POLITECNICO DI TORINO

Master degree
in
Computer Science - Embedded systems

VOSySmonitoRV: a mixed-criticality solution on 64bit Linux-capable RISC-V platform

Supervisor
Luciano Lavagno
Associate Professor
Company tutor
Michele Paolino
Virtual Open Systems SAS

Candidate
Flavia Caforio
Matr. 257750

A.A. 2020/2021

Abstract

This thesis has been developed during a six months internship in Virtual Open Systems SAS company located in Grenoble, France.
The company is specialized in design and implementation of high-performance mixed-critical virtualization solutions on low-power multi-core & heterogeneous platforms for automotive, IoT edge, cloud computing solutions.
Embedded systems are widely used in many fields nowadays, often in mixed-criticality environments i.e., systems that need a real-time component with a certain time and safety constraints alongside a rich operating system. VOSySmonitor is the Virtual Open Systems SAS (company) proprietary solution for mixed-criticality embedded systems on Arm architecture based on Arm TrustZone. The thesis work consists in the participation in the evaluation and extension of this solution on a 64bit RISC-V Linux-capable platform, VOSySmonitoRV, which remains company proprietary. RISC-V is an innovative and open instruction-set architecture that was originally designed at Berkeley to support education and research. The importance of RISC-V ISA is in its open-source licence and open standard, thanks to its frozen ISA everyone can invest in writing software that will run forever on RISC-V-like processors. RISC-V is extensible, the privileged architecture, approved and also frozen, allows many possible software stacks that define different execution environments thanks to the privileged levels.
VOSySmonitoRV has the advantages of virtualization to allow the secure co-execution of two, or more, OSes in an isolated manner, but it does not exploit the virtualization support for security reasons.
VOSySmonitoRV executes in M-mode, the higher privileged level. The first result is a prototype, starting from a company one, that is done on the SiFive HiFive Unleashed platform. The co-execution of the two OSes is on separate

harts (cores), allocating three harts for Linux and one for the real-time OS. In this way there is a really strong isolation thanks to the PMP unit that gives memory and peripheral isolation. However, RTOS workload can be characterized for a long time by scheduled idle tasks. In order to efficiently use the RTOS hart and to give Linux almost native performance, the most challenging feature of VOSySmonitoRV is the co-execution of a safety-critical OS with a non-critical OS on a single hart.
The feasibility of this feature depends on the latency of the context switch between OSes because it must be under a reasonable threshold otherwise both operating systems would have unacceptable performance losses, especially for the real-time OS. Furthermore the application of VOSySmonitoRV in mixed-criticality systems is evaluated. The evaluation is done through a custom benchmark that measures the interrupt latency and the context switch overhead of a simple ECALL. Results are very promising for the use of VOSySmonitoRV in mixed criticality systems and for the realization of the shared core, ensuring that it is a further optimization rather than a loss. A second prototype with the shared core is developed but it is not fully operational yet.

Acknowledgement

Vorrei innanzitutto ringraziare il mio relatore Luciano Lavagno per la sua disponibilità e per avermi dato la possibilità di fare questa esperienza di tesi all'estero.
Inoltre ringrazio Virtual Open Systems SAS e il mio supervisor Michele Paolino per avermi supportato e permesso di lavorare praticamente nell'ambito dei sistemi embedded, per la preziosa esperienza e per l'ambiente piacevole. A tal proposito ringrazio in particolare Pierpaolo, Stefano e Michele.

Ringrazio la mia famiglia, i miei genitori e i miei fratelli per avermi sostenuto sempre, anche negli studi.
Grazie alla mia amica Giulia che mi sopporta da sempre in questo mondo di stelline e arcobaleni.
Grazie Rossana per le giornate di studio intenso, mi hanno motivato molto.
Ringrazio i miei carissimi amici Rossella, Vincenzo, Gianmarco, Tino, Andrea e Sara con i quali ho iniziato la mia carriera universitaria in collegio e di cui ora non potrei fare a meno.
Grazie ai miei compagni e amici, in particolare Silvia, Chiara, Alessia, Pseudo, Samuele, Pierfrancesco e Giuseppe per i quali gli anni a Torino sono stati più leggeri e indimenticabili.
Infine, ma non per importanza, grazie a Odilia che mi sprona sempre con la sua determinazione e ambizione.

Un ringraziamento speciale va alla mia nonna Palma, che sarebbe stata tanto fiera.

Contents

1 Introduction **1**
1.1 Premise and work introduction 1
1.2 Thesis outline 2

2 Background **4**
2.1 RISC-V architecture 4
2.2 SiFive HiFive Unleashed 9
2.3 Linux device drivers 12
2.4 Linux Device Tree 13
2.5 Real-time operating system 15
2.6 Linker script 15
2.7 Related work 15

3 VOSySmonitoRV **18**
3.1 VOSySmonitoRV design specification 20
3.2 Starting prototype 20
3.3 First prototype 22
3.4 Second prototype 27

4 Custom Performance Benchmark **33**
4.1 Environment and implementation details 35
4.2 Results 42

5 Conclusions **46**
5.1 Future work 47

A VOSySmonitoRV benchmark application code **48**

B VOSySmonitoRV benchmark driver code 51

C VOSySmonitoRV benchmark extension code 54

D Benchmark results elaboration 57

List of Figures

2.1 Trusted execution environments overview [21] 5
2.2 Machine mode MSTATUS CSR register [21] 5
2.3 PMP configuration CSR register [21] 6
2.4 PMP address CSR register [21] 6
2.5 Flow of the PLIC external interrupts handling [21] 8
2.6 SiFive HiFive Unleashed platform 10
2.7 Interrupts overview [17] 11
2.8 FreeRTOS execution overview 14

3.1 VOSYSmonitoRV Architecture Overview 19
3.2 Memory overview in the first prototype of VOSySmonitoRV . 24
3.3 `pmpcfg0` register, 8 PMP regions possible [21] 24
3.4 Stress test on Linux harts 28
3.5 A triggered kernel panic in Linux 28
3.6 Context switch from the RTOS to the General Purpose OS when is required by the RTOS 29
3.7 Context switch from the GPOS to the RTOS when is required 30
3.8 Memory overview in the second prototype of VOSySmonitoRV 31

4.1 Benchmark program flow diagram 34
4.2 Interrupt latency measurements, μ is the average value and σ the standard deviation 43
4.3 Context switch measurements, μ is the average value and σ the standard deviation 43

List of Tables

4.1 The comparison of the benchmark results on the duration of the context switch with a x86 processor. 44
4.2 The comparison of the latency of the context switch between operating systems with different criticality 45

Chapter 1

Introduction

1.1 Premise and work introduction

In the new era of Industry 4.0 [20], the importance of solutions to allow the co-execution of activities with different levels of criticality in common hardware platforms has become crucial. It is now a sought-after that there is the need to have at the same time, in recent embedded systems, features typical of generic operating systems and a component of real-time constraints dealing with safety issues. Usually, this feature is possible thanks to virtualization techniques through the virtualization support on the machine. Virtualization is the ability to abstract a server into a virtual machine [14] and it needs a hypervisor, a software monitor below the virtual machine and above the hardware, to manages resources. The idea is to be independent of the hardware implementation in order to have several operating systems on the same machine, to have more efficient use of resources while keeping them functionally isolated.
On the other hand, it is well known that the complexity of a hypervisor solution increases security problems and vulnerabilities have already been found [16]. The malicious use of virtualization technologies can consist of the capability of using virtualization support to execute malicious code at a higher privileged level to control the lower privileged ones.
For these reasons Virtual Open System as a company proposed VOSyS-monitor product, running in Secure Monitor mode on ARM architecture for mixed-criticality systems, compliant with the ISO 26262 standard. This so-

lution is based on ARM TrustZone technology to allow and enforce isolation of a co-execution of multiple operating systems creating two separate worlds without virtualization extension: the secure world and the non-secure world. This idea is very valid in embedded contexts where we need a rich operating system along with a real-time operating system whose safety constraints must be respected. VOSySmonitor is also capable in multicore systems of sharing the cores of the real-time operating system (RTOS) with the general-purpose operating system (GPOS) during periods of only idle tasks scheduled to efficient use of resources.
In recent years RISC-V is actually gaining ground as a new architecture in embedded systems and is now also capable of running complex operating systems such as Linux. So the company decided to port the VOSySmonitor solution to RISC-V architecture, calling it VOSySmonitoRV.
They implemented a prototype from which the thesis work is started with the purpose of enhancing it. The company's initial prototype was compiled with Yocto and it included a Trusted execution environment underlying the Linux operating system as GPOS, responsible to boot FreeRTOS as not-yet-working RTOS, on an unused hart. To improve this prototype, FreeRTOS must work on RISC-V in S-mode properly and must be boot with the PMP configured before Linux was booted (again with the PMP configured). Another improvement is the feature of the shared core between RTOS and GPOS. This feature implies that a context-switching between two different operating systems must be done while keeping them strictly isolated. Furthermore, the thesis includes the evaluation of the feasibility of VOSySmonitoRV in criticality systems. To evaluate this, a custom benchmark to measure the interrupt latency and the context switch overhead of a simple ECALL was done.

1.2 Thesis outline

This section is dedicated to having an overview of the thesis report:

- **Background** contains all the background necessary to fully understand the thesis work including the RISC-V architecture, the detail on the SiFive HiFive Unleashed platform used for testing and supporting

the development of VOSySmonitoRV, some notions about Linux device drivers used in the custom performance benchmark, background on Linux Device Tree that is modified to isolate some peripherals at exclusive use of the RTOS, how it works a real-time operating system and what are the linker scripts. Finally, the state-of-the-art of existent solutions on RISC-V architecture is shown.

- **VOSySmonitoRV** describes what is VOSySmonitoRV, showing all the prototypes and their features starting from the company one.
- **Custom Performance Benchmark** includes the implementation details of a custom benchmark that evaluate interrupt latency e context switch overhead of VOSySmonitoRV running on the SiFive Hifive Unleashed board. The results with the same considerations are done.
- **Conclusions** describes the conclusions of the thesis work done and proposes possible future work
- **Appendix** with the performance benchmark code.

Chapter 2

Background

2.1 RISC-V architecture

This Section is focused on the features used to support the development of VOSySmonitoRV. The RISC-V architecture [22] is quickly gaining relevance in the industry and opens up new challenges regarding the security of new processors, especially Linux-capable ones. RISC-V is an extensible instruction-set architecture that was originally designed to support education and research [22]. The privileged architecture [21] allows many possible software stacks that define different execution environments. Every layer has an interface with the higher layer as shown in Fig.2.1 [21]. In RISC-V there is the concept of hardware thread, or hart, that executes the RISC-V code in a specific execution environment with a privileged level and its behavior depends on it in the processing of legal or illegal instructions, interrupt handling, and environment calls. There are three privileged levels: User (U-mode), Supervisor (S-mode), and Machine (M-mode). The M-mode is the highest privilege, the U-mode the lowest. They have a set of private control and status registers and privileged ISA extensions and they are used to protect the execution environments. So in the lower levels, it is possible to access the privileged registers only by environmental calls which are defined as ECALL. This environment call triggers an environmental-call-from-X-mode exception to the more privileged execution environment to do some privileged operation. There exist three types of registers in RISC-V: general-purpose registers, Control and Status Registers (CSRs), and memory-mapped regis-

ters. RISC-V architecture has 32 general purpose registers `x0-x31` per hart, by default. In the next Section we will see the CSRs.

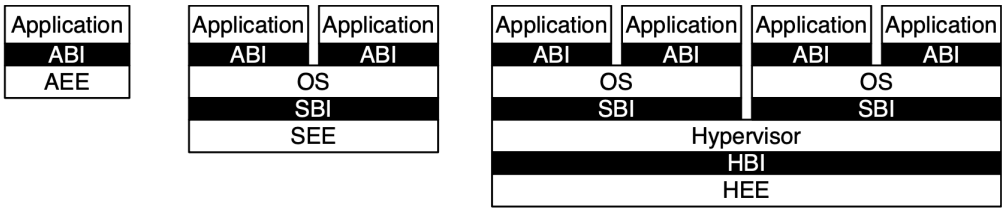

Figure 2.1: Trusted execution environments overview [21]

2.1.1 Control and status registers (CSR) and interrupts

CSRs are SYSTEM instruction, so privileged instructions, that are defined as atomically read-modify-write control and status registers. There exist a set of CSRs for each privileged level, the higher privileged CSR often is composed in part of the lower privileged levels CSRs of the same type.

XLEN-1	XLEN-2 36	35 34	33 32	31 23	22	21	20	19	18	17	
SD	**WPRI**	SXL[1:0]	UXL[1:0]	**WPRI**	TSR	TW	TVM	MXR	SUM	MPRV	
1	XLEN-37	2	2	9	1	1	1	1	1	1	

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS[1:0]	FS[1:0]	MPP[1:0]	**WPRI**	SPP	MPIE	**WPRI**	SPIE	UPIE	MIE	**WPRI**	SIE	UIE
2	2	2	2	1	1	1	1	1	1	1	1	1

Figure 2.2: Machine mode MSTATUS CSR register [21]

For example, the `mstatus` (Machine Status Register) CSR in Fig.2.2, which has the tracks of the hart's operating state and is accessible only by M-mode, has its restricted view in `sstatus` (Supervisor Status Register) CSR accessible by M-mode and S-mode and in `ustatus` (User Status Register) register (accessible by all privileged levels). For interrupts handling, there are a set of registers for every privileged level. Now a brief explanation of how the interrupt mechanism works focusing on the M-mode registers, considering that the structure and the behavior are the same for the other privileged level. So, the `mie` (Machine Interrupt Enable) bit is at 1 when the

interrupts are enabled. Two-level of the stack of interrupt bits are supported, in the `mpp` there is the previous privileged mode (in M-mode two bits wide, `spp` has one bit) and the `mpie` is the interrupt bit before the trap. When there is a trap taken from a privileged level, let's say x-mode, the `mpie` register is set to `mie`, the `mie` is set to 0 and the `mpp` to x-mode encoding. Furthermore, the `mpec` (Machine Exception Program Counter) register contains the address where the interrupt occurs, the `pc` (Program Counter) is set to the `mtvec` (Machine Trap-Vector Base-Address Register) value that is the base address of the interrupt handling. After that, the context switch must be saved, so all the general-purpose registers. In the `mcause` (Machine Cause Register) register, there is encoded the interrupt that causing the trap. After the handling the trap-return instruction is necessary to restore the context and return from the trap handling with `mret` trap-return instruction. When it is executed the `mie` value is set to `mpie` value, that now is set to 1, the privileged mode is set to `mpp`, which is now set to U-mode encoding or M-mode, if U-mode is not supported [21]. Finally `pc` is set to `mepc` to return back where the interrupt occurs. Interrupts can be delegated through the `mideleg` (Machine Interrupt Delegation Register) and `medeleg` (Machine Exception Delegation Register) registers setting. The `mie` (Machine Interrupt Enable) and `mip` (Machine Interrupt Pending) registers contain the information about the enabled and pending interrupts.

