POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Performance-Aware Scheduling in Real Time Operating System

Supervisors

Prof. ALESSANDRO SAVINO

Prof. MAURIZIO REBAUDENGO

Candidate

LUCA COSTA

APRIL 2024

Acknowledgements

Alle persone entrate nella mia vita durante questo percorso.
Alla mia famiglia.
A voi...GRAZIE.

‘"La felicità è vera solo se condivisa" - Chris McCandless

Abstract

One of the main job of an operating system is to schedule tasks' activities. Tasks scheduling is a software routine which assign the execution of a task to a CPU core since each of them can execute one at any given time. The policy behind the scheduling algorithm is critical because it influences the overall performance of the operating system.
Modern microprocessors have built-in special-purpose registers in which the counts of hardware-related activities (number of instructions executed, number of memory loads and stores, number of branches taken, etc.) are saved. These special-purpose registers are called Hardware Performance Counters (HPCs).
This work aims to use the information stored in these registers trying to improve the performance of a real time operative system integrating it in the default scheduling algorithm.

Table of Contents

List of Tables

List of Figures

Acronyms

IoT
: Internet of Things

HPC
: Hardware Performance Counters

PULP
: Parallel Ultra Low Power

OS
: Operative System

RTOS
: Real Time Operating System

ISA
: Instruction Set Architecture

PCMR
: Performance Counter Mode Register

PCER
: Performance Counter Event Register

PCCR
: Performance Counter Counter Register

TCDM
: Tightly-Coupled Data-Memory

CPU	Central Processing Unit
API	Application Programming Interface
SDK	Software Development Kit
HAL	Hardware Abstraction Level
TCB	Task Control Block

Chapter 1

Introduction

The spread of embedded systems has a big impact on different industry sectors and in the way we interact with the world. Furthermore, the expansion of IoT (Internet of Things) has accelerated the adoption of these systems. Embedded systems can be found everywhere, driving a wide array of applications ranging from consumer electronics and automotive systems to industrial machinery and medical devices. Their popularity allowed successful research in microcontroller technology, miniaturization and connectivity, enabling the development of numerous embedded solutions.
RISC-V Instruction Set Architecture had a great impact for the development of embedded solutions due to its open source nature and to its highly customizability and flexibility. These features made this ISA very popular for the development of custom processor targeting embedded systems, thus it has been adopted in various industry and academic researches.
Among all the project adopting RISC-V ISA, PULP Project is one who achieved a great reputation worldwide. Developed from the collaboration of the Integrated Systems Laboratory (IIS) of ETH Zürich and Energy-efficient Embedded Systems (EEES) group of the University of Bologna, it is a platform targeting the IoT devices meeting the power and the timer constraints of a real-time embedded system.
GVSoC is an open-source simulator, specifically targeting the PULP architecture, born in order to facilitate the development of custom applications and analysis over new hardware designs. This event-driven simulator allowed the developers to work on the platform without the need to physically possess the board, simulating it with a small percentage of error.
Power efficiency, real-time performance, and security is a must in systems target of these projects. For these reasons the OS kernel running on the embedded system is a crucial point. The kind of OS specifically designed for embedded system, denoted as RTOS, deals with the real-time computations of processes. FreeRTOS is an

operative system belonging to this family. The part of the real-time OS which should guarantee real-time performance is the kernel scheduler, the one processing events and data with specific time requirements. The scheduler is in charge of taking the decision on the next task/process to run when a switch context occurs. Even if the schedulers of the RTOS put a lot of focus on the constraints about the time window in which the events must be processed, modern architectures and the new hardware design and technology can speed up the execution of tasks.
Hardware Performance Counters (HPCs) are special registers, usually built in the microprocessor, storing the occurrences of specific hardware events. Thanks to performance counters a developer can profile several aspects like the performance of the OS itself or the performance and the debugging of the developed custom software application.
This work presents a new scheduling algorithm policy implemented on FreeRTOS running on a PULP platform, PULPissimo. The platform has been simulated with GvSoc. Chapter 2 gives a detailed explanation of the environment over which FreeRTOS run, covering the RISC-V ISA, GVSoc and the PULP Platform Project. Chapter 3 gives an inside to the features of FreeRTOS, ranging from implementation of the task to the scheduler and the associated structures. Chapter 4 describes the implementation of the new scheduler policy. Chapter 5 offers the results of the analysis performed with the new developed algorithm. Finally, Chapter 6 explains the possible future works to be adopted in order to improve the actual solution.

Chapter 2

Environment

2.1 RISCV Architecture

The RISC-V ISA has been designed to have an important feature: being highly flexible and customizable. It offers support for an extensive range of data type and memory architectures and base instructions can be extended with custom ones enabling a broad spectrum of tailored implementations to address specific application needs [1].
The critical advantage of RISC-V architecture is its open-source essence, allowing to use, modify or distribute the ISA without having any restrictions and without being obliged to pay any charges for licensing [1]. All these features have risen its popularity, enjoying widespread adoption across academic and industrial domains alike.
According to [2] the RISC-V ISA is defined as a base integer ISA voluntarily limited to a minimal set of instructions, offering a sufficiently comprehensive target for compilers, linkers and operating system. The base ISA provides a software tool chain "template" upon which more tailored processor ISAs can be developed. The RISC-V Foundation have developed two versions of the base ISA : the RV32 and RV64. They differ in the size of the user address space respectively 32-bit and 64-bit. The base integer ISA can be enhanced with multiple instruction-set extensions, but can not be altered.
As previously stated, due to its high customizability, a series of standard extensions has been defined in order to support and increase software development around this environment. These standard extensions as the non-standard ones bring additional functionality to the base ISA. A name has been conventionally assigned to these extensions which is used as suffix to RV32 or RV64.
Below is reported a small description for each standard extension used in this work.

- **"I" extension**
  It is mandatory for all RISC-V implementation. It expands the instructions supporting operations over integer data (e.g. memory load and store).
- **"M" extension**
  It brings new instructions which allow multiplication and division on integer values.
- **"F" extension**
  It allows performing computation with single-precision floating-point data.
- **"C" extension**
  It stands for Compression. Thanks to this extension some of the 32-bit RISC-V base instructions falling under particular conditions are shrunk to 16-bit using a compression scheme.

