor. Major time frames are repeated until the board shuts down.

Spacial partitioning is guaranteed with predetermined areas of memory allocated for each partition at compilation time. The space partitioning can be enforced by whatever mechanism is present in the available hardware, such as a memory management unit (MMU) or a memory protection unit (MPU), or by virtualizing every single store/load instruction, albeit at great performance costs. During each partition's execution time, memory access outside of the assigned memory areas is prohibited. Communication with other partitions or I/O devices must be done by requesting ARINC 653 inter-partition services, such as sampling and queuing ports, provisioned by the hypervisor.

2) Fault detection, isolation and recovery: The second major concept of the IMA architecture is FDIR. ARINC 653 defines the existence of a Health Monitor (HM) for monitoring and reporting errors in the entire system, experienced either in hardware or raised by the partitions. The HM foresees a HM table at the hypervisor level with HM callbacks to respond to pre-defined faults in the system and optional HM tables at the partition level to handle partition level errors, one for each partition. The notion of FDIR entails three stages during the course of a fault.

Starting out with detection, faults may be detected either:

- in hardware, such as memory violations, privileged ex-

TABLE I
ARINC HM ERROR LEVELS

Level	Impact
process	one or more processes in a partition, entire partition in the worst case
partition	only one partition
module	every partition in the affected module

ecution violations, overflows, timer interrupts and other I/Os;
- in the hypervisor, such as configuration errors and missed deadlines;
- by the partition OS or similar software, such as wrong sensor readings throwed as errors.

The particular list of all errors is implementation specific, as well as where they are detected. Every possible error in the system must be identified prior to the operation of the device and assigned a specific id.

To help categorize all errors in the system and to determine the appropriate HM callbacks, ARINC 653 defines three levels where an error may occur and their inherent impact, shown in Table I.

It is important to state that errors in the hypervisor are not provisioned in the ARINC 653. However, these errors must still be consistently handled in a complete and recoverable manner, but are outside of the scope of the standard and are the responsibility of the system integrator.

In addition to the id of the occurring fault, the operational state of the system is also taken into consideration (such as module/partition/process initialization, module/partition/process execution, partition switching, etc.) for establishing the level of the fault and the correct HM Table. The operational state is set and kept up-to-date by the hypervisor between each exchange in state. The three parameters in combination will determine the appropriate fault handler to be launched.

After a fault is detected, it must be kept isolated from the rest of the system. The first stage in a fault isolation after being detected is the selection of the correct HM callback, which also determines the degree of isolation of the fault. This is performed by extracting the error id and operational stage of where the fault occurred. Using these two parameters, the correct error level can be derived and the appropriate HM Table is used along the error id to select the proper HM callback.

If the HM Table used is the partition level one, then the fault is handled during that partition time slice, ensuring isolation from the rest of the system. The entire HM callback is performed within that partition's context and will relinquish control when another partition is due. If the error affects an entire module and the Module HM table is the one used, then isolation is kept only within that module's partition, still respecting another module's time slices. All partitions within the faulty module are affected.

Process level faults are also handled during the erroneous partition time slice, but are implementation dependent. These are normally handled by a higher priority task within the partition OS. If a fault is experienced during the process error handler, it becomes a partition level fault and is handled

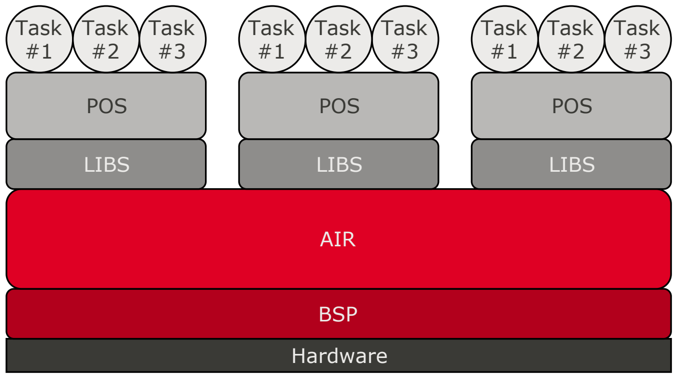

Fig. 3. AIR architecture

appropriately taken into account that it already faulted once before.

Following the fault detection and isolation comes recovery. Each entry of the chosen HM Table is accessed through the error id and the operation state and contains a HM Callback to recover from the pre-defined fault. There are three staple actions that can be taken for all levels of errors, stop, restart and ignore. Additional actions can be implemented through additional functions to deal with every possible error in the system.