2.1.2 Physical Memory Protection

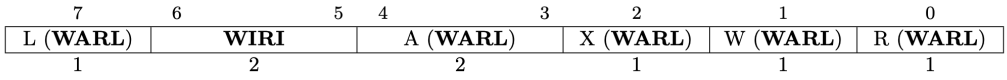

Figure 2.3: PMP configuration CSR register [21]

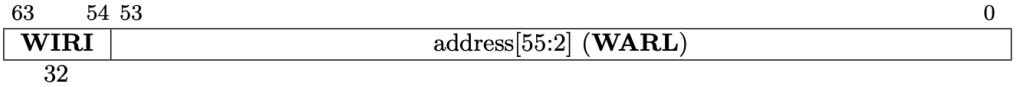

Figure 2.4: PMP address CSR register [21]

To provide also physical protection RISC-V provides a Physical Memory

Protection (PMP) for each hart to allow or not memory access to a specific memory area configuring PMP regions. Only M-mode can program the PMP of a hart and can decide the limit addresses and the permitted operations (Read/Write/eXecute) through configuration registers. By default, M-mode can access every memory area, but it is possible to limit it by locking (setting the L bit 3.3). S-mode and U-mode have no privilege, the PMP must be set to allow memory access. The Address Matching of the PMP is the encoding used to specify the range of addresses for the PMP that can be naturally aligned power-of-2 regions (NAPOT) or naturally aligned four-byte regions (NA4) or top of range (TOR). PMP regions are statically prioritized: a region with an index lower than another has major priority when access occurs. The PMP registers are CSRs and are packed to reduce context switch time. For RV64 there are `pmpcfg0` and `pmpcfg2` to allow the configuration of 16 PMP entries. The `pmpcfg` register format is showed in Fig.3.3 where is clear how to set the Read/Write/eXecute operations, A encodes the address matching, L is the locking bit that if set, indicates that the operations are enforced on M-mode and writing the register is ignored. The L bit is cleared only with a system reset. To set the encoded address of a PMPX region, the `pmpaddrX` must be set according to the encoding in A.

2.1.3 Platform-Level Interrupt Controller

The Platform-Level Interrupt Controller (PLIC) is a hardware unit in RISC-V systems that handling global interrupts. A PLIC is composed of multiple interrupt gateways (one for interrupt source, platform-dependent) and a PLIC core that prioritizes and distributes global interrupts. Each interrupt has an Interrupt Identifier (ID) and a priority to set, the maximum value is platform-specific. Furthermore to enable an interrupt it is necessary to set a threshold. The PLIC mechanism has an interrupt flow that assures correct interrupt handling, other interrupts are ignored until a sort of handshake between the PLIC core and the target hart is complete. This mechanism is well shown in Fig.2.5 from the official documentation [21]. To handle an external interrupt it is necessary to do an interrupt claim, handle the interrupt according to the interrupt source and finally do an interrupt completion.

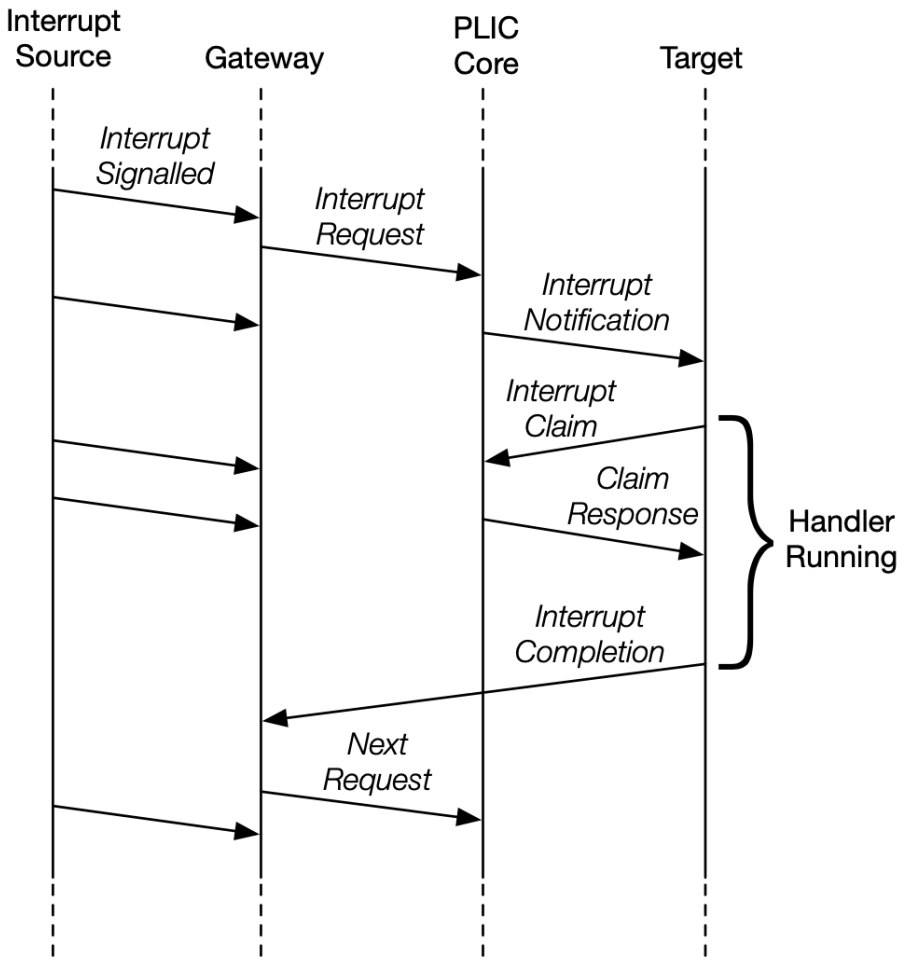

Figure 2.5: Flow of the PLIC external interrupts handling [21]

2.1.4 Performance Counters

In M-mode, the `mcycle` read-only Control and Status Register (CSR) is a privileged RISC-V register with 64-bit precision which holds a count of the number of clock cycles executed by the hart. The `cycle` register is a read-only shadow of `mcycle` and it can be read with the dedicated `rdcycle` instruction from every level of privilege. Technically "the `rdcycle` pseudo instruction reads the low XLEN bits of the `cycle` CSR which holds a count of the number of clock cycles executed by the processor core on which the hart is running from an arbitrary start time in the past" [22]. Along with this, other registers for instructions and events counters. The `time` register is similar to the `cycle` CSR, but counts wall-clock real-time and can be read from others then M-mode through the `rdtime` pseudoinstruction.

2.2 SiFive HiFive Unleashed

The SiFive HiFive Unleashed is used for testing the VOSySmonitoRV prototypes. It is the development platform for the SiFive FU540-C000 SoC [17], the first world multi-core Linux capable RISC-V System on Chip. It is composed of four cores SiFive U54 processor, compliant to RISC-V ISA RV64IMAFDC specification [18], and one monitor core SiFive U51 processor. Furthermore, it is composed of typical processor peripherals such as General Purpose Input Output (GPIO), Universal Asynchronous Receiver/Transmitter (UART), Inter-Integrated Circuit Communication (I^2C), Serial Peripheral Interface (SPI) and Pulse Width Modulator (PWM). The `coreclk` is 1.0 GHz. The L2 cache and peripherals operate at `tlclk`, so at `coreclk`/2 [17].

2.2.1 Interrupts overview

The SiFive FU540-C000 SoC includes both local and global interrupts as can be summarized in Fig.2.7 from the official manual [17]. The platform includes a RISC-V standard PLIC controller described before, that can handle 53 external interrupts with 7 different priority levels. It also has a Core-Local Interruptor (CLINT) that without arbitration triggers a hart with its local

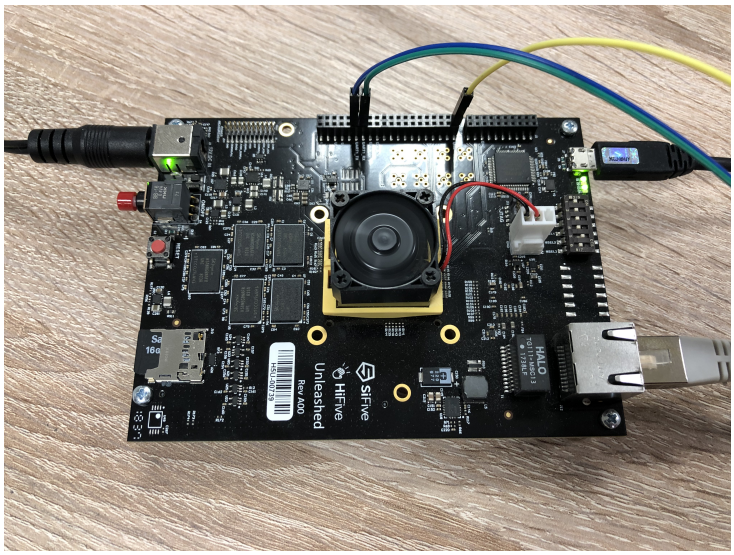

Figure 2.6: SiFive HiFive Unleashed platform

dedicated interrupt.

The interrupt priority depends on the privileged level of the interrupt, internally to the same privileged level we have external, then software, and finally timer interrupts. The priorities are summarized as follow:

- Machine external interrupts
- Machine software interrupts
- Machine timer interrupts
- Supervisor external interrupts
- Supervisor software interrupts
- Supervisor timer interrupts

Core-Local Interruptor (CLINT)

The CLINT block provides M-mode software and timer interrupt to the hart, as we can see in Fig.2.7, and it holds memory-mapped control and status registers related to these interrupt. Timer interrupts are pending if the value of `mtime`, which holds the number of cycles counted from the real-time clock

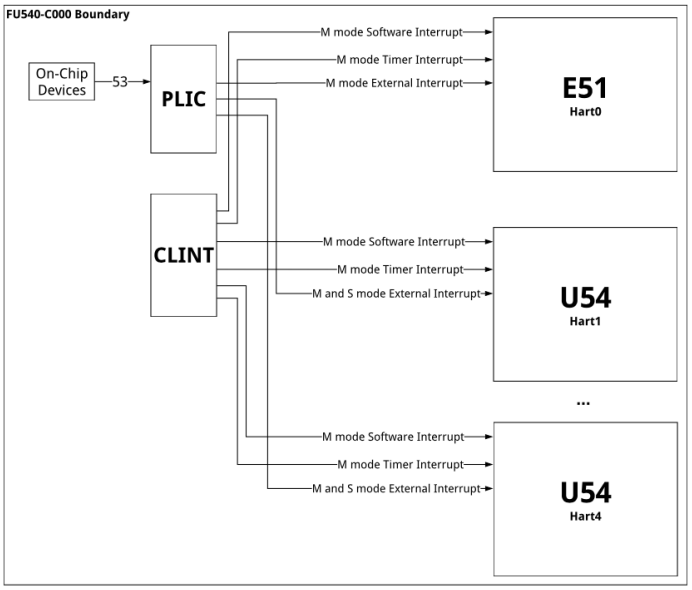

Figure 2.7: Interrupts overview [17]

(RTC), is greater or equal to the `mtimecmp` value, which can be set only in M-mode. To set it by lower privileged levels an ECALL must be done. By default, all interrupt trap to M-mode but it is possible to delegate these interrupts to lower privileged levels via the CLINT block. In this hardware implementation, delegate interrupts allow to manage them from both the M-mode and S-mode in this way: the M-mode interrupt is trap, so it is possible to do something in M-mode and then it is necessary to trigger the S-mode pending interrupt register. The CLINT provides only a timer per hart. For example, for timer interrupts, in the M-mode timer handler the `STIP` (Supervisor Timer Interrupt pending) in the `sip` (Supervisor Interrupt Pending) register can be set at 1 to trigger a S-mode timer interrupt. Software interrupts are triggered by writing the `MISP` (Machine Software Interrupt Pending) bit in the `mip` (Machine Interrupt Pending) register and they are used for inter-processor communication on multi-hart systems.

2.2.2 SoC-related peripherals

The following platform-specific features depend on the hardware implementation of the 64-bit U54 RISC-V processor core used in the FU540-C000 SoC in the SiFive HiFive Unleashed platform. Besides, the SoC-related peripherals used in the prototypes of VOSySmonitoRV are considered.

Universal Asynchronous Receiver/Transmitter (UART)

It is a peripheral for serial communication and FU540-C000 provides two instances of UART: UART0 and UART1. Those are configured as 8-N-1 format through UART configuration registers: 8 bits of data, no parity bit, 1 start bit, and 1 stop bit. The baud rate is 115200 (so the baud rate divisor register has a value of 4340).

Pulse Width Modulator (PWM)

There are two instances of PWM: PWM0 and PWM1. Every PWM has four comparators and it can generate different types of waveforms on the output pin. In the second VOSySmonitoRV prototype we need the PWM to be programmed through its configuration registers as an extra timer that can provide periodic interrupts. The comparator results are latched by the PWM interrupt pending register and routed to PLIC as potential interrupt sources.