2.1.1 RI5CY

It is a RISC-V processor core with 4-stage core with the optional 32-bit FPU implemented by the F extension and the pulp custom extension.
In Figure 2.1 is shown the block diagram of the RI5CY core.

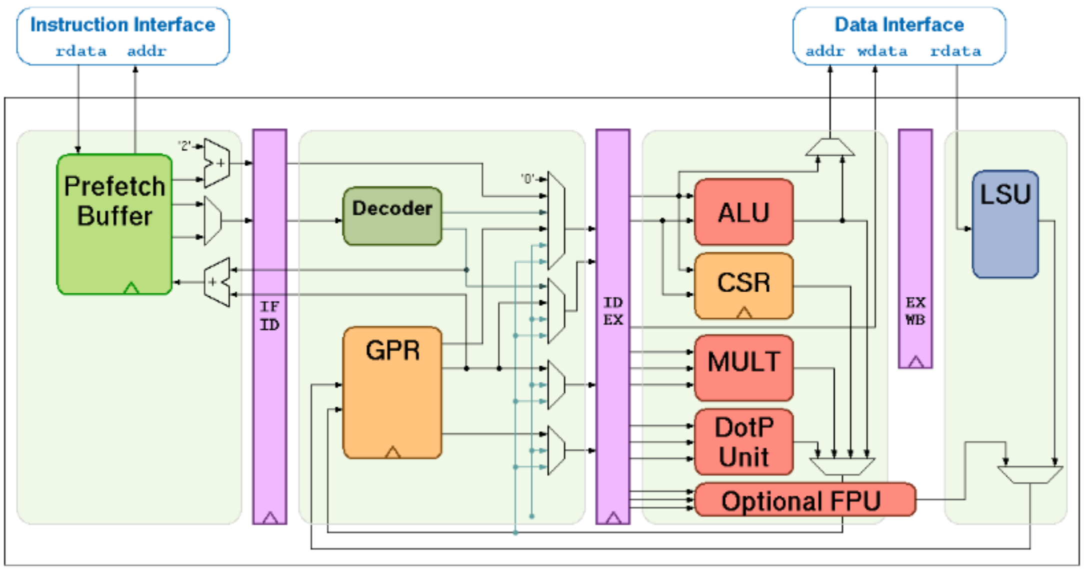

Figure 2.1: Block Diagram of RI5CY RISC-V Core [3]

2.2 Hardware Performance Counters RI5CY

2.2.1 HPCs basic notion

Hardware Performance Counters (HPCs) are special-purpose registers built-in in recent microprocessors where are stored the counts of specific hardware-related activities [4]. They belong in the category of the Control and Status Register (CSR).
They have been designed to track hardware events and performance measures at the micro-architectural level during program execution providing deep understanding into the behaviour and efficiency of software applications.
Hardware Performance Counters can be configured by software allowing developers to monitor events of their interest. Their utility crosses different domains including software performance analysis and system tuning.

2.2.2 HPCs implementation and access

In the RI5CY processor core has 24 performance counters. Each of them is 32-bit wide, an implementation not compliant with the RISC-V standard. The performance counters are accessed through two pseudo-instruction provided by the RISC-V Instruction Set Architecture.
The csrr pseudo-instruction is used to perform a read.
The csrw pseudo-instruction is used to perform a write.

2.2.3 Performance Counter Mode Register (PCMR)

The Performance Counter Mode Register enable/disable the performance counters.
A detailed description is given in Table 2.1
Its memory address is 0xCC1.
It can be reset writing the reset value 0x0003.

Bit Position	**Access Mode**	**Description**
0	R/W	If this bit is set to 0 the performance counter will be activated. Complementary they will be deactivated setting it to 1. After reset, this bit is set.
1	R/W	If this bit is set to 1, saturating arithmetic is employed in the performance counter operations. Following a reset, this bit is enabled.

Table 2.1: Performance Counter Mode Register setting

2.2.4 Performance Counter Event Register (PCER)

The Performance Counter Event Register allows to enable/disable the counting of an event associated to a specific performance counter. When disabled the performance counter will stop counting the occurrences of the event and its value will not change anymore.
Its memory address is 0xCC0.
It can be reset writing the reset value 0x0000.
A detailed description is given in Table 2.2

2.2.5 Performance Counter Counter Register (PCCR)

The performance counter counter register are effectively the ones where is stored the count of the correspondent event. A detailed description is given in Table 2.3

Bit Position	**Access Mode**	**Associated HPC name**
0	R/W	CYCLES
1	R/W	INSTR
2	R/W	LD_STALL
3	R/W	JMP_STALL
4	R/W	IMISS
5	R/W	LD
6	R/W	ST
7	R/W	JUMP
8	R/W	BRANCH
9	R/W	BRANCH_TAKEN
10	R/W	RVC
11	R/W	LD_EXT
12	R/W	ST_EXT
13	R/W	LD_EXT_CYC
14	R/W	ST_EXT_CYC
15	R/W	TCDM_CONT
16	R/W	CSR_HAZARD