III. TECHNOLOGIES

A. AIR

On top of the hardware is the BSP for each board. While this section is also developed in-house, it is intended to be entirely independent from the hypervisor logic. However, there is still some intertwine between the BSP and the hypervisor, being the complete separation one of the objectives of this work. On top of the BSP is AIR, the hypervisor, where all the logic related to temporal and spatial separation is performed, along with the initialization of the system, scheduling, context-switching and housekeeping. AIR is also referred to as the Partition Management Kernel (PMK), and the terms are used interchangeably. On top of AIR are all the partitions that are scheduled to run during the system's runtime. The libraries selected in each partition, including LIBAIR, are built separately for each partition, since there is no shared space between the partitions. The Partition Operating System (POS) is then left to its own execution, with the ability to schedule tasks, setup filesystems, etc. The final output of the build process is structured as in Fig. 3.

Two files are involved in the initialization of AIR in any board, the `start.S` and the `init.c`. The `start.S` is an assembly file required in all bare-bones software and is architecture-dependent and software-independent. As such it can be seen as part of the BSP box in Fig. 3. It is responsible for:

- allocating space for the board exception vector, either by filling the space with dummy function handlers or jumps to the actual exception handlers;
- clearing the **.bss**;
- setting up the stack pointers;

- invalidate all cache entries, the translation lookaside buffer (TLB) and other speculative mechanisms;
- configuring the floating-point unit (FPU), if enabled.

After the previous procedures the `start.S` calls the `pmk_init()` function, located in the `init.c`. From here onwards, the remainder of the system initialization is carried from AIR, with the appropriate calls to the BSP where required.

As a TSP hypervisor, AIR is responsible not only for maintaining the VMs, but also for enforcing temporal and spatial isolation. After the initialization is complete, AIR launches a partition responsible for maintaining the core in idle and enables preemption.

The temporal correctness is maintained by reserving a timer for only supervisor access, triggering an interrupt when it reaches the desired counter value, and auto-restarting, thus generating interrupts at regular intervals. The timer interrupt is raised at the beginning of each minor frame and the associated handler is dispatched. This handler is responsible for saving the previous partition's context, checking what partition is allocated for the current minor frame, and restoring the succeeding application's context. The overhead introduced by AIR has been evaluated to $1 \sim 2\%$ in single core while running on the ESA Next Generation Microprocessor (NGMP) [1], and grows inversely proportional to the duration of the minor time frame, deteriorating in performance for periods of under $0.001\,\mathrm{s}$ [14]. There are yet no extensive studies as to the overhead present in a multicore scenario.

The spatial isolation is preserved using the hardware structures of the underlying hardware, therefore it is architecture dependent. So far, AIR has only been developed expecting a MMU, but it keeps the separation between hypervisor and BSP by calling generic functions that can be adapted for every architecture.

In order to respect the ARINC 653 standard, AIR also realizes the concept of a HM. It follows the same approach as detailed in section II-C2, and the actual implementation can be visualized in Fig. 4. After a fault is experienced, an exception is raised and the execution jumps to the HM handler. It starts by performing a lookup using the error id and operational state in the system HM table to determine the level of the fault. After it has determined the level of the error, it performs a search in the correspondent HM table using the error id, finally performing the pre-determined action for the detected fault. The actions allowed in a module level error are SHUTDOWN, RESTART and IGNORE, where IGNORE later calls a function handler specified for the experienced error. The partition level table also offers the SHUTDOWN and IGNORE actions, with similar behaviour to the module HM table, but distinguishes between a COLD_START and a WARM_START. Both the COLD_START and WARM_START can be seen as a complete reset of the partition, and the equivalent to the module RESTART, but the information on which of the two occurred is passed on to the partition, that can later act differently based on that information.