2.3 Linux device drivers

The custom benchmark needs a char device driver, so a basic knowledge of Linux device drivers is needed. One method to write a Linux device driver is to build a kernel module so it won't need to recompile the kernel. It is necessary to register the device in the Linux kernel through the `register_chrdev` function with the major and minor numbers, that are identifying the group of devices and a specific device among the group. Then the pointers of the device itself and the `file_operations` structure associated with this character device driver. Moreover, to use the device driver methods in userspace, open and release methods must be implemented, along with other

operations. Device file operations that are possible to use from user space are in the `file_operations` structure and the focus is on the `ioctl` function that is a system call that "offers a way to issue device-specific commands" [4]. To use them all, they must be register in the device driver methods in the `file_operations` structure, otherwise, it returns an error.
The result of the compilation of the kernel module is a .ko extension file. To test the driver it must be loaded and used by a user program. To load the driver in Linux at run time it is therefore required to copy the .ko file on the board and type on the shell the `insmod` command followed by the .ko file. The `copy_from_user` and `copy_to_user` functions can be used in the `ioctl` method to copy the data from the user space to the kernel space, and vice versa.

2.4 Linux Device Tree

During the boot phase, the bootloader is loading the kernel image into memory and the execution switches to the kernel from its entry point. The kernel, at this stage, like any other bare-metal application, needs to perform some hardware initialization and configuration tasks, such as virtual memory configuration, processor configuration, and console configuration. All these are hardware-dependent operations: the kernel must therefore know which register addresses to write, depending on the hardware on which it is run. To do this, a description of the hardware is needed, and it must be easy to change and recompile, hence the idea of using a Device Tree. The Device Tree is a Hardware Description Language that is used to describe the hardware of the system in a tree-like data structure. In this structure, each node in the tree describes a device. The source code of the Device Tree is compiled by the Device Tree Compiler (DTC) to produce the Device Tree Blob (DTB), which can be read by the kernel at boot time. The .dts file is the description of the hardware at the board level.

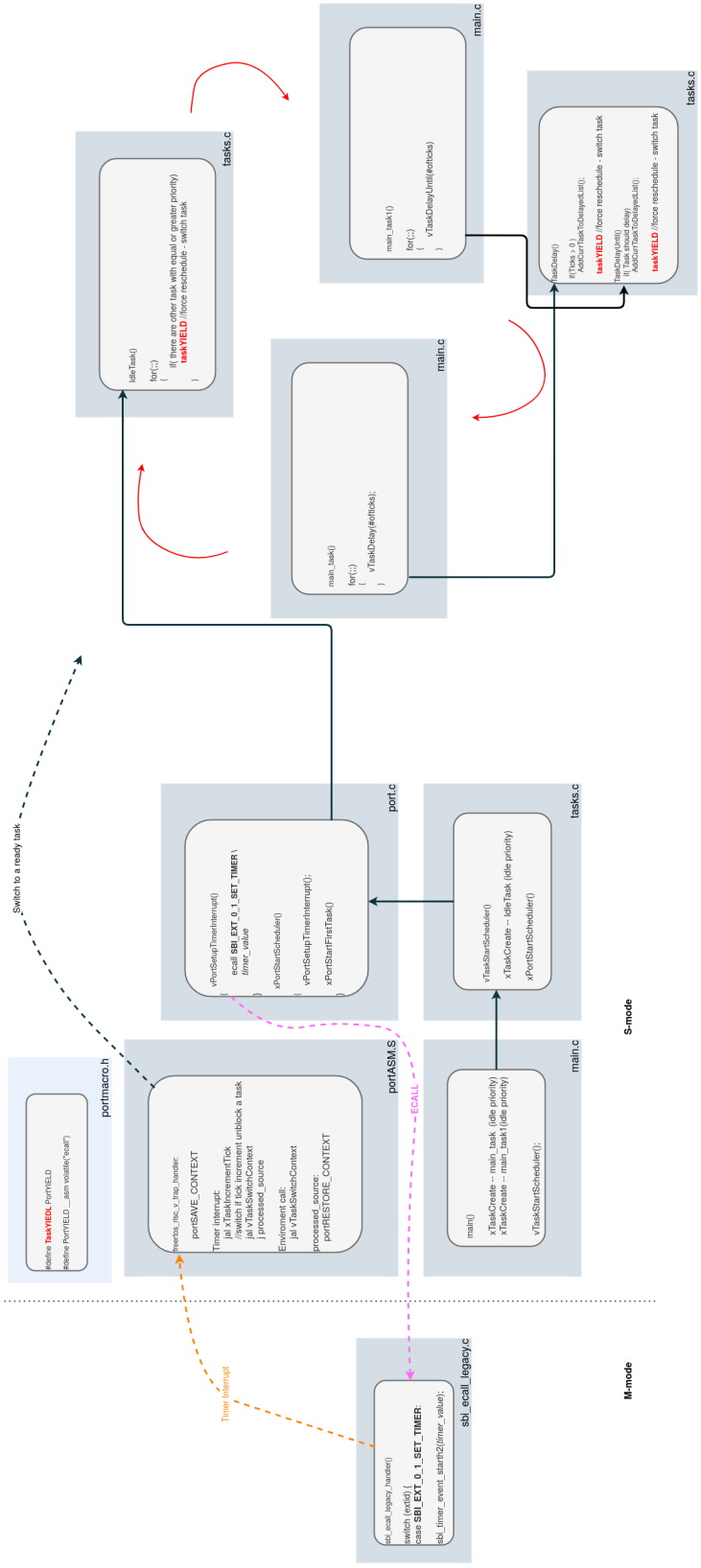

Figure 2.8: FreeRTOS execution overview

2.5 Real-time operating system

Understand how a real-time OS is working was very important to make FreeRTOS working and to correctly porting from M-mode to S-mode. FreeRTOS is a free and supported OS under MIT open source license [1]. It has to be small to be used on microcontrollers in embedded systems applications. The most important thing to know is that a real-time operating system scheduler's purpose is to be deterministic to respond to an event within a deadline and to respect real-time requirements. The scheduler uses the user set priority of the thread execution (task) to know which one can be run next. The main building block is the RTOS tick. Every time there is a timer interrupt the tick counter must be incremented. Therefore the RTOS kernel must check if is the case to unblock or wake up a task. Preemption is allowed to higher privileged tasks. So idle task is always running unless a more privileged task needs to be executed. FreeRTOS execution scheme is shown in Fig.2.8, Task_YIELD() function in red is used to reschedule and switch task, that means the CPU has release the CPU.

2.6 Linker script

This notion is useful for the integration of the RTOS in the basic firmware file. The linker is what combines input files to make a single output file. Each input or output file is an object file with several "sections". Every section has a block of data that can be loadable, so a physical address is needed to specify where the data section must be copied from ROM to RAM when the program starts up. Linker scripts are written in linker script command language. The 'SECTIONS' command is used to define the memory layout of the outcoming file. The '.' symbol is a special one that is the location counter. The linker script is basically written as a series of commands, a keyword with arguments or an assignment to a symbol.

2.7 Related work

Nowadays, there are several solutions with the purpose to offer a Trusted Execution Environment (TEE) exploiting the powerful primitives of the RISC-

V Privileged architecture. The following solutions have been designed on RISC-V for trusted computing and not for critical applications as VOSyS-monitoRV.

Keystone [9] is an open-source framework for building customized TEEs exploiting the so-called enclaves. VOSySmonitoRV adopts the same secure monitor memory isolation: it configures the first PMP region for its code, accessible only in M-mode. On the other hand, Keystone uses a Linux kernel driver to manage the enclaves via SBI and it has to handle the enclave allocation dynamically configuring every time the PMP set of every core. In VOSySmonitoRV the dynamic PMP configuration is done only on the shared core, which is less costly in terms of latency and performance.

Sanctum [5] is a minimal hardware extension, provides secure monitor software **Sanctorum** [8], and also relies on the untrusted OS to enclave handling, that is executed in U-mode.

MI6 is an out-of-order processor [3] which extends RISC-V with the same hardware modifications with a greater focus on protection from known attacks. It is basically focused on the creation of enclaves. It also uses a trusted security monitor that executes in M-mode.

Donky [15] is a hardware-software co-design for memory isolation of user processes. It has a secure monitor called Donky Monitor, part of the software design, that handles in-process access policies in userspace. Donky is used basically for memory isolation of protection keys and it has no kernel interaction, using the RISC-V 'N' extension (the user-level interrupts extension).

On the other hand, solutions for a mixed-criticality environment exist. A similar approach to VOSySmonitoRV is seen in the **MultiZone** solution [6] that creates different "Zones", running in U-mode, with a defined memory portion and peripherals to allocate to each zone through a configuration file. The execution of the Zones is scheduled by the MultiZone nanoKernel with a round-robin or cooperative scheme. The main difference is that Linux cores are not shared with the Zones cores.

Bao [12] is a security bare-metal hypervisor that focuses on mixed-criticality systems. It provides minimal monitor virtualization support for the static partitioning architecture. It is mainly developed for ARMv8 architecture platforms, the RISC-V port depends on the hypervisor extension and for

now, is only deployed on the QEMU emulator. Bao is used to managing VMs, interrupts are mapped to physical ones and also virtual to physical CPU. VOSySmonitoRV embraces the same philosophy, the aim is to safely manage resources, with no performance overhead, fault-containment, and real-time constraints, except for the choice to use the hypervisor extension and the static partitioning of the CPUs. Efficient allocation of resources is one of the focuses of VOSySmonitoRV.

SiFive WorldGuard [19] is a security hardware model that offers World ID markers to every hart and every process to protect and isolate different domain execution (data and code). VOSySmonitoRV can easily integrate this SiFive hardware feature as future work.

Chapter 3

VOSySmonitoRV

VOSySmonitoRV is a mixed-criticality solution that provides spatial and temporal isolation between co-executing operating systems (OSes) on a multi-core RISC-V processor. It is a software layer that handles the memory area partitioning and isolation in terms of memory, permissions, peripherals, interrupts, and harts, but also software execution. VOSySmonitoRV makes sure that the safety-critical and trusted OSes are booted before un-trusted OSes, guaranteeing for the former best boot time and certifiability. Secure boot is fundamental to the reliability of the real-time operating system and to enforce a chain of trust. Besides, VOSySmonitoRV can provide additional services to the operating systems running on the platform, e.g., power management, trusted execution environment, custom vendor-specific functions, etc. VOSySmonitoRV features include platform initialization, secure interrupt handling, secure boot, and all operations not accessible to lower privilege levels. To achieve these operations, operating systems must execute ECALLs that can be trapped in M-mode and managed by the monitor.
Moreover, another important feature is isolation between operating systems. As a matter of fact, VOSySmonitoRV can easily detect and isolate attacks from an un-trusted OS that may propagate and lead during the execution of safety-critical OS. Therefore is very important to meet the stringent real-time constraints of it to hold a high level of reliability in a mixed-criticality system. To achieve this, VOSySmonitoRV exploits standard features of RISC-V's privileged architecture [21], which aims for greater security between the different privilege levels by providing dedicated registers for each of them.

Furthermore thanks to the Physical Memory Protection (PMP) unit, which is explained in detail in Section 2.1.2, it is also possible to define at boot time and run-time, for each hart, different memory partitions with certain permissions, always and only from the highest privilege level i.e., VOSySmonitoRV. For architectural choice, VOSySmonitoRV is not developed at the hypervisor level to not rely on third-party code for firmware and operating systems booting. Although the hypervisor extension provides more flexibility and performance, the security of a monitor with a limited attack surface, acting at the highest level of privilege is more reliable. In fact, VOSySmonitoRV is working at Supervisor Binary Interface (SBI) level, as we can see in Fig.2.1. One of the features of VOSySmonitoRV is the co-execution of the OSes on a single hart. This need comes from the fact that real-time operating system (RTOS) workload can be characterized for a long time by scheduled idle tasks and it is a waste of resources. That will be further discussed in Section 3.4 where the second prototype is described.

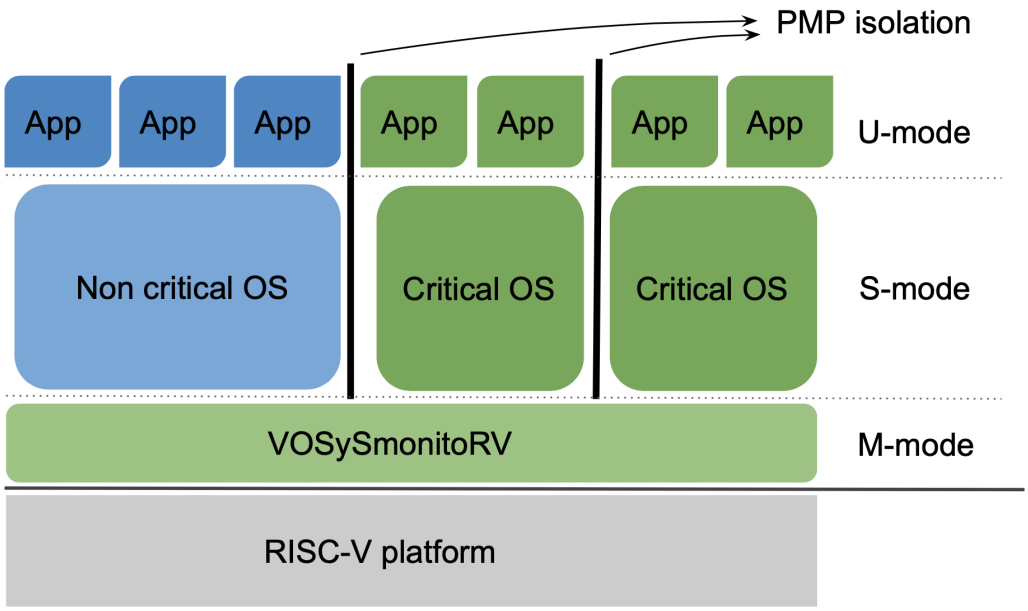

Figure 3.1: VOSYSmonitoRV Architecture Overview

ECALLs are managed in VOSySmonitoRV in such a way that they are recognized by an extension ID and a function ID, they may include some arguments. As mentioned before, to test the features and support the de-

velopment of the prototypes of VOSySmonitoRV it was used HiFive SiFive Unleashed, as a secure real-time operating system FreeRTOS and finally Linux as a generic and rich operating system. FreeRTOS starting version was in M-mode, but we need it to be in S-mode. The Linux version used is v5.8, which already works on 64-bit RISC-V in S-mode.
This chapter is briefly described the design specification of such a system and then the prototypes realized before and during the thesis work.