Table 2.2: Performance Counter Event Register setting

Address	**Name**	**Description**
0x780	CYCLES	Count the number of cycles the core was running
0x781	INSTR	Count the number of instructions executed
0x782	LD_STALL	Number of load data hazards
0x783	JMP_STALL	Number of jump register hazards
0x784	IMISS	Cycles waiting for instruction fetches. i.e. the number of instructions wasted due to non-ideal caches
0x785	LD	Number of memory loads executed. Misaligned accesses are counted twice
0x786	ST	Number of memory stores executed. Misaligned accesses are counted twice
0x787	JUMP	Number of jump instructions seen, i.e. j, jr, jal, jalr
0x788	BRANCH	Number of branch instructions seen, i.e. bf, bnf
0x789	BRANCH_TAKEN	Number of taken branch instructions seen, i.e. bf, bnf
0x78A	RVC	Number of compressed instructions
0x78B	LD_EXT	Number of memory loads to EXT executed. Misaligned accesses are counted twice. Every non-TCDM access is considered external
0x78C	ST_EXT	Number of memory stores to EXT executed. Misaligned accesses are counted twice. Every non-TCDM access is considered external
0x78D	LD_EXT_CYC	Cycles used for memory loads to EXT. Every non-TCDM access is considered external
0x78E	ST_EXT_CYC	Cycles used for memory stores to EXT. Every non-TCDM access is considered external
0x78F	TCDM_CONT	Cycles wasted due to TCDM/log-interconnect contention
0x790	CSR_HAZARD	Cycles wasted due to CSR access
0x79F	ALL	A write to this register will set all counters to the supplied value

Table 2.3: Performance Counter Counter Registers

2.3 GVSoC

GVSoC is an open-source simulator specifically targeting the PULP architecture. It is an event-driven simulator capable to replicate a PULP platform by modeling the micro-architecture and putting up the common components like cores, memory, cluster, peripherals, interconnect and TCDM [5]. In GVSoC, the simulation system adopts a component-based methodology to model the system. Interactions among components occur exclusively through clearly defined connections between a master port of one component and a slave port of another. Each port delineates a collection of methods that the component can invoke on the other component [6].
Due to its remarkable performance compared to others simulator present in the actual scenario, it is on of the best choice for developers which have to test new functionalities and the related performances brought in the PULP environment. It has been found to have less than 10% error with respect to the physical SoC and to be 2500 times faster than others simulators [5].
GVSoC is able to simulate the main hardware components like CPU, Memory(TDCM/L2), DMA, Interconnect, I-Cache, Accelerators and I/O. Its functionality takes place thanks to its three major components:

- C++ files modelling the behaviour of each component.
- JSON configuration files to modify as needed the parameters of the architecture.
- Python scripts also denoted as generators. Instantiate and assemble each component of the system to be simulated. Each Python components has a set of properties which are passed to the C++ model through JSON file.

The compilation of GVSoC produces a shared library named libpulp.so which contains the engine code and allows loading code of other components through other shared library. The functionalities of GVSoC are exposed through an API written in C called GVSoC API. GVSoC can be executed with several kinds of tools. One of the tool is the gvsoc_launcher which is the default one. The figure 2.2 report the structure of the GVSoC simulator.

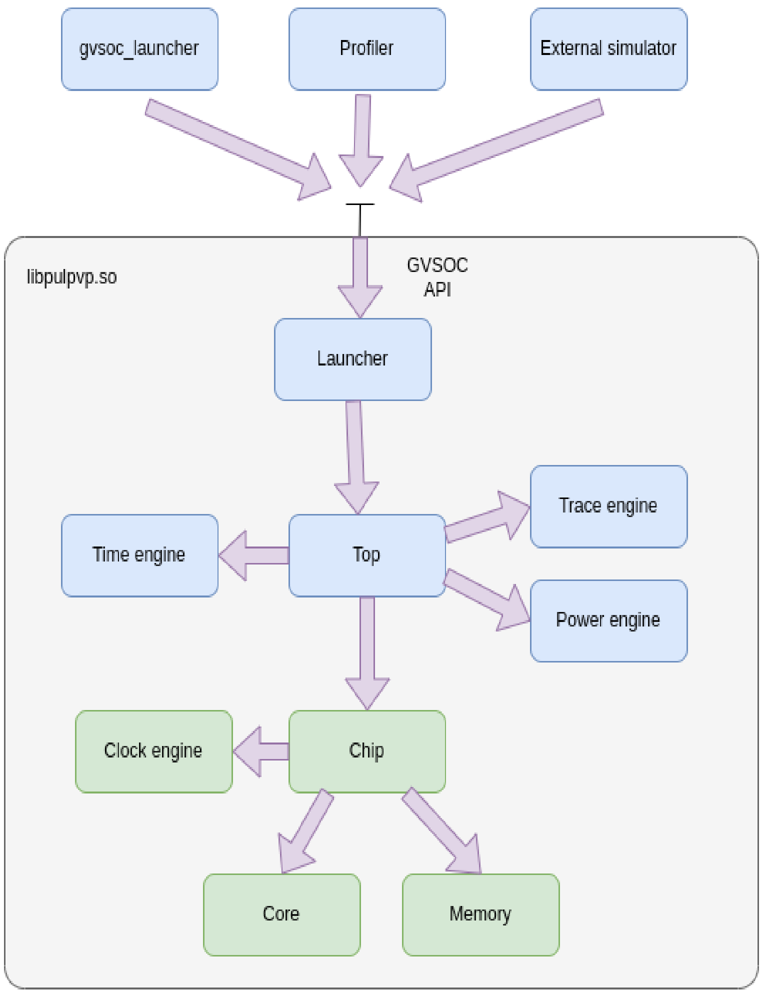

Figure 2.2: GVSoC structure [6]

2.4 PULP Project

PULP, acronym of Parallel Ultra Low Power, is a platform born to meet the computational requirements of IoT applications domain while keeping its power consumption at the lowest [7].
The PULP team who brought to life this project since the beginning has designed it in order to be open-source and to make customization and flexibility two main key points. These principles allowed PULP to be a suitable platform for research and development in the academic and industry domain.
PULP platform is built upon the open-source RISC-V instruction-set architecture and unleashed two versions: 32-bit and 64-bit. The platform covers several RISC-V processors: CV32E40P (RI5CY), IBEX (Zero-riscy), Micro-riscy, CVA6 (Ariane)

and Snitch.
Peripherals are crucial part when building a system especially in IoT environment. The team developed customized accelerators, interconnect solutions (i.e. logarithmic interconnect, APB-peripheral BUS, AXI4-interconnect) DMA engines and various peripherals like GPIO,SPI, I2S, JTAG and so on.
The most basic PULP-based systems are microcontrollers which can be configured to use any of the 32-bit RISC-V cores developed (RI5CY, Zero-riscy, Micro-riscy). PULP project includes two single-core microcontroller units:

PULPino : a minimal single-core RISC-V SoC. An overview is given in figure **??**.