[1]http://microelectronics.esa.int/gr740/index.html

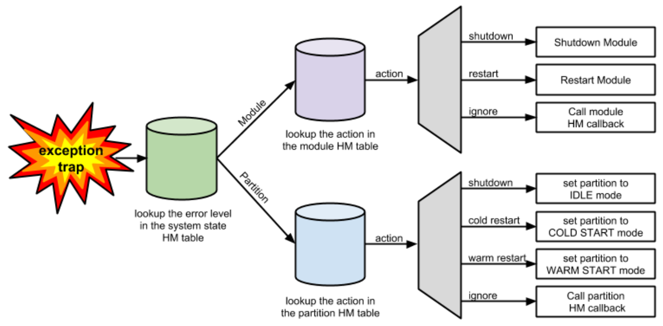
exception trap
lookup the error level in the system state HM table
Module
Partition
lookup the action in the module HM table
lookup the action in the partition HM table
action
action
shutdown
restart
ignore
Shutdown Module
Restart Module
Call module HM callback
shutdown
cold restart
warm restart
ignore
set partition to IDLE mode
set partition to COLD START mode
set partition to WARM START mode
Call partition HM callback

Fig. 4. AIR HM procedure

The HM configuration is performed through an extensive table present in a configuration `.xml` that is comprised of the system HM table plus a module HM table and a partition HM table for each partition. The specific function handlers related to the IGNORE action are placed inside the partition files during each partition's compilation. The error handler is then responsible for retrieving the operational state of where the error occurred and the error id through a system call.

IV. Architectural differences

The architectural differences between the two architectures were distilled from both architectures reference manuals, presented in Table II.

TABLE II
Architectures' Reference Manuals

Documents procured	Ref.
SPARC The SPARC Architecture Manual Version 8	[15]
GR740 Data Sheet and User's Manual	[16]
ARM ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition	[17]
ARM® Compiler armasm User Guide	[18]
ARM Architecture Reference Manual Thumb-2 Supplement	[19]
Cortex™-A9 Technical Reference Manual	[20]
Cortex™-A9 MPCore® Technical Reference Manual	[21]
ARM® Generic Interrupt Controller Architecture version 2.0 Architecture Specification	[22]
Zynq-7000 SoC Technical Reference Manual	[23]

A. Instruction set architecture

The two processor architectures are designed as reduced instruction set computer (RISC), making their ISA similar in nature. A RISC instruction set is typically defined by the following traits:

- each instruction undertakes a reduced amount of work;
- uniform instruction format and length, leading to simplified processor logic;
- all general-purpose registers can be used either as source or destination in all instructions;
- data-access is performed in a limited number of instructions, with that being their entire objective.

These ISA attributes result in a similar instruction set with an approximately equal number of instructions between SPARC and ARM. Nevertheless, while a big set of both architecture's instructions share much of the same purpose, their architectural dissimilarities translate into additional instructions on both sides. Table III presents the encountered instructions that either perform a behaviour not found in the other architecture or that realize the same goal differently.

TABLE III
ARCHITECTURE SPECIFIC INSTRUCTIONS

Function	SPARC	ARM
window management	SAVE, RESTORE	-
memory barriers	FLUSH, STBAR, NOP	ISB, DSB, DMB
change instruction set	-	BX, BLX
change operating mode	-	CPS
mutual exclusion	SWAP	LDREX, STREX

B. Stack

The first of the two necessary initializations before jumping into a C function is the stack initialization. Compiled C code will assume that the stack pointer (SP) is correctly initialized and make extensive use of it when jumping between functions, to pass and use arguments.

Another very important use of the stack is during exceptions. Upon entering an exception, the previously executing application context is dumped into stack memory. This stack zone is called the interrupt stack frame and holds the necessary information to resume the application when returning from the exception. The GNU SPARC compiler always reserves the necessary space on the stack to be used on the event of an exception, and this occurs for every register window.

ARM on the other hand, since exceptions always jump to a operating mode with its own stack, does not need to reserve this additional space on every procedure entry, instead it can save the interrupt stack frame on each of the operating modes' stack.

C. .bss

The second of the two necessary initializations is zeroing the **.bss** section. In the C standard statically-allocated variables without an explicit initializer are expected to hold that value by the programmer, but when starting up a computer no assumptions can be made about the initial state of the memory.

Since this step could possibly take a long time depending on the amount of statically-allocated uninitialized variables, it is important to consider the size of the data bus and take advantage of it. Both the GR740 and the Arty Z7 use an AHB/AXI data bus of 64 bit, so for the best performance when writing to memory, two registers are stored at a time. Both SPARC and ARM offer instructions to take advantage of this characteristic of most modern boards. SPARC offers the STD/LDD pair to store/load doubles (64 bit variables) to/from memory. ARM boosts a more powerful version of the previous instructions, the STM/LDM pair, that can store/load multiple registers to/from memory. To take advantage of the data bus size, ARM's version needs be used with a pair number of registers. This principle holds true not only in the **.bss** zeroing, but everywhere else in the code, with compiled code taking advantage of both pairs of instructions.