3.1 VOSySmonitoRV design specification

The purpose of VOSySmonitoRV is to create a trusted execution environment (TEE) for one or more RTOS. TEE is a protected part in which it is supposed to have only trusted software. TEEs are often designed to be alongside a rich execution environment that runs a conventional OS.
The target applications of VOSySmonitoRV are secure embedded devices in mixed-criticality environments. Such as system has the following design goals:

- **High-security and isolation**: the monitor has to strongly and effectively isolate the execution environments leveraging the PMP technology and the RISC-V privileged levels;
- **Real-time features**: having a real-time OS in which the mission-critical tasks must have the full priority;
- **High-performance and small footprint**: the firmware must have a small footprint with limited attack surface and be performing and efficient during crucial operations such as context switches and PMP configurations (so to be implemented in assembly);

3.2 Starting prototype

The starting company prototype of VOSySmonitoRV is a basic firmware that executes in M-mode on the machine. It boots U-boot, which boots Linux on three harts over four. To make this happen it is necessary to change the kernel parameter (`CONFIG_CMDLINE`) in the Linux configuration file (`.config`)

adding to the existent command line `maxcpus = 3`. In this prototype, FreeRTOS is booted from VOSySmonitoRV on the unused hart, triggered by an ECALL from a custom Linux driver. To do so it is necessary to copy the FreeRTOS binary on the board. To better understand how FreeRTOS is boot is important to know that in VOSySmonitoRV every hart, at reset, starts to execute the cold boot, and at a certain point, all the harts are in wait for interrupt (WFI), except for the cold boot hart. The cold boot hart executes platform initialization code and finally writes in `mpec` CSR the next instruction address to execute (in this prototype the address of U-boot first instruction). It also writes in the `mpp` the next mode in which execute after the Trap-Return instruction, so S-mode, (see Section. 2.1.1) and in the `mpie` writes 0. In the very end, it executes the `MRET` instruction, to start executing U-boot in S-mode. When Linux is operational, the driver is loaded and the application is executed. Through an `ioctl` function (see Section.2.3) the driver is triggered to copy the binary from user space to a specific physical address in the physical memory. In this way, it can be booted from VOSySmonitoRV in response to the ECALL from the driver (in M-mode are used only physical addresses). So one of the Linux harts when executing VOSySmonitoRV triggers a software interrupt to the unused hart. The unused hart is waked from the WFI, it finishes to execute the platform initialization and boots FreeRTOS with the same procedure as for U-boot. The FreeRTOS version, at this stage, was not working properly yet.

3.2.1 Simulation on Quick EMUlator (QEMU)

One interesting thing to do, when it is developed software like VOSySmonitoRV, is to use a QEMU emulator. The advantage to use QEMU in a company consists of being cheap and easy than buying a board and very powerful for development because of the debug environment and quick prototyping. VOSySmonitoRV starting prototype was simulated on QEMU `virt` machine and the emulated SiFive board.
In order to correctly emulate, first of all, it is necessary to compile QEMU with architecture target RISC-V on 64 bits by running in the folder `./configure --target-list=riscv64-softmmu`. Then compiling a rootfs is needed, we choose **busybox** that generates an image `rootfs.img.gz`

to be used during QEMU emulation. This is important also because efficient use of QEMU is fundamental for development or testing so managing the transfer of files between the guest and the host is a must. Other solutions are possible by using an emulated network.
The following command is used to emulate VOSySmonitoRV firmware on `virt` machine:

```
qemu-system-riscv64 -M virt -smp 4 -m 2G -nographic
        -bios vosysmonitorv/plat/gen/firmware/fw.bin
        -kernel linux/arch/riscv/boot/Image
        -drive file=busybox/rootfs.img,format=raw,id=hd0
        -device virtio-blk-device,drive=hd0
        -append "console=ttyS0 ro root=/dev/vda earlycon=sbi"
        -serial mon:stdio
```

To emulate the SiFive, the target machine is not `virt` anymore but `sifive_u`. To emulate this machine we use another method because unfortunately now we don't have support for the -hda option. This alternative leverages the `initrd` method that creates a `ramdev` block device, it is ram based instead of using physical disks. Then we create a gzipped cpio archive from busybox, which is extracted and mounted as a root file system. The other difference is that the console is the ttySIFO uart0 of the SiFive. The resulting command to run is the following:

```
qemu-system-riscv64 -M sifive_u -smp 2 -m 1G -nographic
    -bios vosysmonitor/plat/sifive/firmware/fw.bin
    -initrd rootfs.img.gz
    -kernel linux/arch/riscv/boot/Image
    -append "console=ttySIF0 root=/dev/ram0 rdinit=/sbin/init
    earlycon=sbi"
```

To know all the options please refer to official QEMU documentation.

3.3 First prototype

The most important feature of the first VOSySmonitoRV prototype is to boot FreeRTOS before U-boot and Linux, with the PMP configured in both cases.

This is also done for security reasons, booting first the critical OS avoids that the non-critical OS can do some manipulation or causing the RTOS not to boot because of a crash or a fault. At reset, VOSySmonitoRV is initialized and immediately takes care of properly configure the PMP device in order to guarantee absolute isolation between operating systems. In all the harts it isolates its own code with no operations allowed at the lowest privilege levels.

3.3.1 PMP configurations

Each operating system has its own PMP configuration. A good strategy to program the PMP is to identify the forbidden parts of memory for that hart when executes one of the OSes and remove permissions. So then configure the last PMP, with the lowest priority, with all the memory and all permissions. In this way sort of "holes" are created in memory where the hart cannot access. When the hart executes Linux cannot access the FreeRTOS memory area, the UART1, which is reserved for RTOS output. Vice versa, FreeRTOS cannot access Linux and UART0. In order to prevent Linux from initializing the UART1 and generating an exception from the PMP for trying to access a prohibited memory area, the UART1 has been removed from the Linux DTS and therefore also from the Linux DTB (see 2.4). As for the UART peripheral, it is possible to isolate any other peripheral mapped on the System-on-Chip (SoC).
More specifically PMP regions when the hart executes FreeRTOS are:

- PMP0: VOSySmonitoRV code, access allowed only in M-mode
- PMP1: Linux code, no permissions
- PMP2: UART0, no permissions
- PMP3: all memory, all permissions

When the hart executes Linux:

- PMP0: VOSySmonitoRV code, access allowed only in M-mode
- PMP1: FreeRTOS code, no permissions

- PMP2: UART1, no permissions
- PMP3: all memory, all permissions

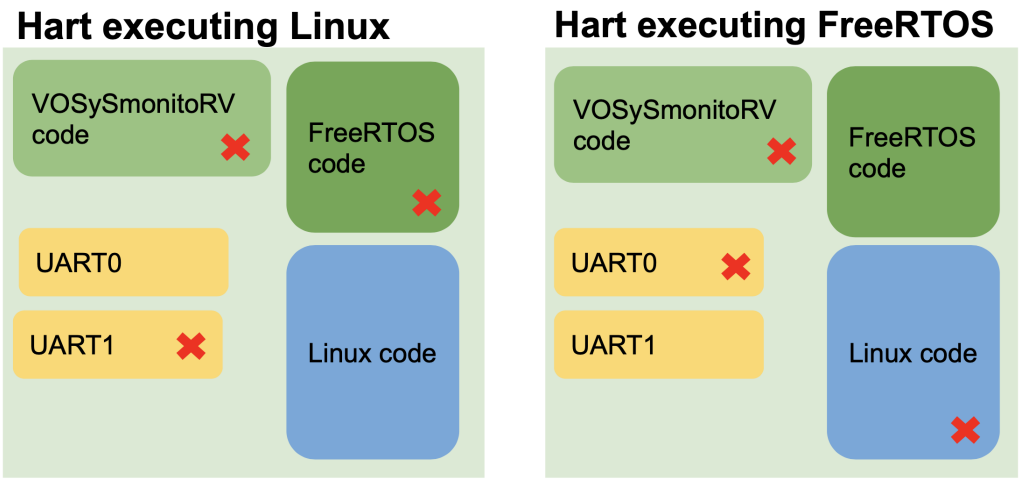

Figure 3.2: Memory overview in the first prototype of VOSySmonitoRV

These configurations are more clear looking at the memory map in Fig.3.2. VOSySmonitoRV is executed in M-mode in all hart when privileged operations must be done and can access by default to all memory regions.
It is important to specify that PMP configuration is done before the hart actually executes the operating system, so it is done at boot time.

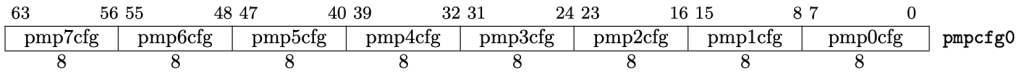

Figure 3.3: `pmpcfg0` register, 8 PMP regions possible [21]

Configuring the PMP is explained in detail in Section 2.1.2. The U54 processor [18] supports 8 PMP entries of a minimum of 4 bytes region, whereas there is only one `pmpcfg0` register implemented that is not hardwired at zero. So as shown in Fig.3.3 to configure, for example, PMP2 region for the Linux harts, the `pmp0cfg` byte must be set with the permissions in the first three bits, so all zeros and how the address is encoded, in this case, NAPOT. In order to avoid that the Linux harts can access in any way, the UART1 is necessary to lock the region by setting the locking bit. Then `pmpaddr2` must

be set with the NAPOT encoded address of the UART1, which start address is 0x10011000 and its memory map registers are wide 4 kbytes. In the U54 processor reference manual we can find how to correctly program the PMP of this processor. It is written: "A = 0x3: Naturally aligned power-of-two region (NAPOT), 8 bytes. When this setting is programmed, the low bits of the pmpaddrX register encode the size, while the upper bits encode the base address right-shifted by two. There is a zero bit in between, which we will refer to as the least significant zero bit (LSZB)" [18].

3.3.2 FreeRTOS porting and VOSySmonitoRV timer handling

FreeRTOS is an open real-time operating system 2.5. A FreeRTOS porting to RISC-V architecture already exists [2] but it is developed to be run in M-mode. Since it has to be co-executed with another operating system, the RTOS must be executed at a lower privilege level i.e., S-mode, like Linux kernel. Therefore FreeRTOS was ported from M-mode to S-mode. The porting consists of modifying the registers accessible to M-mode in S-mode ones, setting and modifying the interrupt handling related registers, and the Supervisor Cause Register (`scause`) values to be checked after the trap. Finally, the timer setting through an ECALL. The timer setting is fundamental for the RTOS task scheduling, it cannot be done in an S-mode environment but only in M-mode. So VOSySmonitoRV when receiving this particular system call with the related extension ID, it knows how to set the timer of the FreeRTOS hart (from which is received the ECALL) according to the wanted frequency. In this prototype, FreeRTOS runs alone on a specific hart where VOSySmonitoRV boots it after platform initialization. Handling the timer interrupt was critical because the CLINT (see Section.2.2.1) can trigger only M-mode timer interrupt, as said, so it is first reached by M-mode. This is right for our purposes because we want to handle it in both M-mode and S-mode. So even if the timer interrupt is delegated, it is first handled by M-mode in VOSySmonitoRV, the timer interrupt is trapped and processed, then the related bit in the `sip` register is set to trigger a timer S-mode interrupt to be handled in FreeRTOS.

3.3.3 FreeRTOS boot

The first thing to do is to add the FreeRTOS binary to the firmware on the flash, to avoid booting first Linux for the only purpose to copy the binary on the board. In this way it is also well known FreeRTOS first instruction physical address, that is useful to boot the OS. To do so it is necessary to change the linker script (see Section 2.6) adding this memory partition:

```
/* Copyright (C) 2020 - Virtual Open Systems SAS
*  Author Flavia Caforio <f.caforio@virtualopensystems.com> */
SECTIONS{
...

. = FW_TEXT_START + 0X30000;
.freertos :
        {
                PROVIDE(_freertos_start = .);
                *(.freertos)
                . = ALIGN(8);
                PROVIDE(_freertos_end = .);
        }
...
```

Then through modifying the fw.S file we add the `.freertos` section where is possible to include the binary through the `.incbin` directive. To pass the binary is better to pass directly the path of it, so the Makefile is modified in order to include it in the VOSySmonitoRV compilation. The final firmware `fw.bin` is then copied on the flash through the `dd` Linux utility. So, FreeRTOS boot is done during VOSySmonitoRV cold boot with the same technique as the starting prototype i.e., by writing in `mpec` CSR the address of the first instruction of FreeRTOS, change `mpp` to S-mode (1), `mpie` to 0 and finally `mret` instruction, so the hart can start to execute the real-time OS.

3.3.4 Isolation tests

After the realization of the first prototype some isolation tests are done on the SiFive HiFive Unleashed board:

- The simpler test to do is to try to access one of the forbidden PMP regions from the non-critical OS. In this test, there is an attempt to write a character on the UART1 with a Linux application (U-mode). The result is an exception to the Linux hart that tried to access a PMP region with no permissions. In the meanwhile, the RTOS operates as usual.

- Another test is to push the Linux harts to work at a maximum rate through a stress test (stress-ng). FreeRTOS has a task that has to make a print every 1000 ms. This test demonstrates that FreeRTOS can meet the deadline of its periodic task even if the Linux harts are used at 100%. This test is shown in Fig. 3.4 where we can see in the HTOP console that the Linux harts are under stress and that FreeRTOS keeps working regularly.

- An other isolation test is to trigger a kernel panic in Linux with the following command on the Linux shell:

  ```
  echo c > /proc/sysrq-trigger
  ```

  This is done while FreeRTOS has the same task as before, in which it has to meet a deadline regularly. In Fig. 3.5 we can see that the kernel panic does not affect the FreeRTOS operations.