PULPissimo : an improved version of the PULPino microcontroller unit. The SoC includes a logarithmic interconnect, a built on chip μDMA and optional accelerators. It is the one adopted for the work of this thesis. A detailed description is given in 2.4.1

2.4.1 PULPissimo

PULPissimo is a microcontroller implemented in the newest PULP chips [8]. A of the block diagram is given in Figure 2.3. It is a single core platform including:

Core
includes either the RI5CY core or the Ibex core, serving as the primary processing unit. The work of this thesis is based on RI5CY.

Autonomous Input/Output Subsystem
This subsystem handles input and output operations independently, reducing the burden on the main core and improving overall system efficiency.

Memory Subsystem
PULPissimo features a new memory subsystem, enhancing memory management and access efficiency.

Hardware Processing Engines
These engines provide dedicated hardware acceleration for specific tasks, improving performance for targeted workloads.

Simple Interrupt Controller
The architecture incorporates a new interrupt controller, simplifying interrupt handling and improving system responsiveness.

Peripherals
PULPissimo introduces new peripherals to support various input/output operations and connectivity requirements.

SDK
A new SDK is provided to facilitate software development for the PULPissimo architecture, offering tools, libraries, and documentation to streamline application development.

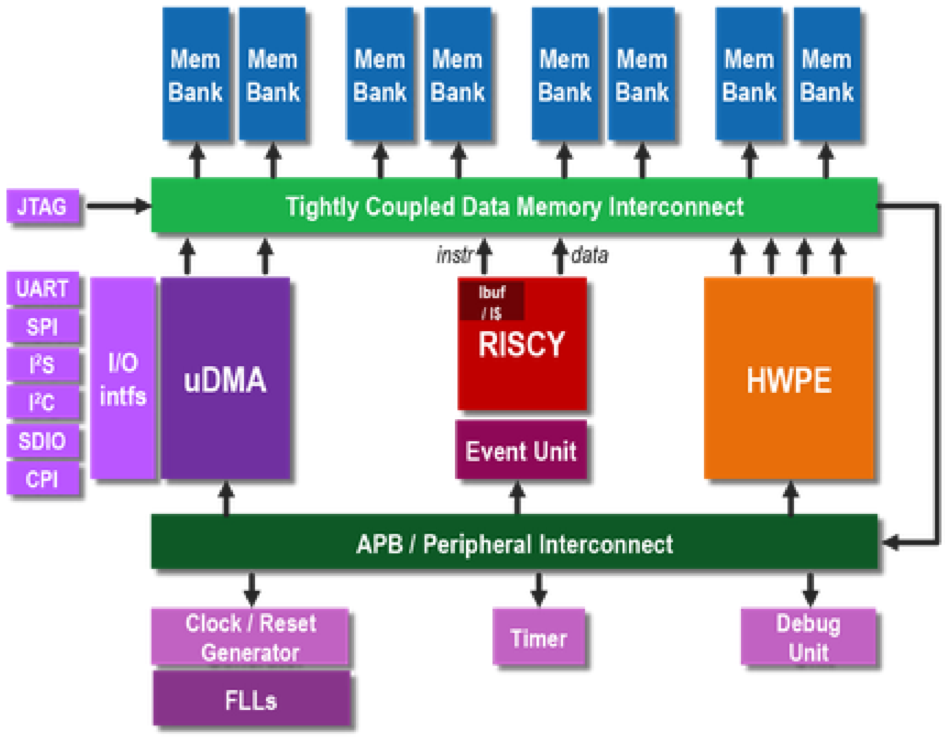

Figure 2.3: PULPissimo block diagram from [8]

PULPissimo supports Input/Output (I/O) operations on various interfaces, including: SPI (as master), I2S, Camera Interface, I2C, UART and JTAG.

2.4.2 PULP software prospective

The PULP PMSIS (Micro-controller Software Interface Standard) includes the Board Support Package, the Application Programming Interface and drivers required in order to be able to run applications on PULP-based MCU.
PULP works with GNU GCC and LLVM compilers. They support RISC-V standard ISA and other extensions like Xpulpv3, which is the one adopted in this work.

The HAL (Hardware Abstraction Level) includes a collection of functions designed to abstract the underlying hardware like concealing the registry level of the memory map. It simplifies the development process by offering common entry points for accessing hardware functionalities [7].

PULP SDK (Software Development Kit) is a set of tool chain, libraries and scripts. One of the tool is the GVSoC event-driven simulator discussed in chapter 2 section 2.3. The aim of PULP SDK is to help and facilitate the development of applications for PULP-based system.

PULP FreeRTOS includes FreeRTOS plus additional drivers for the development of real-time applications always based on PULP system. The execution of programs is done through RTL simulator (simulating the hardware design) or GVSoC.
In figure 2.4 is given an illustration of how all the software components described in this chapter interact to make the execution possible.