D. Cache, branch predictors and translation lookaside buffer

After both the stack pointers are initialized and the **.bss** is cleared, any remaining initialization procedures can be done in C.

For the same reason the **.bss** needs to be cleared, so do the L1 instruction and data caches, branch predictors and TLB need to be invalidated. As defined in the ARMv7 reference manual, both the caches, branch predictors and the TLB are disabled at reset. There is not, however, information on the starting content of the caches at reset, so they are considered to not be empty, so they must be cleaned at start-up. All these hardware structures must have their contents invalidated at start-up, and only after can they be enabled.

SPARC differs has there is no constraint for the caches and the TLB to start disabled. The branch predictor is not programmable in the LEON4 processor. While the SPARC architecture manual does not specify the initial state of the cache, the GR740 one does, stating that both caches are disabled at reset. Nevertheless, to ensure compatibility with other BSPs, both the caches and the MMU are disabled during start-up, followed with the cache and the TLB being invalidated, and only after re-enabled. The L2 caches are dependent of the board used. The GR740 L2 cache is disabled after reset and it is invalidated at the same time it is enabled. The Zynq-7000 SoCs L2 cache is cleared upon reset, but the entries must still be invalidated before enabling it.

E. Register window

The immense disparity between SPARC's and ARM's register layouts only comes into play when changing context. Removing SPARC's register windows, shown in Fig. 5, only eases the hypervisor development when directly dealing with the registers, as the maintenance code on the windows is removed. The interrupt stack frame is also reduced since it only has to keep one set of general-purpose registers, plus the FIQ's registers 8 through 12 (if used) and the LR, SP and saved processor state register (SPSR) present in every mode, with the exception of the shared LR and SP, and inexistence SPSR, between User and System, up to a total of 29. This is in contrast with SPARC's n windows, 8 in the case of the GR740, with 16 registers each, plus 8 globals, totaling 136. In both cases, the total number of registers is multiplied by the level of nesting permitted, per partition context.

While in theory SPARC's register window improves the performance when changing context during normal operation (no hypervisor), by reducing the number of accesses to the stack when entering/exiting function calls, it introduces a penalty when virtualization comes into the picture. Besides the higher space usage, when changing from one partition to another, all used register windows need to be dumped into memory. Although AIR increases performance by only restoring the last used register window and only restoring additional windows when the partition requires them, the worst-case execution time (WCET) still comes into play, since all windows could be requested. ARM's plain register design increases the predictability of context switching.

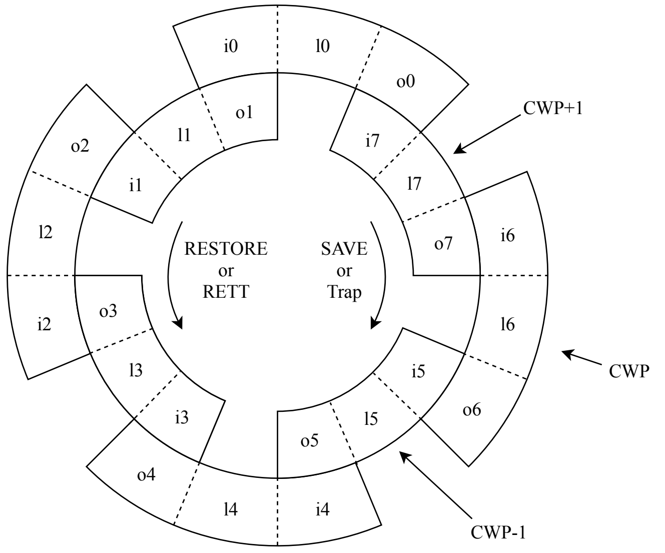

Fig. 5. SPARC general-purpose register windows

TABLE IV
DIFFERENCE BETWEEN ARCHITECTURES' SYSTEM CONFIGURATION

Function	SPARC	ARM
window management	CWP + WIM	-
exception table base address	%TBR	cp15
system configuration	RDASR/WRASR %ASR17	cp15
cache control	ASI 0x2	cp15
cache operations	ASI	cp15
MMU control	ASI 0x19	cp15
MMU operations	ASI	cp15

F. System configuration and identification

The two architectures access information on the system differently, with ARM providing a slightly simpler model, accessing most information through a single coprocessor using the MRC/MCR instruction pair. SPARC varies between ancillary registers and Address Space Identifier (ASI) accesses.