The last two tests on VOSySmonitoRV's first prototype are shown in a demonstration video that has been realized during the thesis work [7].

3.4 Second prototype

The second prototype of VOSySmonitoRV main feature is the shared hart. As mentioned, the RTOS workload is often characterized by only idle tasks.

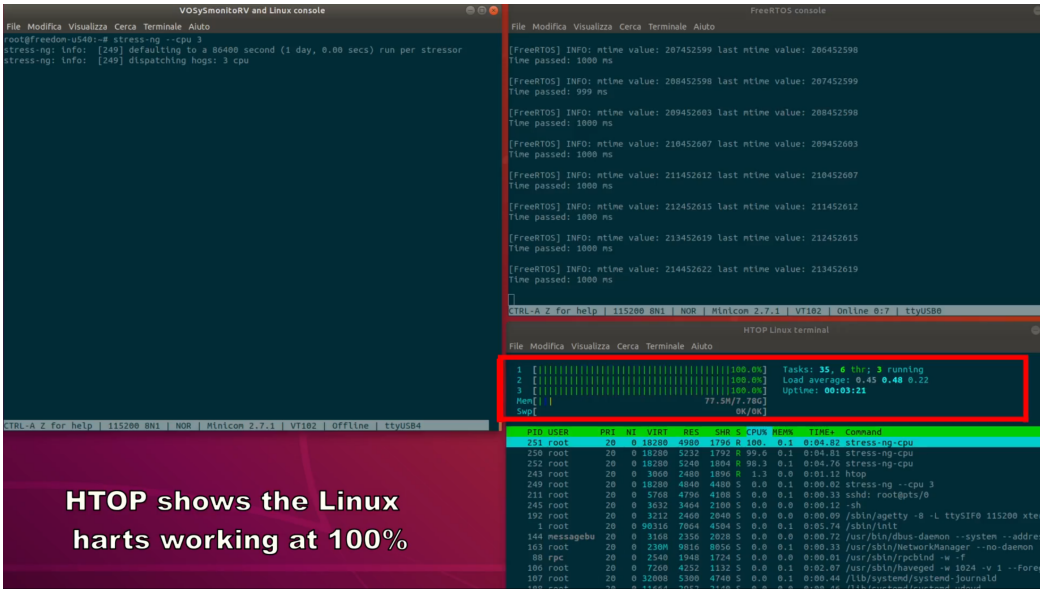

Figure 3.4: Stress test on Linux harts

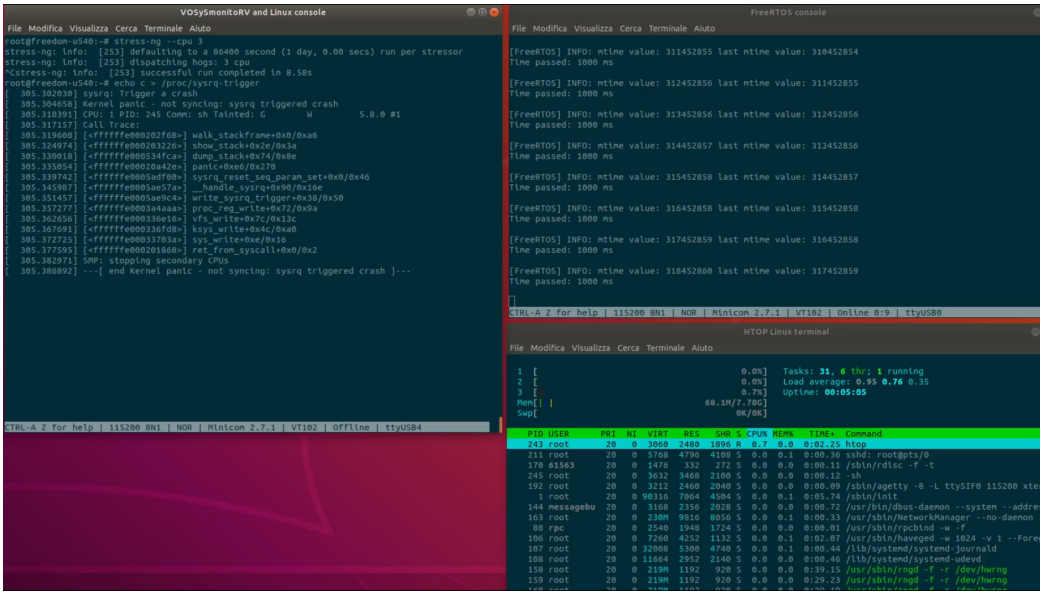

Figure 3.5: A triggered kernel panic in Linux

As a further performance improvement, use this time in which the hart resource is wasted executing Linux code on the RTOS hart. This feature already exists on ARM architecture with VOSySmonitor product, so it is interesting to port it on RISC-V architecture. To do so, a context switch between operating systems is necessary. This second prototype doesn't work properly but, the company is now working on the development of this prototype to make it works correctly. So the implementation, that will be discussed in this Section, is on the last modification done during the thesis. Sharing hart brings two critical issues: the context switch between the operating systems and the timer. The context switch issues are due to the fact that both operating systems use the same M-mode handler even during normal operations, so a clean context switch between privilege modes must always be guaranteed.

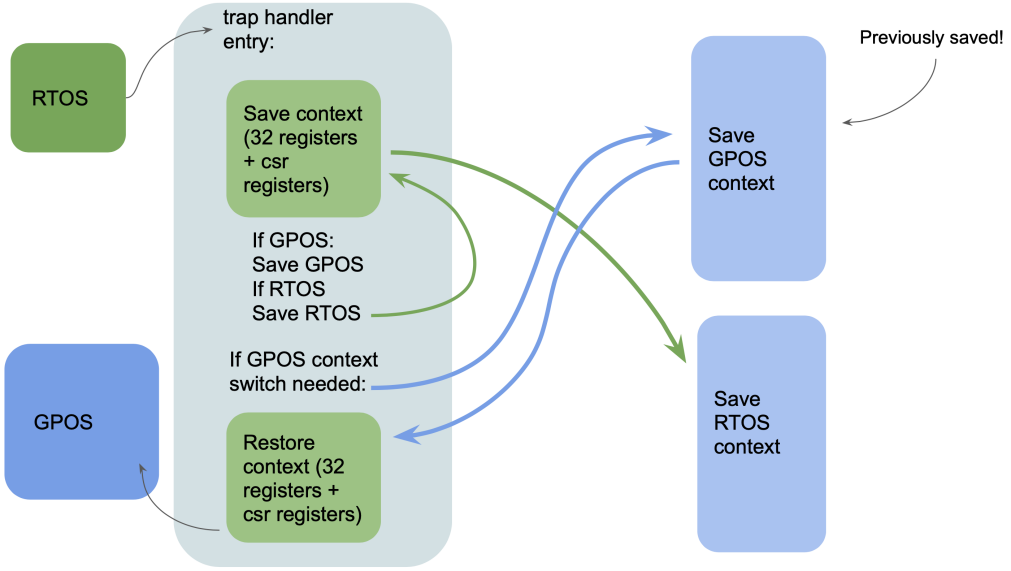

Figure 3.6: Context switch from the RTOS to the General Purpose OS when is required by the RTOS

3.4.1 Main modifications to VOSySmonitoRV firmware

Context switching between operating systems requires saving all useful registers every time there is an interrupt of whatever kind. There is a global variable, WORLD, in which is saved a flag that indicates if the hart is executing Linux or FreeRTOS. This variable is checked so the registers can be

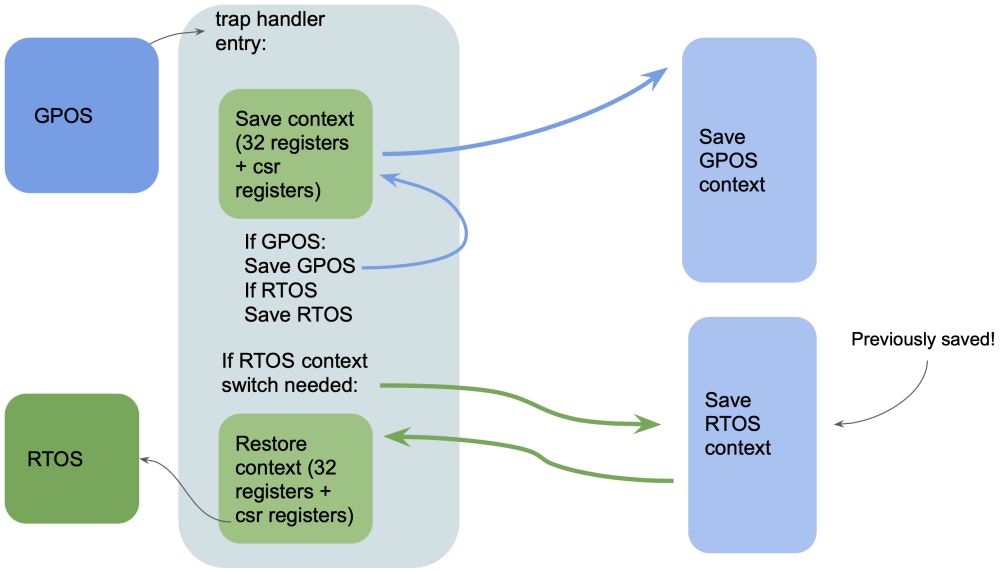

Figure 3.7: Context switch from the GPOS to the RTOS when is required

saved correctly. This operation consists of the saving from the stack of the 32 general-purpose registers and the main CSRs that support the correct execution of the operating system, such as `mstatus`, `sstatus`, `stvec`, `spec`, `sie` and `sip`. The idea is to save the registers of each operating system in a dedicated memory area. In case it is necessary to make a context switch, based on the target operating system, the corresponding registers, the same ones that were saved, are restored. This mechanism can be more clear looking at Fig.3.6 and Fig.3.7 .

The timer issue depends on the CLINT which provides only one timer per hart and shares the same timer is not possible and not useful (the interrupt can be lost or not correctly handled during the OS context switch). The solution is to use a separate timer e.g., the on-chip PWM peripheral as a timer for the RTOS. The PWM generates machine external interrupts that have the maximum priority [17] over software and timer interrupts. In ARM architecture this was not a problem because the ARM solution leverages ARM TrustZone technology, so both the OS has one timer each. Every time there is an OS context switch, the PMP is reconfigured on the shared hart according to the destination OS. The PMP configurations are the same as the first prototype except for the Linux one that now has also the PWM0

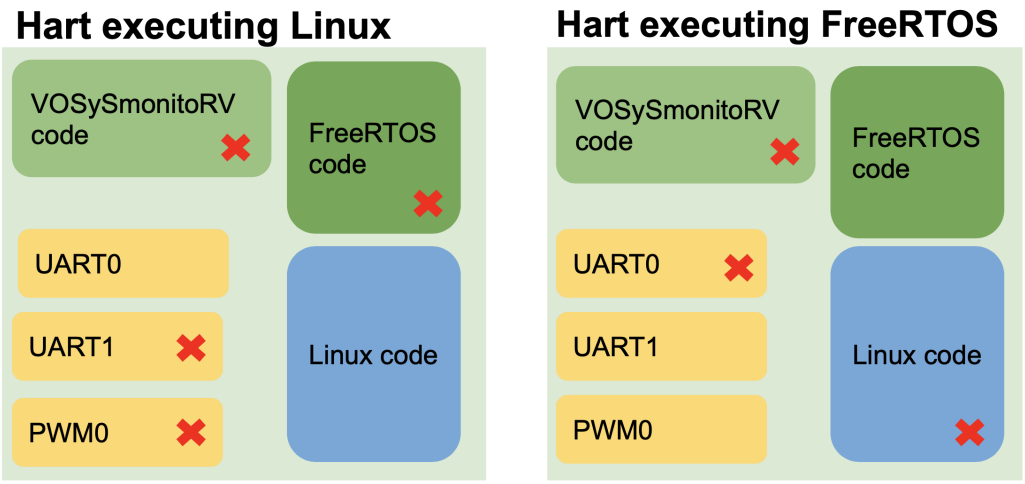

Figure 3.8: Memory overview in the second prototype of VOSySmonitoRV

as a third PMP region. The resulting memory overview is in figure 3.8. To ensure that the RTOS can always be available, every time that its timer is elapsed, VOSySmonitoRV gives back the hart to FreeRTOS execution, so it can check if has an important task to do or not. This is done in the PWM0 handler, where there is also the `pwmcmpXip` bit cleaning. The PWM0 handler is in the external interrupt handler, added from scratch with this purpose since before it was not implemented. In fact, it is added also the enabling of the global external interrupt in M-mode either in S-mode in the `mie` and `sie` registers. To receive the interrupt is necessary to configure properly the PLIC. The PLIC has a register map with configurable interrupts enable and priority threshold for every global interrupt source (53) to be configured for each hart. So it is necessary to configure the registers corresponding to the FreeRTOS hart, enabling the interrupt source `PWM0` and setting the maximum threshold (7). The external interrupt handling process with the PLIC is composed of the claim process and interrupt completion. When an external interrupt is a trap on a specific hart, to know from which interrupt sources are coming, it is necessary to read the `claim/complete` register in the PLIC register map. Then the interrupt can be handled correctly knowing the interrupt source ID. Finally, the same ID must be written in the same `claim/complete` register to signal to the PLIC that the interrupt has been handled. This is the process described in Section 2.1.3, here explained more

specifically for the SiFive HiFive Unleashed board. So an ECALL to be used by FreeRTOS to set the PWM0 and the PLIC is added. The PWM0 setting is done following the official documentation [17]. The bits in the `pwmcfg` register to be set are:

- `pwmzerocmp` resets to zero the counter of the PWM when it reaches the compare register. Usually is done to make a regular timer
- `pwmenalways` means that the counter never stops.
- `pwmsticky` ensures that `pwmcmpXip` is not cleared to be trapped as interrupt in the external interrupt handler
- `pwmdeglitch` latch the `pwmcmpXip` value within the same cycle

The PWM has a counter scale, the counter is compared with the `pwmcmpX` value, where X is the index of the compare registers (there are four). Being able to share the hart might question the total isolation and availability of the RTOS, VOSySmonitoRV ensures the highest priority to the critical operating system for safety reason using the PWM0 as a timer with the maximum priority, while giving Linux performance very close to the native one.