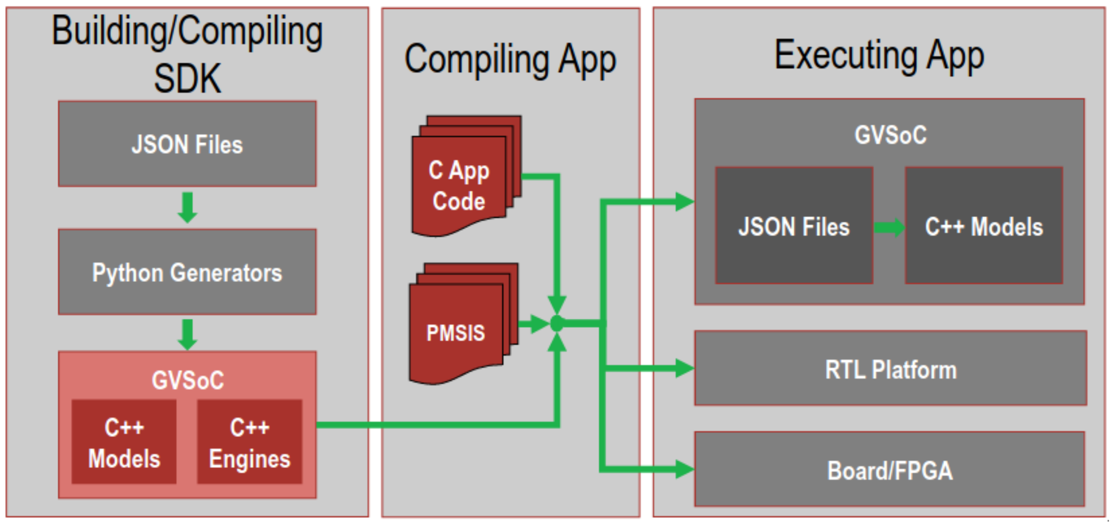

Figure 2.4: PULP Software Environment [7]

Chapter 3

FreeRTOS

3.1 Real-Time Operating System (RTOS)

A Real-Time Operating System is an Operating System designed for real-time computing applications. This kind of OS implement a scheduler which provide a deterministic execution pattern making it particularly suitable for embedded systems that have to process events and data with time requirements [9]. These requirements state that the embedded system must react to a specific event within a precisely defined timeframe. This functionality is essential in systems where even slight delays can have significant negative impact, such as life support systems or air traffic control mechanisms, or more in general when processes or threads must execute within a designated deadline.
A Real-Time Operating System provides robust management of system resources, enabling developers to manage the distribution of processing power, prioritizing critical tasks over less essential ones.
Real-time applications usually consist of a mix of both hard and soft real-time requirements [10]. The RTOS can be distinguished in three categories:

Hard Real-Time Operating Systems
These systems are tailored for applications where missing a time constraint entails the system failure;

Soft Real-Time Operating Systems
In this kind of systems you always try to meet the time constraints, but the missing of a deadline is not disastrous;

Firm Real-Time Operating Systems

These systems lie between hard and soft real-time operating systems (RTOS). Here, failing to meet a deadline is still deemed a system failure but it doesn't imply a disaster.

3.2 FreeRTOS overview

FreeRTOS is a free and open-source real-time operating system suitable for embedded real-time applications running on microcontrollers or microprocessors. It is under the Massachusetts Institute of Technology (MIT) license and can be used for any use (commercial and non) without having to pay any fees. [10].
It has been built focusing on reliability and user-friendliness. FreeRTOS has minimal ROM, RAM and processing overhead and offers a single and independent solution for more than 40 different architectures (including latest RISC-V microcontrollers) and more than 15 development tools. FreeRTOS includes a kernel alongside an expanding collection of IoT libraries, suitable for deployment across various industry domains with a binary image is in the range of 4000 to 9000 bytes. FreeRTOS supports compilation with approximately twenty different compilers and as already mentioned can run on more than 40 different processors architectures. Every pairing of the supported compiler and processor is a unique FreeRTOS port. FreeRTOS is distributed as a collection of C source files. While some of these files are shared across all ports, others are specific for each port. For each official FreeRTOS port is provided a demo application as reference. This demo application comes pre-configured to facilitate the correct assembly of source files and the inclusion of necessary header files.

3.2.1 FreeRTOS kernel

The FreeRTOS kernel is a real-time kernel (or real-time scheduler) enabling application built-upon to meet their hard real-time demands. Applications are organized as a set of autonomous execution threads. On a processor with a single core, only one executing thread can be active at any given time. The kernel determines the execution order by evaluating the priority assigned to each thread by the application designer. The application designer may decide to assign higher priorities to threads fulfilling hard real-time requirements, while assigning lower priorities to threads addressing soft real-time needs. This prioritization scheme guarantees that hard real-time threads have the precedence over soft real-time threads in execution order [10].
The kernel is configured through constants defined in a header file named FreeRTOSConfig.h. The purpose of FreeRTOSConfig.h header file is to customize the

FreeRTOS kernel for a particular application, thus it should be situated within an application directory rather than within one of the FreeRTOS kernel source code directories.
Several stable methodologies exist for developing high-quality embedded software without relying on a multithreading kernel. According to [10] kernel brought a lot of benefits like :

Abstracting away timing information
The real-time operating system (RTOS) oversees execution timing and provides the application with a time-related application programming interface (API). This simplifies the structure of the application code and reduces the overall code size;

Maintainability/Extensibility
By abstracting timing details, fewer interdependencies between modules are introduced, facilitating controlled and predictable software evolution. As the kernel manages timing, application performance is less vulnerable to variations in the underlying hardware;

Code reuse
The software can be structured with greater modularity and reduced interdependencies, thus can be reused;

Improved efficiency
The application code leveraging an RTOS can be entirely event-driven, eliminating the need to spend processing time in polling for events that have not occurred;

Power Management
The efficiency gains achieved through RTOS utilization enable the processor to allocate more time to low-power modes thanks to the presence of the Idle Task;

3.2.2 FreeRTOS kernel Memory allocation

The RTOS requires RAM each time a task, queue, or other RTOS object are instantiated. RAM can be allocated in two ways:

- dynamically and in an automated way from the heap making use of RTOS API object creation functions;
- Statically at compile time, where the developer specifies the size of memory.

The application developers can decide in which mode memory will be allocated by setting the constant `configSUPPORT_DYNAMIC_ALLOCATION` and `configSUPPORT_STATIC_ALLOCATION` respectively to 1 and 0 or vice versa. In the first case memory will be allocated dynamically, in the latter it will be allocated statically. In the work presented in this thesis it has been used dynamic allocation, thus in the following chapter will be used only function API calls that allocates memory dynamically.