The accesses to the system configuration are shown in Table IV.

G. Exceptions, Traps and Interrupts

The vocabulary used by each architecture manuals differs when exceptions, or traps in the case of SPARC, are referred to. In essence, they specify the same and can be used interchangeably. An exception or trap is an anomalous event that changes the normal flow of a program and jumps the execution to a table of predefined handlers. If the handlers do not exist, unexpected behaviour will ensue.

Synchronous exceptions are the result of instructions that directly trigger an event on the execution stream. Within the bounds of this category fall SVCs and undefined instructions, either by attempted execution of an instruction not available at that privileged level or an unimplemented instruction, such as a floating-point operation with the FPU disabled. Precise asynchronous exceptions normally encompass interrupt requests (IRQs). The state previous to the interrupt is known, but won't give any insight as to the interrupt itself. Imprecise asynchronous exceptions are normally the result of memory errors. Due to the optimizations present in most present-day CPUs, such as caches and write buffers, errors when writing/reading to/from memory are seen with some delay and the exact instruction that generated the abort cannot be derived from the state of the computer when the exception is taken. A common example is a read-only memory area marked as cacheable and with a write-back policy. Writes to the cache won't trigger any error and the exception will only appear when the cache line gets evicted and written back to memory.

All the aforementioned scenarios point directly to an operating mode in ARM, making the distinction between each exception type faster but grouping several distinct errors into one function handler. SPARC on the other hand holds a 256 entries table, with 80 entries reserved for a combination of undefined instructions, data aborts, the *window_underflow* and *window_overflow*, and the *reset* exception, 16 entries for IRQs and 128 for user determined SVCs. Not all of these exceptions need to be implemented in a BSP, but they have to respect the order defined in the architectural standard. While failed instruction fetches and data errors can cause imprecise asynchronous exceptions, ARM offers some insight through the Instruction Fault Status Register and the Data Fault Status Register, accessible through coprocessor 15. IRQs in ARM are distinguished later in the interrupt handler through the Interrupt Acknowledge Register (ICCIAR) available in the interrupt controller. Finally, SVCs are differentiated by a value passed in the instruction itself, accessed in the exception handler through the preferred return address. In T32, the maximum number of SVCs is restricted to 256, due to the $8\,\mathrm{bit}$ encoding (imm8), but the range goes up to 2^24 in A32 due to the $24\,\mathrm{bit}$ encoding of the SVC id.

The major difference going from one architecture to the other is the use of two tables in ARM for handling any exception in contrast with the one in SPARC. While the method employed by ARM introduces an additional level of redirection, it reduces the initial amount of exception handlers, possibly negating duplication of the same code. It also reduces the size of the exception handler table, which is mandatory for the two architectures, going from 256 entries in SPARC to only 8 in ARM.

H. Memory Management Unit

Both SPARC's and ARM's MMU's are organized largely in the same manner, with the exception of SPARC's reference MMU proposing three translation levels and ARM's using only two. The respective Page Table Entries are referred in each of the architecture's manuals, and must be followed for the TLB to work as expected. However, not much differs in their implementations and the code developed for ARM is mostly a repetition of the one for SPARC, adjusting only the Page Table Entries format.

V. BSP HYPERVISOR-RELATED PROCEDURES

The primary role of the hypervisor is to maintain several VMs running concurrently. To accommodate this, it requires two features:

```
void core_context_init(
   core_context_t *context, air_u32_t id);
void core_context_setup_idle(core_context_t *context);
void core_context_setup_partition(
   core_context_t *context, pmk_partition_t *partition);
```

Fig. 6. BSP functions to initialize and setup the partitions' context

1) memory earmarked for each VM context;
2) a mechanism to change context.

The first is allocated for each core and for each partition during initialization. The space assigned for each core contains the context of an idle VM, and is the one used when first booting up the core and on every vacant time slice. The one for each partition is loaded with the initial parameters specific to that particular partition. The PMK can use three BSP calls to setup these contexts, which declarations can be seen in Fig. 6.