3.4.2 FreeRTOS modifications

To achieve this feature it is necessary to modify FreeRTOS in such a way that it can ask VOSySmonitoRV to change OS on the hart because it releases the CPU. This is done, in the version that works with ARM architecture, through the SVC instruction, which is a system call. In the same way in RISC-V, when in FreeRTOS a task_YIELD() (see Section. 2.5) function is called, an ECALL is done and so VOSySmonitoRV can switch the context. As said before, the timer is now the PWM0. To let the FreeRTOS structure as the same as before, the ECALL to set the timer is substituted by the new ECALL to set the PWM0 and the PLIC.

Chapter 4

Custom Performance Benchmark

The initial company VOSySmonitoRV prototype performance has been measured to assess its overhead and responsiveness in mixed critical applications. More, in particular, the interrupt latency and the context switch overhead between Linux and M-mode has been measured to assess the time needed by an operating system to request one of the services provided by VOSySmonitoRV or to trigger a context switch (when multiple OSes share a hart). As said, this is done also to further optimize the performance of the GPOS, the hart on which the critical OS is running can be used by the non-critical OS during scheduled idle tasks in the workload of the real-time operating systems. Furthermore, to check the feasibility of a shared hart, it is necessary to see if a context switch between operating systems is too costly thus compromising the RTOS safety. The benchmark purpose is to compute the duration in clock cycles of the interrupt latency and interrupt context switch of a simple ECALL from kernel space (S-mode) to M-mode, using one of the performance counters in the FU540-C000 SoC. The performance counter used in this benchmark is described in Section 2.1.4. The custom benchmark will be presented in the following, the description of the environment in which the benchmark was done, then the implementation, and finally the results.

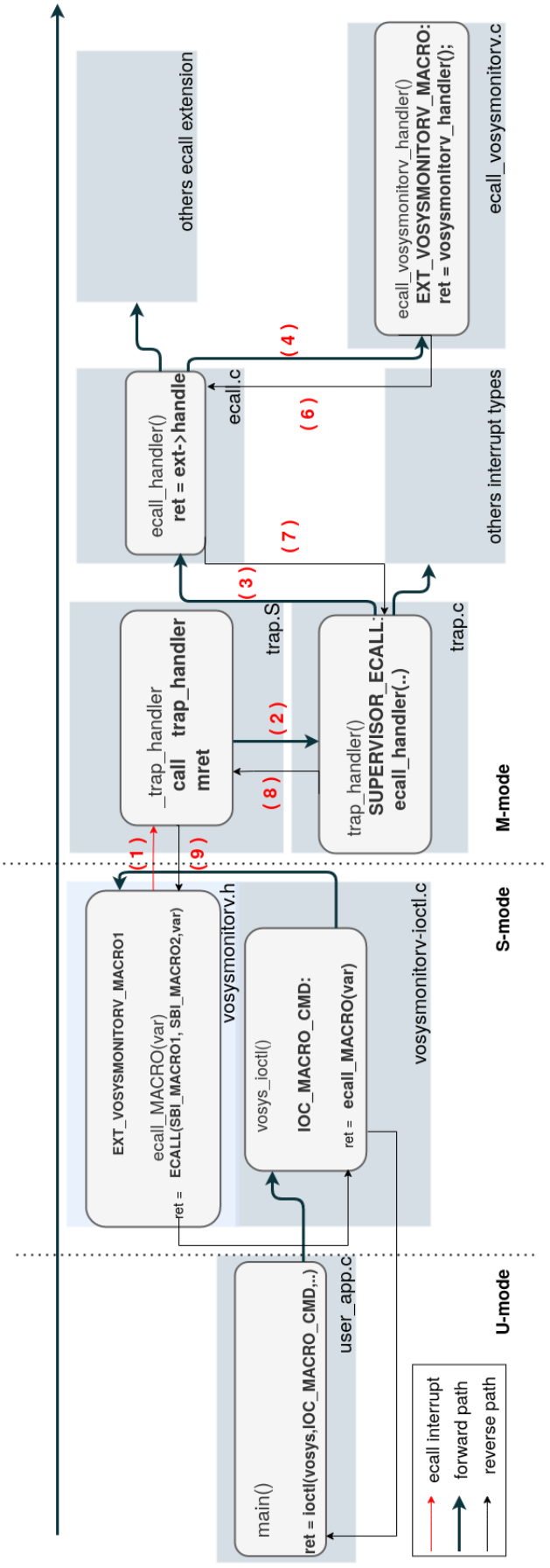

Figure 4.1: Benchmark program flow diagram

4.1 Environment and implementation details

For the implementation of this benchmark, there were developed an application (U-mode), a dedicated Extension in VOSySmonitoRV (M-mode), and a Linux driver that would operate in S-mode to make VOSySmonitoRV ECALLs. VOSySmonitoRV as previously mentioned is working at the SBI level, these S-mode ECALLs are trapped in the main trap handler but then processed in a specific handler that is defined as ECALL VOSySmonitoRV handler. In the following will be references to the red number on the arrows (Fig.4.1).
Measurements were done in mainly two variants:

- **ECALL interrupt**:
  - **Interrupt latency**: from the driver to the ECALL VOSySmonitoRV handler function, or in other words, the delay between the ECALL occurring and the first instruction in the handler of that interrupt. This is the path from `(1)` to `(4)` (Fig.4.1).
  - **Interrupt response**: from the ECALL VOSySmonitoRV handler back to the driver, basically the interrupt response latency of an ECALL. In Fig.4.1 is the path from `(6)` to `(9)`.
  - **Interrupt handling overhead**: from the driver to the ECALL VOSySmonitoRV handler function and back. This is the path from `(1)` to `(9)` (Fig.4.1). It represents a full timing overhead to handle a very simple ECALL.
- **Interrupt context switch**:
  - **Save context**: from the driver to the VOSySmonitoRV main trap handler. It mainly consider the hardware latency and the saving of the context (32 registers). This is the transition `(1)` (Fig.4.1).
  - **Restore context**: from the VOSySmonitoRV handler to the driver. It consider the restore of the context (32 registers) and the hardware latency. This is the transition `(9)` (Fig.4.1).

To compute the context switch basically, it is necessary to read the `cycle` counter register through the `rdcycle` instruction (see Section.2.1.4) first in

the starting point and then in the final point and compute the difference. For example in the **Interrupt context switch** (Save context) measurement, the performance counter is read in the driver before the ECALL and then again after the context saving. The choice to use `rdcycle` is because using `rdtime` would generate an exception that needs to be handled to read the RTC value, as it is outside the core. This is an architectural choice on SiFive's part. Instead the `mcycle` register is a "Fixed-Function Cycle Counter" [18] that is hardware wired, only counts that specific event and it is on 64 bit. Since it is difficult to isolate the number of clock cycles per hart, according to the official documentation [22], Linux is forced to run with one processor core by changing the boot arguments with *maxcpus = 1*. This can be done at run time, by changing it through U-boot through the command:

```
$ setenv bootargs "root=/dev/mmcblk*␣rw␣console=ttySIF0␣
                        earlycon=sbi␣maxcpus␣=␣1"
```

Or can be done through changing the related kernel parameter (`CONFIG_CMDLINE`) in the Linux configuration file (`.config`).

So the application executes in userspace that makes an `ioctl` call (see Section.2.3) to trigger the driver to do an ECALL, that will be trapped in VOSySmonitoRV. This operation in the application is done 500 times to collect enough results to do statistical consideration about the duration of the interrupt latency and context switch overhead that is better presented in the results Section 4.2.

4.1.1 VOSySmonitoRV application

Looking at Fig.4.1 starting from the left we can see the user application running in U-mode. In this file there are two functions `ECALL_test()` and `context_test()` that are called in the main.

- `ECALL_test()`: in this function there is a simple `ioctl` system call (see Section.2.3) with arguments `VOSYSMONITORV_IOC_ECALL_INTERRUPT` and `cycles_value`. This variable is set to 0 if the computation has to be from the driver to the SBI function (**interrupt latency**). It is set to 1 if the computation target is from the SBI function to the driver (**interrupt response**). After the `ioctl` function

in the same `cycles_value` variable there will be the value of the interrupt latency (or response) in clock cycles.

- `context_test()`: also in this function there is a simple `ioctl` function, the arguments are `VOSYSMONITORV_IOC_CONTEXT` and `cycles_value`. `cycles_value` variable is set to 0 if the computation has to be from the driver to the first reached SBI function (the trap handler in the trap.S file), basically the **save context** phase of an **interrupt context switch** considering hardware latency. It is set to 1 if the computation target is from the first SBI function reached back to the driver (**restore context** and hardware latency). After the `ioctl` function the `cycles_value` will be the value of the interrupt context switch time in one case, the restore phase overhead in the other case, both as always in clock cycles.

`ECALL_test()` function is invoked 500 times with `cycles_value` 0, then the `cycles_value` variable which is returned is saved in a file on the board. Then again 500 times with `cycles_value` at 1, also here we save the variable value in a different file. The same operation is done with the `context_test()` function. We call these test functions 500 times to have a good number of samples and do considerations about them. All the code is provided in Appendix A.

4.1.2 VOSySmonitoRV driver

As said, in the user application we use the `ioctl` system call to send a command to the driver. Taking always as a reference the Fig.4.1 we can see that the VOSySmonitoRV driver runs in S-mode. The `ioctl` functions the vosysmonitorv-driver are:

- `VOSYSMONITORV_IOC_ECALL_INTERRUPT`: From the user space we have the `cycles_value` variable thanks to the `copy_from_user` function (see Section.2.3) that according with the desired computation, the corresponding function invocation is done. This variable is checked in the `sbi_ecall_vosysmonitor_ecall_interrupt()` function in the header file. This function gets back the context switch duration between the driver and the ECALL VOSySmonitoRV handler in

VOSySmonitoRV, basically the **Interrupt latency** in one case and the **Interrupt response** in the other. The `cycles_value` variable can be read and saved in the user application thanks to the `copy_to_user` function (see Section.2.3).

- `VOSYSMONITORV_IOC_CONTEXT`: Here we have the same situation of `VOSYSMONITORV_IOC_ECALL_INTERRUPT` but the purpose is to compute the interrupt context switch overhead of an interrupt triggered from the driver and trapped in the first reached function in VOSySmonitoRV, actually in the _trap_handler in the trap.S file (Fig.4.1)). The function that is invoked here is `sbi_ecall_vosysmonitor_context()` in vosysmonitorv.h file. This function gets back the context switch duration, so according to `cycles_value` value we can measure both the **save context** phase and the **restore context** phase.

The SBI ECALL functions in the vosysmonitorv.h file are:

- `SBI_ECALL`: This is basically an asm volatile function in which is done the `ecall`, the RISC-V privileged instruction [21]. As parameters, we can pass arguments in the registers from a0 to a5. The a6 and a7 registers are reserved because they are the registers dedicated to special macros useful to be recognized in VOSySmonitoRV as a type of extension (a6 function ID, a7 the extension ID). According to these parameters the ECALL will be handled and processed. To be more clear, in the following there is the C-code of the `SBI_ECALL` function, an open-source example is in the official Linux repository [10].

```
#define SBI_ECALL(__num, __a0, __a1, __a2, __num1)
        ({
          register unsigned long a0 asm("a0") =
          (unsigned long)(__a0);
          register unsigned long a1 asm("a1") =
          (unsigned long)(__a1);
          register unsigned long a2 asm("a2") =
          (unsigned long)(__a2);
          register unsigned long a6 asm("a6") =
          (unsigned long)(__num1);
          register unsigned long a7 asm("a7") =
          (unsigned long)(__num);
```

```
        asm volatile("ecall"
                     : "+r"(a0)
                     : "r"(a1), "r"(a2), "r"(a6), "r"(a7)
                     : "memory");
                     a0;
    })
```

- `sbi_ecall_vosysmonitor_ecall_interrupt()`: If the value variable is 0 we said that the computation is done let's say "forward" which practically means that the performance counter (in the cycle register) is read two times, one here in the driver and then in the ECALL VOSySmonitoRV handler in VOSySmonitoRV. Then the difference is done there and gives back the actual duration of the interrupt latency in clock cycles. On the opposite case if the value is 1 is "backward", the `rdcycle` instruction is done in the VOSySmonitoRV ECALL handler and we have the value of the performance counter in that position. So in the driver after the ECALL the `rdcycle` instruction is done again. The difference is returned to the application as the interrupt latency duration from VOSySmonitoRV to the driver in clock cycles. In both cases, we do an `SBI_ECALL` with parameter `SBI_EXT_VOSYSMONITORV_ECALL_INTERRUPT` that is the function ID in the switch case statement in the ECALL VOSySmonitoRV handler (that will be better explained in the next Section). In both cases a variable is passed through the arguments registers to VOSySmonitoRV, in the first case is containing the just taken cycles value, in the second is equal to 1. This variable is also checked in the ECALL VOSySmonitoRV handler.

- `sbi_ecall_vosysmonitor_context()`: Also here we have the same behaviour. If the variable value is 0 is "forward" (`SBI_EXT_VOSYSMONITORV_CONTEXT`) otherwise is "backward" (`SBI_EXT_VOSYSMONITORV_CONTEXT_BK`) with the same meaning as before. Here we want to measure the context switch duration. To do so it is necessary to read the cycle register in the assembly file trap.S in both "directions". It is important to notice that here we have two different function IDs to be checked in VOSySmonitoRV.

  – `SBI_EXT_VOSYSMONITORV_CONTEXT` when the computation is

"forward" and it is also passed the cycles variable so the difference can be done in VOSySmonitoRV.