3.2.3 FreeRTOS applications memory management

Applications can allocate memory from the FreeRTOS heap when required. FreeRTOS provides various heap management schemes, each with specific features. FreeRTOS offers also the possibility to the application writer to chose its own implementation of heap management. As stated in [11] implementation offered by the OS are the following :

`heap_1` : Among all is the simplest solution. It is suitable for applications where tasks are never deleted. It is a scenario common in many applications using FreeRTOS;

`heap_2` : Employs a best fit algorithm and let already allocated memory blocks to be freed. However, with this implementation adjacent free blocks can't be merged into a single large block;

`heap_3` : It entails a basic wrapper for the standard C library malloc() and free() functions, ensuring thread safety;

`heap_4` : This implementation employs a first fit algorithm and, unlike `heap_2`, merges adjacent free memory blocks into a single large block;

`heap_5` : This implementation utilizes the same first fit and memory coalescence algorithms as `heap_4`, and allows the allocation of multiple non-contiguous memory regions in the heap area;

The total size the heap can be defined by setting the costant `configTOTAL_HEAP_SIZE` declared in FreeRTOSConfig.h

3.3 FreeRTOS task

3.3.1 Task/Process basic concept

A process is the unit of work in a system. It can be defined informally as a program in execution. As mentioned in [12] it's important to highlights the difference between a program and a process. A program is essentially a passive entity, a file (often called executable file) stored on a disk containing a list of instructions. When this file is loaded into memory it becomes a process, an active entity capable to perform tasks and interact with the resources of the system. A process is represented in memory by several components as shown in Figure 3.1 :

Text Segment : the text segment, also referred as the code segment, contains the executable code of the program. It includes the machine instruction executed sequentially by the CPU to perform a specific task. Usually this segment can be accessed in read-only mode and is shared among multiple instances of the same program.

Stack Segment : the stack segment is a memory area used to store local variables, function parameters and return addresses. It dynamically grows and shrink in response to function calls and returns;

Data Segment : the data section contains global and static variables. This segment is read-write since these variables can change their values during the execution of the program;

Heap Segment : the heap segment is a memory region allocated dynamically used to store data which allocated at run-time. This area memory can be accessed in read or write mode.

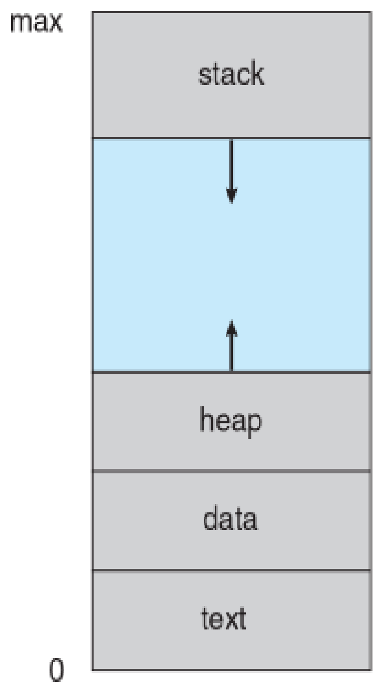

Figure 3.1: Memory Representation of a process [12]

Each process is represented in the operating system by a Process Control Block (PCB), also called Task Control Block (TCB). The PCB serves as the repository for any information that can be used from the OS for that process/task. It holds important information of the related tasks such as :

- **the process state** : this part is described in detail in 3.3.3;
- **the program counter** : it stores the memory address of the next instruction to be executed by the process. As the program executes, the program counter dynamically updates, ensuring the program counter is updated to point to the next instruction in memory;
- **the CPU registers** : in these registers are included both special-purpose (i.e the program counter and status register) and general-purpose (i.e. accumulator, index registers, and stack pointer) registers. They change in number and type depending on the architecture. The values must be stored in presence of a context switch or interrupt;
- **the process ID (PID)** : it is a number used as the identifier of the process. This label allows the operating system to recognize a specific task during

process management and commumnication;

- **the memory-management information** : it contains comprehensive information about the process's memory allocation like the base address and the size of its code, data, stack and heap segment. Depending on the OS it could include the address of segment and page tables.

3.3.2 Task implementation

In FreeRTOS tasks are implemented as C functions, so it is basically a small program in its own right [10].
Tasks must implement a function that returns void and accepts a void pointer as a parameter. A task function should have the structure shown in Listing 3.1. An application writer must never let a FreeRTOS task to return from the function it implements for any reason, so it doesn't include a return statement. If a task is no longer necessary, it should be explicitly deleted.

```
void vATaskFunction( void *pvParameters )
    {

        for( ;; )
        {
            -- Task application code here. --
        }

        /* Tasks must not attempt to return from their implementing
        function or otherwise exit.  In newer FreeRTOS port
        attempting to do so will result in an configASSERT() being
        called if it is defined.  If it is necessary for a task to
        exit then have the task call vTaskDelete( NULL ) to ensure
        its exit is clean. */
        vTaskDelete( NULL );

    }
```

Listing 3.1: FreeRTOS task function

You can generate other tasks from within the task function itself, each having a distinct execution instance with its own stack. The FreeRTOS API function to create a task is TaskCreate() having the prototype listed in 3.2:

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        configSTACK_DEPTH_TYPE usStackDepth,
                        void * pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t * pxCreatedTask );
```

Listing 3.2: FreeRTOS task create prototype API function

As stated in [13], here is given a detailed description of the parameters required in Listing 3.2 :

pvTaskCode
It is the pointer to the function performed by the task;

pcName
A name given to the task. The purpose of this parameter is to help developers during debugging session, and it is not used in any relevant way by FreeRTOS. The constant `configMAX_TASK_NAME_LEN` sets the maximum length for a task name, including the NULL terminator. Providing a longer string leads to its truncation;

usStackDepth
This variable is used by the kernel to know how many bits has to be allocated for the stack associated to that task;

pvParameters
This parameter is of type pointer to void to enable the task parameter to effectively receive a parameter of any type indirectly through casting;

uxPriority
Specifies the priority of the task used by the kernel scheduler. Priorities can range from 0, representing the lowest priority, to (`configMAX_PRIORITIES` – 1), which is the highest priority. If this variable holds a greater value it will be truncated to the maximum priority available;

pxCreatedTask
This variable provides a handle to the created task. It enables referencing

the task in API calls to modify the task priority or delete the task. If the application does not require the task handle, it can be assigned the NULL value.

The returned values are:

pdPASS
It means the task creation process has been completed successfully;

`errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY`
It means the task creation process has failed due to insufficient heap memory available

The xTaskCreate API function allocates memory for the Task Control Block as for the stack. Then it properly initializes the task by calling the prvInitialiseNewTask kernel function and finally add it to the list of the task in the Ready state.

3.3.3 Task states

An application can create and execute several tasks during its lifetime. In a single-core processor, only one task at a time can be executed. This means a task can be in one of two states: Running and Not Running [10]. A task enters the Running state when the processor is actively executing its code. A task in the Not Running state, it is temporarily paused, and its state is preserved allowing to be resumed later. When its execution has been resumed (enters the Running State), the task continues from the instruction it was about to execute before entering the Not Running state. The Not Running state can be expanded in the following ones :

Blocked
a task is in this state when it is waiting for a particular event to occur. The event the task is waiting for can fall into two different categories:

a. **Temporal (time-related) events** These events happen either upon the expiration of a specified delay period or when an absolute time is reached;
b. **Synchronization events** These events are triggered by another task or interrupt;

A task can block on a synchronization event with a timeout, effectively waiting for both types of events simultaneously.

Suspended
A task which is Not Running and doesn't fall in any of the two categories mentioned before is in the Ready State. They are ready to be selected by the scheduler and start or resume the execution.

3.3.4 Task representation

Each task is represented in the operating system by its Task Control Block. Here a brief decription of the most important variable of the struct named `TCB_t` shown in Listing 3.3:

pxTopOfStack
It points to the address of the last item inserted into the stack;

xStateListItem
It is an item used to be inserted into a list used by the scheduler to manage tasks in different states. The lists in which this item can be stored identify the current state of the task;

xEventListItem
It is an item used to be inserted into a list used by tasks to wait for synchronization events. This item allows the scheduler to quickly identify and unblock tasks when the corresponding event occurs;

pxStack
It points to the start address of the stack;

```
typedef struct tskTaskControlBlock{

    /*------------------------other variables------------------------*/

    volatile StackType_t     *pxTopOfStack  ;
    ListItem_t              xStateListItem;
    ListItem_t              xEventListItem;
    UBaseType_t             uxPriority;
    StackType_t             *pxStack;
    char                    pcTaskName[ configMAX_TASK_NAME_LEN ];
    UBaseType_t         uxBasePriority;

```

```c
} tskTCB;

typedef tskTCB TCB_t;
```

Listing 3.3: FreeRTOS most relevant variables in tcb struct

3.3.5 Task Priority

In FreeRTOS as in other RTOS tasks are marked by a priority number. The priority associated to the task is used by the scheduler to know which task need to be switched in first when occurs a context switch.
The priority of a task is assigned during its creation through the uxPriority parameter but can be changed later on by calling the API function vTaskPrioritySet(). The application-defined and compile-time configuration constant `configMAX_PRIORITIES` sets the number of the available priorities which can range from 0 to (`configMAX_PRIORITIES` − 1).

3.3.6 Idle Task

In FreeRTOS there must be always one task in the Running state, so when the scheduler is started a task called Idle Task is automatically created. The Idle Task must have a priority number equal to 0, allowing other tasks with higher priority to enter the Running state first.

3.4 FreeRTOS time measurement and tick interrupt

The FreeRTOS real-time kernel measures time using a tick count variable [11]. This tick count is incremented by a timer interrupt known as the RTOS tick interrupt. With each increment of the tick count, the kernel checks if it is time to unblock or wake a task, allowing for precise time measurement up to the resolution defined by the frequency of the timer interrupt.

3.5 FreeRTOS scheduling overview

The FreeRTOS scheduling algorithm determines which task will be moved from the Ready state to the Running one. The behaviour of the scheduler can be changed by setting three constant in the FreeRTOSConifg.h header file :
`configUSE_PREEMPTION`, `configUSE_TIME_SLICING`
and `configUSE_TICKLESS_IDLE`. If `configUSE_TICKLESS_IDLE` is set to 1 the tick interrupt is turned completely off for extended periods. This option should be used by specifically for scenario in which is required to minimize the power consumption of the embedded system. If this variable is left undefined it will be automatically set to the default value of 0.
In all single-core configuration, the FreeRTOS scheduler follows a "take in turn" policy for tasks that share the same priority. This policy, also known as Round Robin Scheduling, ensures that Ready state tasks of equal priority are selected to enter the Running state in turn and not guarantee equal time-sharing between those tasks.

3.5.1 Fixed-priority preemptive scheduling

The default FreeRTOS scheduling algorithm policy is the Fixed-Priority Premptive with Round-Robin time slicing of tasks sharing the same priority.

- Fixed Priority
It means the scheduler is not allowed to change to priority that has been assigned to the task when it ha been created. The priority of a task can be changed or by the task itself or by another task;

- Preemptive
A preemptive scheduler will 'preempt' a task from the Running state if there is at least one task with higher priority in the Ready state. A task is preempted when it is switched out without its will (i.e. without an explicit yielding or blocking). The preemption can happen at any given time, not only when occurs a tick interrupt;

- Time Slicing
Is technique employed to distribute processing time among tasks of equal priority. It ensures fair utilization of CPU resources among the cited tasks and prevent them from monopolizing the CPU;

The `configIDLE_SHOULD_YIELD` compile time configuration constant can be used to change how the Idle task is scheduled:

- `configIDLE_SHOULD_YIELD` sets to 0 : The idle task is keept in the Running state for the entire time slice unless preempted by a task with a higher priority;
- `configIDLE_SHOULD_YIELD` sets to 1 : The idle task yields giving the reamining time slice time to other tasks having its same priority;

The FreeRTOS scheduler can be configured to use prioritized preemptive scheduling without time slicing by setting the `configUSE_TIME_SLICING` to 0. When time slicing is disabled, the scheduler only selects a new task to enter the Running state when either:

- a higher priority task enter the Ready state;
- The task in the Running state transitions to the Blocked or Suspended state;

When time slicing is turned off fewer context switches occure. The advantage is to reduce the scheduler processing overhead, the negative point instead is to assign different amount of processing time to tasks sharing the same priority.

3.5.2 Cooperative Scheduling

The FreeRTOS scheduling algorithm is cooperative when the `configUSE_PREEMPTION` constant is set to 0. In the cooperative scheduler algorithm, a context switch only occurs when the task in the Running state transitions to the Blocked state or explicitly yields by calling taskYIELD(). Time slicing is useless because tasks are never preempted, thus `configUSE_TIME_SLICING` can be set to any value.

3.6 FreeRTOS scheduling features implementation

3.6.1 Scheduler execution

The FreeRTOS scheduler is runned by calling the FreeRTOS API function vTaskStartScheduler. This function is called inside the main body of the application code as shown in Listing 3.4. After scheduler is run, only interrupts and tasks will be executed.