Both contexts are composed of:

- CPU id and registers
  - on SPARC: processor state register (PSR), Window Invalid Mask (WIM), Trap Base Register (TBR), cache control and MMU control;
  - on ARM: PSR, VBAR and MMU control.
- Interrupt stack frame pointer.
- Interrupt nesting level.
- Interrupt controller registers.
- FPU context if compiled for it.
- AIR current state information.
- HM event currently being serviced (if one exits).

The second feature of the hypervisor is a mechanism to change context. The behaviour in SPARC was copied to ARM. A single timer is used exclusively for the scheduler interrupt, and is the only one set to the highest priority. Both architectures allow nested interrupts, so the highest priority interrupt, if only one, will always be taken. Code segments with disabled interrupts are only used in first context save upon entering an exception and during the timer handler. Never during partitions. Any request by the partition to disable interrupts is only simulated in the virtualized CPU registers, never actually taking place.

A. *AIR Paravirtualization*

AIR follows the paravirtualization approach to achieve correct virtualization. This approach requires guest OS modifications, but more importantly for the hypervisor, it must offer a direct route of communication between the VMs and AIR. This route is maintained by HVCs to AIR, using the SVCs available in both architectures. The HVCs available to the guest OSs can be separated into two main groups:

1) These HVCs are used to implement hardware virtualization. The group is composed of:
   - disable/enable interrupts;
   - disable/enable exceptions;
   - disable/enable the FPU;
   - get/set the CPU registers;
   - get/set interrupt mask;
   - return from an HM event.
2) These HVCs are used to either retrieve information from AIR or to perform more complex procedures using AIR's abstractions. The group is composed of:
   - get physical address;
   - get core id;
   - get µs per tick and get elapsed ticks (these two HVCs can be used to implement timers within the partitions);
   - get partition information;
   - get schedule information;
   - get ports information;
   - get HM event;
   - print;
   - boot secondary core.

Only the first group requires direct migration from one architecture to the other. The second group of calls interact directly with the PMK, although some will later require interaction with the BSP, e.g. when booting secondary core. The "return from an HM event" belongs in the first group because its purpose is to jump to the pre-HM program counter (PC), which is accessible directly from the BSP.

In this scenario, ARM achieves the lowest disabled interrupts downtime. Since HVCs can be interpreted as normal function calls, the handler takes advantages of a similar behaviour of the stack usage of normal function entries/returns. After the execution enters the Supervisor mode, the preferred return address and SPSR are saved as per usual. Using ARM's unique instruction SRS (Store Return State), the Supervisor mode can save the preferred return address and the SPSR into any stack available. By using the System stack, which is shared with the User mode, it attains nearly identical behaviour to that of a function. Furthermore, is can now push onto the System stack the interrupt stack frame before the exception occurred, since the previous instruction does not use any of the general-purpose registers. At this point, preemption can be enabled again, and the exact procedure can happen again, even though it is extremely unlikely, since HVCs do not use HVCs again. The timer interrupt can however happen, without loss of information, as long as the stack is kept coherent. Since the execution will not return to User mode before all exceptions are handled, there is no possible way for the stack to become corrupted. After the HVC is completed, the preferred return address and SPSR are loaded onto the PC and the current processor state register (CPSR) through the RFE (Return From Exception) instruction.

Using the System stack is important in this behaviour, because it keeps thing clean, all the while saving in space and time. If the Supervisor stack were used, it would also need to be saved in memory during a context change. This way, the System stack is the only one used across the entire partition time slice.

B. *Interrupt virtualization*

Also required to complete the virtualization process, is the virtualization of the interrupts destined to the VMs. For the hypervisor to maintain control over the system, it must catch all exceptions, even if these are bound to the VMs. The code developed is similar in both architectures. First an interrupt is identified about its type by the hypervisor. Afterwards, it can be checked if it belongs to any of the scheduled partitions. If it does, the hypervisor recreates the same process of identifying the interrupt by modifying the appropriate virtualized registers and when returning to the partition, it resumes execution from the guest OS' exception table. The paravirtualized guest OS can then redo the same procedures used by AIR, but in its case, by accessing the virtualized information.

The small discrepancy is due to the different exception models. As identified in the previous chapter, ARM employs an additional redirection level. As such, the virtualized exception checks are present across all exception handlers, where in SPARC all exceptions have the same entry point, and this code only appears once.