- `SBI_EXT_VOSYSMONITORV_CONTEXT_BK` when the computation is "backward" we don't pass anything else, we want that the return value is the value of the cycle register just before the ECALL interrupt is ended and returns to the lower privileged mode (so in the trap.S assembly file).

The driver code is shown in Appendix B.

4.1.3 VOSySmonitoRV benchmark extension

Starting from the existent prototype of VOSySmonitoRV some modifications to support the benchmark was done. We are now talking about the right part of the Fig.4.1, VOSySmonitoRV running in M-mode.

Let's start first with the measurements of the **ECALL interrupt latency**. The ECALL was done in the driver, so we can follow the path from *(1)* to *(4)* to be recognized as a custom ECALL to be handled in the ECALL VOSySmonitoRV handler (sbi_ecall_vosysmonitorv.c file). Here, as already said, we have the function ID SBI_EXT_VOSYSMONITORV_ECALL_INTERRUPT used as a parameter (in a7 register) in the ECALL from the driver. In this switch case statement, we check if in the register a0, which is the flag from the driver, there is a 1 so we have to do the "backward" computation (the **Interrupt response** duration from VOSySmonitoRV to the driver). Otherwise, in the variable there is the value of the performance counter before the ECALL, the computation of the **Interrupt latency** can be done here. In this case, we simply do the `rdcycle` instruction, and then we compute the difference that is returned. The counter reading is done immediately, just before checking the variable, for optimization reasons.

For the **ECALL interrupt response** measurement we are in the "backward" case, path from *(6)* to *(9)*. The ECALL has the function ID value of SBI_EXT_VOSYSMONITORV_ECALL_INTERRUPT. After the check of the flag from the driver that is 1 in this case, the `rdcycle` instruction is done again, to avoid counting also the clock cycles of the "if" statement, and its value is saved in the a5 register. The return value is set to SBI_EXT_VOSYS-

MONITORV value to be recognized in the ecall_handler function. There, the return value is set in the a0 register. This "backward" computation must be managed because the return value of the handler functions are integers and are too small to contain the value of the cycle register read in the ECALL handler, and we need it to be returned to the driver to compute the difference. So we have placed this value in the a5 register, which is unsigned long, and in the ecall_handler function, we can save this value in the a0 register (to return it by the ECALL as explained before).

The measurements of the **Interrupt context switch** are done between the driver and the trap.S file. In the trap.S file we have the trap handler entry where the context is almost immediately saved. The `rdcycle` instruction is done just after the saving of the context and saves the value of the performance counter in the a5 register. Then there is the extension ID check, so only and only if the type of the extension ECALL is of type `EXT_VOSYSMONITORV` (by checking the a7 register) then the a5 register is pushed on the stack to be used in the VOSySmonitoRV ECALL handler later. This extension check is done after reading the cycle register as an optimization, to avoid counting the overhead due to the branch instructions. After that, there is the invocation of the `trap_handler` function that is in the trap.c file (Fig.4.1). Program execution continues until transition *(4)* where we have the switch case statement for the function ID. For this measurement, the ECALL was done with the function ID SBI_EXT_VOSYSMONITORV_CONTEXT. Here we have to compute the **Context switch** (Save context), so we have to take the value of the register in a5 and the value we had as an argument of the function at kernel space level (in a0) and make the difference between the two. The result is returned back. To compute the **Restore context** the ECALL from the driver has as function ID SBI_EXT_VOSYSMONITORV_CONTEXT_BK. The program execution goes from *(1)* to *(4)* (Fig.4.1). Here doesn't happen anything, this function ID is useful only to the function ID check in the trap.S file. Indeed the program flow goes back to the *(8)* transition. Here, as said, there is the check of the processed ECALL extension to see if was an EXT_VOSYSMONITORV, and also we check the function ID, because we want to read here the cycle register if and only if was a `SBI_EXT_VOSYSMONITORV_CONTEXT_BK` ECALL. Then we check again this value to decide if we can pop or not this value from the stack (it is

a "backward" computation we don't want to pop it because the register a5 would be overwritten). The Trap-Return instruction is done and we are back in the driver. The extension code is in Appendix C.

4.2 Results

The results of the measurements are collected in "txt" extension files. Those files are processed in a python script (see Appendix D) to compute the mean and the standard deviation of the results of the benchmark. The python script also represents the results in two graphs. As can be seen in Fig.4.2 we have the measurements of the **ECALL interrupt latency** with an average of 0.46 μs, the **ECALL interrupt response** means the time elapsed between the handling of the interrupt and the return to the ECALL caller, the average value is of 0.2 μs. These values are strongly software-dependent, due to the handler software structure and the software performance variations due to caches and interrupts. Therefore, we can compute an average and a standard deviation. The **interrupt handling overhead**, that is the sum of the interrupt latency and the interrupt response values, has the same behavior with an average equal to 0.66 μs.
As expected, the context switch measurement in Fig.4.3 is hardware-dependent, its average value is most of the time respected (0.071 μs) and it can be done the same consideration on the restore context phase (0.048 μs). Results are very promising for the realization of the shared core, ensuring that it is a further optimization rather than a loss, and also to be in such a system where these measurements are considered as an indicator of the time needed by VOSySmonitoRV to switch operating system or to full fill a request coming from an application (e.g., power management).

In the table 4.1, are summarized the interrupt and context switch measurements on a RISC-V processor compared with measurements on an x86 processor [13]. The context switch latency on a RISC-V processor is very low and suitable to support mixed-criticality applications concerning the already positive results in the x86 processor.

Those results are, as said, related to the interrupt operation of a very simple ECALL, through which the RTOS can ask to switch to the OS because

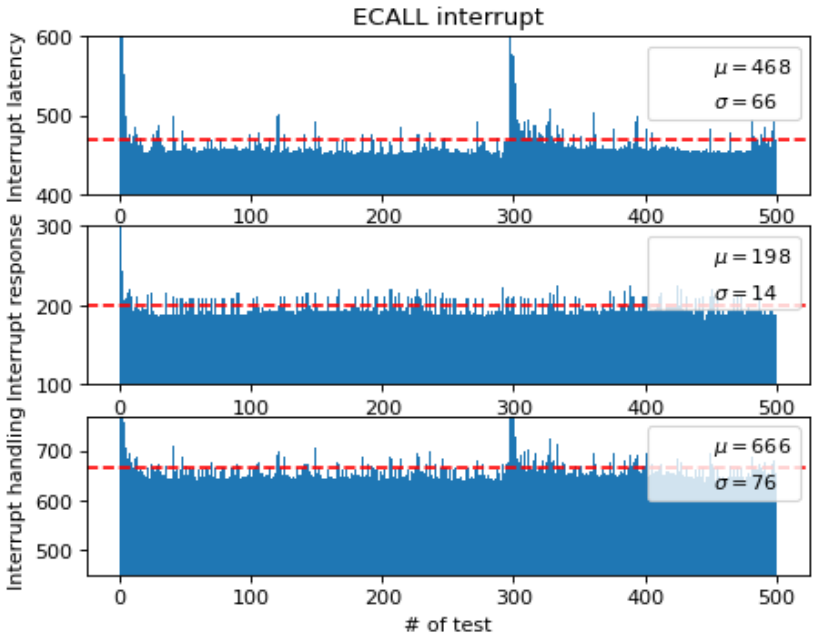

Figure 4.2: Interrupt latency measurements, μ is the average value and σ the standard deviation

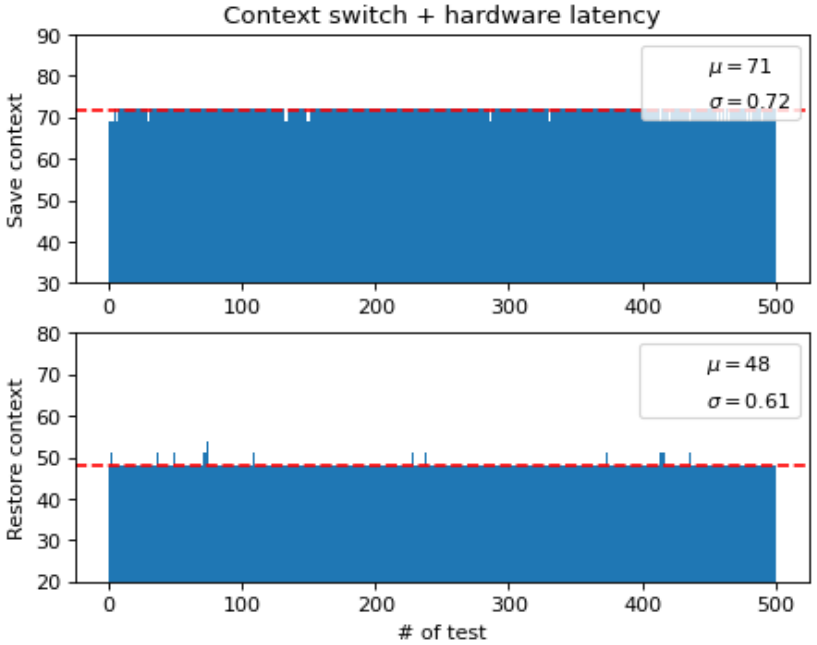

Figure 4.3: Context switch measurements, μ is the average value and σ the standard deviation

Table 4.1: The comparison of the benchmark results on the duration of the context switch with a x86 processor.

Variants	**Values in μS**	
	RISC-V	***Intel E3845***
Interrupt latency	0.46	-
Interrupt handling overhead	0.66	272
Context switch (save context)	0.071	1.39
Restore context	0.12	12.73

has the only idle task to schedule. The feasibility of this VOSySmonitoRV feature to co-execute a critical OS with a non-critical OS on a single hart depends on the latency of a context switch between them, the current partition registers shall be saved and the context of the other OS should be restored. Both partitions must be able to resume their execution where they have been preempted. Thanks to these measurements and the actual implementation of the context switch (see Section 3.4) between OSes we can evaluate that the latency is in the worst case (considering the hardware latency) near to 700 clock cycles (0.7 μS) that is still an acceptable latency for the purpose. To achieve this approximation, the number of instructions to do the context switch has been counted. These include the saving from the stack of the 32 general-purpose registers and the main CSRs that are required for the correct operation of the operating system. It is possible to compare these values with previous measurements on VOSySmonitor done on ARM architectures [11] in table 4.2.

Table 4.2: The comparison of the latency of the context switch between operating systems with different criticality

Processor and Platform	**Values in μS** ***context switch between OSes***
RISC-V SiFive Freedom U540	0.7
A53 Juno	1.02
A57 Juno	1.4
A53 R-CarH3	0.6
A57 R-CarH3	0.96

Chapter 5

Conclusions

Nowadays mixed-criticality applications are used in some fields such as automotive and industry, usually, they reached different critical environments through virtualization. A valid alternative is to use one firmware with high-security features that run directly on the hardware, in such a way that it is not necessary to have a virtualization extension to run securely two or more operating systems on the same machine. This can be useful with a RISC-V board, such as SiFive HiFive Unleashed. RISC-V has in its frozen ISA the hypervisor extension for virtualization but it has not already a hardware implementation. Anyway, VOSySmonitoRV is designed to not use it as an architectural choice, acting at the higher privileged level. In this thesis work, the starting prototype was enhanced with a first prototype realization that includes the RTOS running on one hart that it is securely booted before the GPOS. Moreover, PMP configurations of each hart are set before actually booting the OSes, so a total and secure isolation is achieved. The RTOS is now working and executing in S-mode. To further optimize the first prototype, a second prototype with a shared hart has been developed but it is not fully working. The interrupt latency and context switch benchmark results are very promising for VOSySmonitoRV applications in mixed-criticality environments with an **ECALL interrupt latency** value of 0.46 μS and a **Interrupt context switch** equal to 0.071 μS . Interrupt latency and context switch are considered key operations because these are response time performance meters, very important in mixed-criticality systems. A paper on VOSySmonitoRV was submitted to an IEEE embedded conference.

5.1 Future work

This project is under development and improvement by the company, which includes making the context switch between operating systems work correctly and efficiently. Furthermore, porting these prototypes to other boards such as the Andes AE350 and the new SiFive HiFive Unmatched have been considered as future work. As mentioned, integration with SiFive WorldGuard technology would be very interesting as there is already the use of a variable that determines the "world" that is running on the hart.