C. *Time and space partitioning*

After the hypervisor achieves correct and efficient virtualization, the TSP aspect simply boils down to the schedule and communication protocols used. And the TSP standard is rather concise on it. The schedule is fixed and cyclic, composed of minor and major time frames that repeat themselves. There is no shared memory between VMs. Communication must be done through queuing and sampling ports supplied by the hypervisor and the contents must be written from one VM to the hypervisor and read from the hypervisor to the receiving VM. Nothing in this implies architectural changes, since its application is done entirely in the PMK. The only BSP calls are the writes to and from the hypervisor, that check if the partitions have the correct permissions to access that memory and do a *memcpy*. There are no differences between architectures in this regard.

D. *Health Monitor*

The architectural differences in the HM implementation are derived from the different exception models. The HM is responsible for all the errors that occur in the system, and is designed to be generic enough to accommodate different BSPs. The job of the BSP in this design is to determine the error id of the fault. For this objective, SPARC installs the same HM handler in all relevant exceptions and defines the error via a lookup in a 256 entry vector based on the exception table entry.

In contrast with SPARC, ARM's exception table is very reduced in size, having only three entries for HM errors (undefined, prefetch and data aborts), and the error id is found through if-else statements. However, for more detailed error ids, the instructions that generated the errors must be decoded according the ARM's instruction set encoding. To implement lazy FPU switching, where the FPU is only re-enabled after a context switch if it is used, the undefined exception faulted instruction is decoded to determine its identity.

While ARM benefits in size from its simplified HM design due to its exception model, when entering into detail in the determination of the error id, it is clear that SPARC has the advantage. Even with the exception table entries locked in number to take into consideration future versions of the architecture, reverse-engineering every instruction in ARM becomes a very time consuming job.

TABLE V
TIMER TESTS

Nr P.	Ticks/Second	Schedule
2	10	P1 0.5 s - idle 0.5 s - P2 0.5 s - idle 0.5 s
2	100	P1 0.5 s - idle 0.5 s - P2 0.5 s - idle 0.5 s
2	1000	P1 0.5 s - idle 0.5 s - P2 0.5 s - idle 0.5 s
2	10000	P1 0.5 s - idle 0.5 s - P2 0.5 s - idle 0.5 s

TABLE VI
HEALTH MONITOR TESTS

Nr P.	State	Schedule
1	Module initialization	P1 1 s
1	Partition Execution	P1 1 s
2	Partition Execution	P1 0.5 s - P2 0.5 s
2	HM execution	P1 0.5 s - P2 0.5 s
1	Partition Execution due to FPU error	P1 1 s

VI. TESTS

To reliably test a hypervisor that employs paravirtualization, a paravirtualized OS is required. It was created a barebones OS, whose sole function is to test the functionalities of the TSP hypervisor. This OS can run any executable compiled in T32 or A32, as long as it does not have any external dependencies apart from the GCC libraries.

The OS only runs in single-core and makes extensive use of the LIBAIR API. The tests were run both on the QEMU emulator and on the Arty Z7, yielding similar results. Nr P. is the number of partitions.

A. *Timer test*

The timer test aims at analysing the behaviour of the timer HVCs, the *air_syscall_get_us_per_tick* and the *air_syscall_get_elapsed_ticks*. The combination of these two calls provides a timer for the partition, updated at every minor timer frame.

This test performs as expected, with higher resolutions attained for higher ticks per second, but there was a noticeable delay in the 10000 Ticks/Second test in comparison with the previous ones. The actual delay needs to be further studied with timing analysis tools.

B. *Health monitor test*

The health monitor tests are performed to gauge the response of the HM to different kinds of errors in different states of the program.

The tests performed as expected, always launching the correct actions from the HM configuration table. The two partitions tests did not affect each other negatively, and the errors were successfully contained within the partitions. The FPU lazy-switch test also performed as expected, with the partition recovering after an undefined error caused by a FPU instruction with the FPU disabled, by enabling the FPU and repeating the instruction.

VII. CONCLUSION

In conclusion, the new BSP for AIR performed innocuously, yielding the same results as the SPARC one. It maintained the same BSP API calls and performed equivalent procedures when compared side-by-side. The architectural differences were mostly felt in virtualizing aspect of the hypervisor, due to incompatible register and exception models. However, a similarities were found and the new BSP achieved the same result through different methods. It improved on the HVC handling, by providing a lower timer frame where the interrupts are disabled, thus increase the accuracy of the global timer. The objective of creating a BSP for a TSP hypervisor was successful and is currently being used in production and in proposals for future projects.