Appendix A

VOSySmonitoRV benchmark application code

This is the main of the VOSySmonitoRV application in the user_app.c file (Fig.4.1). This application has to be run on the board on Linux v5.8, after the driver loading.

```
/* Copyright (C) 2020 - Virtual Open Systems SAS
*  Author Flavia Caforio <f.caforio@virtualopensystems.com> */
int main()
{
   char* devfilename = devname;
   FILE *fp  = fopen("benc-res-int-latency.txt","w+");
   FILE *fp1 = fopen("benc-res-cont.txt","w+");
   FILE *fp2 = fopen("benc-res-int-response.txt","w+");
   FILE *fp3 = fopen("benc-res-cont-bk.txt","w+");

   if(fp == NULL || fp1 == NULL || fp2 == NULL || fp3 == NULL)
   {
     printf("Error file at user app level\n");
     exit(1);
   }
   int i = 0;
   int ret = 0;
   g_devFile = open(devfilename, O_RDWR | O_NONBLOCK);

   /* We call these test functions a certain number of times
    * to do statistical consideration about the duration
    * of the context switch */
```

```
 /* Interrupt latency measurement */
for(i = 0; i < 500 ; i++)
{
   cycles_value = 0;
   ret = ECALL_test();
   /* Values are written on a file on the board */
   fprintf(fp,"%ld\n",cycles_value);
}
fclose(fp);
sleep(1);

 /* Save context measurement */
for(i = 0; i < 500 ; i++)
{
   cycles_value = 0;
   ret = context_test();
   /* Values are written on a file on the board */
   fprintf(fp1,"%ld\n",cycles_value);

}
sleep(1);
fclose(fp1);

 /* Interrupt response measurement */
for(i = 0; i < 500 ; i++)
{
   cycles_value = 1;
   ret = ECALL_test();
   /* Values are written on a file on the board */
   fprintf(fp2,"%ld\n",cycles_value);
}
fclose(fp2);
sleep(1);

 /* Restore context measurement */
for(i = 0; i < 500 ; i++)
{
   cycles_value = 1;
   ret = context_test();
   /* Values are written on a file on the board */
   fprintf(fp3,"%ld\n",cycles_value);
```

```
    }
    fclose(fp3);
    sleep(1);

    close(g_devFile);

    return 0;
}

/* ECALL_test() function returns the interrupt latency and the
                    iterrupt response of an ECALL interrupt */
int ECALL_test()
{
    long ret;
    u_int64_t ret_value = 0;

    ret = ioctl(g_devFile, VOSYSMONITORV_IOC_ECALL_INTERRUPT,
                    &cycles_value);
    return ret;
 }

/* context_test() function returns the context switch save and
                    restore phase duration of an ECALL
                    interrupt */
int context_test()
{
    long ret;
    u_int64_t ret_value = 0;

    ret = ioctl(g_devFile, VOSYSMONITORV_IOC_CONTEXT,
                    &cycles_value);
    return ret;
 }
```

Appendix B

VOSySmonitoRV benchmark driver code

This is the implementation of the functions explained in the benchmark chapter, in the driver description 4.1.2. Those functions are i the file vosysmonitorv.h (Fig.4.1). This driver has to be loaded on the Linux kernel v5.8 (so compiled for this version) on the board.
The function implementation for the ECALL interrupt measurements:

```
/* Copyright (C) 2020 - Virtual Open Systems SAS
*  Author Flavia Caforio <f.caforio@virtualopensystems.com> */
uint64_t sbi_ecall_vosysmonitor_ecall_interrupt(uint64_t value)
{
  uint64_t ret = 0;
  unsigned long cycles_back = 0;
  uint64_t cycles = 0;
  if(value ==1)
  {
      /* Here we read the cycle register with the *rdcycle*
      * instruction just after the ecall.
      * The *value* is 1 so the measurement is done
      * from the ECALL VOSySmonitoRV handler
      * to the driver. The *rdcycle* instruction is done in the
      * VOSySmonitoRV handler and then the
      * value is returned back in cycles_back.
      * The final computation is done here. */
      cycles_back = SBI_ECALL(SBI_EXT_VOSYSMONITORV,
                    SBI_EXT_VOSYSMONITORV_ECALL_INTERRUPT,
```

```
                    value);
    asm volatile ("rdcycle␣%0" : "=r" (ret));
    ret = ret - cycles_back;

  } else
  {
      /* Here we read the cycle register with the *rdcycle*
      * instrunction just before the ecall.
      * The *value* is 0 so the context switch duration is
      * from the driver to VOSySmonitoRV.
      * The *rdcycle* instruction is done in the
      * VOSySmonitoRV handler and then the
      * computation is done and returned back */
    asm volatile ("rdcycle␣%0" : "=r" (cycles));
    ret=SBI_ECALL(SBI_EXT_VOSYSMONITORV,
                    SBI_EXT_VOSYSMONITORV_ECALL_INTERRUPT,
                    cycles);
  }
  return ret;
}
```

The function implementation for the context switch measurements:

```
/* Copyright (C) 2020 - Virtual Open Systems SAS
*  Author Flavia Caforio <f.caforio@virtualopensystems.com> */
uint64_t sbi_ecall_vosysmonitor_context(uint64_t value)
{
    uint64_t ret = 0;
  unsigned long cycles_back = 0;
  uint64_t cycles = 0;

        if(value == 1)
  {
        /* If value is 1 means that we want to measure the
        * restore context overhead from
        * VOSySmonitoRV to the driver. So we don't pass
        * anything, we want that the return value
        * is the value of the *cycle* register just before
        * return back from the ECALL.
        * Then again here, and we compute the difference
        * to have the actual duration. */
        cycles_back = SBI_ECALL_1(SBI_EXT_VOSYSMONITORV,
                    SBI_EXT_VOSYSMONITORV_CONTEXT_BK);
```

```
    asm volatile ("rdcycle␣%0" : "=r" (ret));
    ret = ret - cycles_back;

  } else
  {
      /* Here we read the cycle register with the *rdcycle*
      * instruction just before the ECALL
      * and then in the first file reached in VOSySmonitoRV
      * the *rdcycle* instruction is done.
      * The difference is computed by the ECALL
      * VOSySmonitoRV handler and returned back */
        asm volatile ("rdcycle␣%0" : "=r" (cycles));
        ret = SBI_ECALL(SBI_EXT_VOSYSMONITORV,
                    SBI_EXT_VOSYSMONITORV_CONTEXT, cycles);
        }
        return ret;
}
```

Appendix C

VOSySmonitoRV benchmark extension code

This is the VOSySmonitoRV ECALL handler in the ecall_vosysmoitorv.c (Fig.4.1).

```
/* Copyright (C) 2020 - Virtual Open Systems SAS
*  Author Flavia Caforio <f.caforio@virtualopensystems.com> */
static int sbi_ecall_vosysmonitorv_handler(unsigned long
                        extensionid, unsigned long funcionid,
                        unsigned long *regs)
{
  int ret = 0;
    static unsigned long start_cycles = 0;
  static unsigned long cycles = 0;
  static uint64_t vosys_var=0;

  switch (funcid) {
    case SBI_EXT_VOSYSMONITORV_ECALL_INTERRUPT:
      /* In the register a0 we have the flag from the
       * driver, if
       * is 1 the computation is backward (from
       * VOSySmonitoRV to the driver)
       * otherwise is 0 and means that the computation
       * is from kernel space to m-mode. */
      asm volatile ("rdcycle %0" : "=r" (cycles));
      start_cycles = regs[0]; /* a0 register, the argument of
                               * the SBI_ECALL in the driver */
      if (start_cycles == 1)
```

```
      {
      /* We read again to avoid to count also the clock
       * cycles of the if statement.
       * The return value is set to SBI_EXT_VOSYSMONITORV
       * because when the return
       * value of the ecall is set in the sbi_handler
       * function we want to check
       * if it is from this specific function.
       * We do that because the return value
       * of the handlers are integers and are too small
       * for the value of the *cycle*
       * register read here. So we put this value in the
       * regs[5] (a5 register) that is unsigned
       * long and then in the sbi_handler function we put
       * this value in a0 (to return it
       * back after the ECALL). */
          asm volatile ("rdcycle␣%0" : "=r" (cycles));
          regs[5] = cycles;
          ret = SBI_EXT_VOSYSMONITORV;
      }else
      {
      /* The computation is returned back to the driver */
          ret = cycles - start_cycles;
      }
      break;
    case SBI_EXT_VOSYSMONITORV_CONTEXT:
      /* The *rdcycle* instruction is done in trap.S
       * before the calling of the sbi_handler function.
       * the value is passed in the a5 register, and in a0 we
       * have the value from the driver. We compute the
                    difference here
       * and we return back the computation of the minimum
                    context switch */
                       cycles = (unsigned long) regs[5];
                       start_cycles = regs[0];
                       ret = cycles - start_cycles;
                       break;
    case SBI_EXT_VOSYSMONITORV_CONTEXT_BK:
      /* The *rdcycle* instruction is done in trap.S
       * after the return of the sbi_handler function.
       * There, if and only if this SBI function was
       * invoked, the value
```

```
       * of the *cycle* register is returned to the driver */
                        ret = 0;
                        break;
    default:
      ret = SBI_ECALLNOTSUPP;
          };

  return ret;
}
```

Appendix D

Benchmark results elaboration

This is a python script that was created to elaborate the benchmark data results.

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import numpy as np



x = np.linspace(1, 500, 500, endpoint = True)
y1 = np.loadtxt('benc-res-int-latency.txt', delimiter='\n',
                    unpack=True)

mu1 = np.mean(y1)
sigma1 = np.std(y1)

fig,(ay1, ay2,ay3) = plt.subplots(3,1)
ay1.bar(x,y1,width=1)
ay1.set(ylim=[400,600],ylabel='Interrupt␣latency',title='ECALL␣
                    interrupt')
ay1.axhline(mu1, ls='--', color='r')
extra = Rectangle((0,0),1, 1, fc="w", fill=False,
                    edgecolor='none', linewidth=0)
ay1.legend([extra,extra],["$\mu␣=␣468$","$\sigma␣=␣
                    66$"],loc="upper␣right")

y2 = np.loadtxt('benc-res-int-response.txt', delimiter='\n',
                    unpack=True)
```

```
mu2 = np.mean(y2)
sigma2 = np.std(y2)

ay2.bar(x,y2,width=1)
ay2.set(ylim=[100,300],ylabel='Interrupt␣response')
ay2.axhline(mu2, ls='--', color='r')
extra = Rectangle((0,0),1, 1, fc="w", fill=False,
                    edgecolor='none', linewidth=0)
ay2.legend([extra,extra],["$\mu␣=␣198$","$\sigma␣=␣
                    14$"],loc="upper␣right")

y3 = np.loadtxt('handling_overhead.txt', delimiter='\n',
                    unpack=True)

mu3 = np.mean(y3)
sigma3 = np.std(y3)

ay3.bar(x,y3,width=1)
ay3.set(ylim=[450,770],ylabel='Interrupt␣handling',xlabel='#␣of␣
                    test')
ay3.axhline(mu3, ls='--', color='r')
extra = Rectangle((0,0),1, 1, fc="w", fill=False,
                    edgecolor='none', linewidth=0)
ay3.legend([extra,extra],["$\mu␣=␣666$","$\sigma␣=␣
                    76$"],loc="upper␣right")
plt.show()

fig.savefig('figures/ECALL_interrupt.png',transparent=False,
                    dpi=80, bbox\_inches="tight")

y4 = np.loadtxt('benc-res-cont.txt', delimiter='\n',
                    unpack=True)

mu4 = np.mean(y4)
sigma4 = np.std(y4)

fig2, (ay4,ay5) = plt.subplots(2,1)
ay4.bar(x,y4,width=1)
ay4.set(ylim=[30,90],ylabel='Save␣context',title='Context␣
                    switch␣+␣hardware␣latency')
ay4.axhline(mu4, ls='--', color='r')
```

```
extra = Rectangle((0,0),1, 1, fc="w", fill=False,
                      edgecolor='none', linewidth=0)
ay4.legend([extra,extra],["$\mu␣=␣71$","$\sigma␣=␣
                      0.72$"],loc="upper␣right")

y5 = np.loadtxt('benc-res-cont-bk.txt', delimiter='\n',
                      unpack=True)

mu5 = np.mean(y5)
sigma5 = np.std(y5)

ay5.bar(x,y5,width=1)
ay5.set(ylim=[20,80],ylabel='Restore␣context')
ay5.axhline(mu5, ls='--', color='r')
extra = Rectangle((0,0),1, 1, fc="w", fill=False,
                      edgecolor='none', linewidth=0)
ay5.legend([extra,extra],["$\mu␣=␣48$","$\sigma␣=␣
                      0.61$"],loc="upper␣right")

plt.show()

fig2.savefig('figures/contextswitch.png',transparent=False,
                      dpi=80, bbox_inches="tight")
```

Bibliography

[1] Amazon Web Services Inc. or its affiliates. *The FreeRTOS™ Kernel*. URL: https://www.freertos.org/RTOS.html.

[2] Amazon Web Services Inc. or its affiliates. *Using FreeRTOS on RISC-V Microcontrollers*. URL: https://www.freertos.org/Using-FreeRTOS-on-RISC-V.html.

[3] Thomas Bourgeat et al. "Mi6: Secure enclaves in a speculative out-of-order processor". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 42–56.

[4] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. ISBN: 0596005903.

[5] Victor Costan, Ilia Lebedev, and Srinivas Devadas. "Sanctum: Minimal hardware extensions for strong software isolation". In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 857–874.

[6] Cesare Garlati and Sandro Pinto. "A Clean Slate Approach to Linux Security RISC-V Enclaves". In: ().

[7] Virtual Open System Inc. *VOSySmonitoRV: a mixed criticality virtualization solution for RISC-V*. 2020. URL: http://www.virtualopensystems.com/en/solutions/demos/vosysmonitorv-risc-v-demo/.

[8] Ilia Lebedev et al. "Sanctorum: A lightweight security monitor for secure enclaves". In: *arXiv preprint arXiv:1812.10605* (2018).

[9] Dayeol Lee et al. "Keystone: An open framework for architecting trusted execution environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[10] *Linux Kernel Source Tree*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/riscv/kernel/sbi.c.

[11] Pierre Lucas et al. "Vosysmonitor, a low latency monitor layer for mixed-criticality systems on armv8-a". In: *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[12] José Martins et al. "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems". In: *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020.

[13] Nikos Mouzakitis et al. "x86 System Management Mode (SMM) Evaluation for Mixed Critical Systems". In: *International Conference on Applications in Electronics Pervading Industry, Environment and Society*. Springer. 2020, pp. 164–170.

[14] Matthew Portnoy. *Virtualization essentials*. Vol. 19. John Wiley & Sons, 2012.

[15] David Schrammel et al. "Donky: Domain keys–efficient in-process isolation for RISC-V and x86". In: *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020, pp. 1677–1694.

[16] FEDERICO SIERRA-ARRIAGA, RODRIGO BRANCO, and BEN LEE. "Security Issues and Challenges for Virtualization Technologies". In: (2020).

[17] *SiFive FU540-C000 Manual*. English. Version Version v1p0. SiFive, Inc.

[18] *SiFive U54 Manual*. English. Version Version 20G1.03.00. SiFive, Inc.

[19] *SiFive WorldGuard White Paper*. Tech. rep. Version Version 1.2. SiFive Inc., 2020. URL: https://sifive.cdn.prismic.io/sifive/aa27fffb-cf24-4077-8103-682f26141b69_WorldGuard_White_Paper_v1.2.pdf.

[20] José Simó et al. "The Role of Mixed Criticality Technology in Industry 4.0". In: *Electronics* 10.3 (2021), p. 226.

[21] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10*. 2017.

[22] Andrew Waterman and Krste Asanović. *"The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121*. 2019.