In addition to the created BSP, a comparative study between SPARC and ARM was also performed and can be used as reference material when migrating from SPARCv8 to ARMv7.

Parallel to the development of the new BSP was also the development of the toolchain used to compile AIR, which was reinforced with new templates and script to streamline the development process.

A. Future work

The most immediate development to be made is the expansion of the BSP to multicore, accompanied by the a multiprocessing barebones OS. This would not only bring the most immediate performance boost, but also adding another dimension to the scheduling possibilities.

Since this code is expected to run in space, validation and verification facilities should also be added to speed-up the process of qualification.

With the attention given to the modularity of AIR, the addition of new architectures and more interesting BSPs could also bring more depth into AIR. This case is specially true for the addition of the ARMv8, that incorporates the Virtualization Extensions into a more consolidated architecture.

References

[1] *Technologies for European non-dependence and competitiveness*, COMPET-1-2016, Funding & tender opportunities, European Commission, Nov. 2015. [Online]. Available: https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/opportunities/topic-details/compet-1-2016.

[2] *DAHLIA Deep sub-micron microprocessor for spAce rad-Had appLIcation Asic*, DAHLIA Consortium, 2017. [Online]. Available: https://dahlia-h2020.eu/.

[3] J. L. Poupat, T. Helfers, P. Basset, A. G. Llovera, M. Mattavelli, C. Papadas, and O. Lepape, *DAHLIA, Very High Performance Microprocessor for Space Applications*, Proc. DASIA 2018 - DAta Systems In Aerospace, to be published, Oxford, May 2018.

[4] Portugal Space, 2019. [Online]. Available: https://www.ptspace.pt/.

[5] *Atlantic International Satellite Launch Programee: Launch services to Space from the Island of Santa Maria, Azores*, International Call for Interest, ESA, Luísa Ferreira and CEiiA, Sep. 2018. [Online]. Available: https://www.portugal.gov.pt/download-ficheiros/ficheiro.aspx?v=28d4b86b-9b43-4005-a968-b4c4730f28a7.

[6] Infante, 2016. [Online]. Available: http://infante.space/.

[7] R. P. Goldberg, "Survey of Virtual Machine Research," *Computer*, vol. 7, no. 6, pp. 34–45, Jun. 1974.

[8] J. P. Buzen and U. O. Gagliardi, "The evolution of virtual machine architecture," in *AFIPS Conf. Proc. National Computer Conference and Exposition*, New York, NY, 1973.

[9] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[10] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," University of Washington, Seattle, WA, Tech. Rep. 02-02-01, 2002.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *SOSP'03 Proc. 19th ACM Symposium on Operating Systems Principles*, vol. 37, Bolton Landing, NY, 2003, pp. 164–177.

[12] N. Diniz and J. Rufino, "ARINC 653 in Space," in *Proc. DASIA 2015 - DAta Systems In Aerospace*, vol. 602, Edinburgh, 2005.

[13] *Avionics Application Software Standard Interface Part 1 - Required Services*, Specification 653-2, ARINC, Dec. 2005.

[14] C. Silva and C. Tatibana, "MultIMA - Multi-Core in Integrated Modular Avionics," in *Proc. DASIA 2014 - DAta Systems In Aerospace*, vol. 725, Warsaw, 2014.

[15] *The SPARC Architecture Manual Version 8*, SPARC International, Inc., Campbell, CA, 1991.

[16] *GR740 Data Sheet and User's Manual*, Cobham, Jul. 2018.

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

[18] *ARM® Compiler armasm User Guide*, ARM Limited, Oct. 2018.

[19] *ARM Architecture Reference Manual Thumb-2 Supplement*, ARM Limited, Dec. 2005.

[20] *Cortex™-A9 Technical Reference Manual*, ARM Limited, Jun. 2012.

[21] *Cortex™-A9 MPCore® Technical Reference Manual*, ARM Limited, Jun. 2012.

[22] *ARM® Generic Interrupt Controller Architecture version 2.0 Architecture Specification*, ARM Limited, Jul. 2013.

[23] *Zynq-7000 SoC Technical Reference Manual*, Xilinx, Jan. 2018.