```

/* Define a task function. */

void vATask( void )
{

    for( ;; )
    {

       /* Task code goes here. */

    }

}

void main( void )

{
    system_init();

    /* Create at least one task, in this case the task function defined above is created. Calling vTaskStartScheduler() before any tasks have been created will cause the idle task to enter the Running state. */

    xTaskCreate(vTaskCode,
                "task name",
                STACK_SIZE,
                NULL,
                TASK_PRIORITY,
                NULL );

    /* Start the scheduler. */

    vTaskStartScheduler();

    /* This code will only be reached if the idle task could not be created inside vTaskStartScheduler(). An infinite loop is used to assist debugging by ensuring this scenario does not result in main() exiting. */

    for( ;; );

}
```

Listing 3.4: How properly run FreeRTOS scheduler

In Listing 3.5 is shown the main important section of the body of the vTaskStartSCheduler API function.

```
void vTaskStartScheduler( void )
{
BaseType_t xReturn;

    /* The Idle task is being created using dynamically allocated RAM
    . */

    xReturn = xTaskCreate(  prvIdleTask,
                            configIDLE_TASK_NAME,
                            configMINIMAL_STACK_SIZE,
                            ( void * ) NULL,
                            portPRIVILEGE_BIT,
                            &xIdleTaskHandle );


    if( xReturn == pdPASS )
    {
        /* Interrupts are turned off here, to ensure a tick does not
    occur before or during the call to xPortStartScheduler().  The
    stacks of the created tasks contain a status word with interrupts
    switched on so interrupts will automatically get re-enabled when
    the first task starts to run. */

        portDISABLE_INTERRUPTS();

        xNextTaskUnblockTime = portMAX_DELAY;
        xSchedulerRunning = pdTRUE;
        xTickCount = ( TickType_t ) configINITIAL_TICK_COUNT;

        traceTASK_SWITCHED_IN();

        /* Setting up the timer tick is hardware specific and thus in
     the portable interface. */

        if( xPortStartScheduler() != pdFALSE )
        {
            /* Should not reach here as if the scheduler is running
    the function will not return. */
        }
        else
        {
            /* Should only reach here if a task calls
    xTaskEndScheduler(). */
        }
    }
    else
    {
        /* This line will only be reached if the kernel could not be
    started, because there was not enough FreeRTOS heap to create the
    idle task or the timer task. */

        configASSERT( xReturn !=
   errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY );
    }

    /* Prevent compiler warnings if INCLUDE_xTaskGetIdleTaskHandle is
    set to 0, meaning xIdleTaskHandle is not used anywhere else. */

    ( void ) xIdleTaskHandle;
}
```

Listing 3.5: vTaskStartScheduler FreeRTOS API body function relevant instructions

In FreeRTOS there must be at least one task in the Running state, thus as first the Idle task is created. The function will return only if there is enough heap memory space to allocate the Idle task. If the idle task has been created successfully all interrupts are disabled because they could interfere when xPortStartScheduler function is called. They will be re-enabled when the first task start to run.
The Tick is set and a call to `trace_TASK_SWITCHED_IN` is made.
`traceTASK_SWITCHED_IN` is a function belonging to the FreeRTOS tracing features, which provides runtime visibility into the execution of tasks, interrupts, and other events within the FreeRTOS kernel. The `traceTASK_SWITCHED_IN` function is called whenever a task is switched in. It allows for tracing or logging of task switching events, which can be useful for debugging, performance analysis, or profiling purposes. By default, this function may be implemented as a no-op if tracing is not enabled in the FreeRTOS configuration.
At the end a call to the function xPortStartScheduler is made. This function is architecture-specific and is responsible for initializing the hardware-specific components required to start the scheduler.
The xPortStartScheduler function performs the following tasks:

- Setting up the hardware timer: The scheduler relies on a hardware timer to generate periodic interrupts, which are used to trigger task context switches. The xPortStartScheduler function initializes this timer and sets up the interrupt handler for the timer interrupt;

- Setting up the system tick timer: The FreeRTOS scheduler uses a system tick timer to track time and manage task scheduling. The xPortStartScheduler function initializes this timer and configures it to generate interrupts at regular intervals, typically at a frequency of 1 kHz;

- Enabling interrupts: Once the hardware timer and system tick timer are set up, the xPortStartScheduler function enables interrupts to allow the scheduler to respond to timer interrupts and schedule tasks accordingly;

- Transitioning to the scheduler: After initializing the necessary hardware components and enabling interrupts, the xPortStartScheduler function transitions control to the FreeRTOS scheduler, allowing it to begin scheduling tasks;

- Set up the hardware ready for the scheduler to take control. This generally sets up a tick interrupt and sets timers for the correct tick frequency;

3.6.2 Scheduler context switch

When a context switch needs to take place, the vTaskSwitchContext API function is called (its main body with the most relevant instructions are shown in Listing 3.6). More specifically, the vTaskSwitchContext function performs the following actions:

- Saves the context (registers, stack pointer, etc.) of the currently executing task.

- Selects the next task to run from the ready list.

- Restores the context of the selected task, allowing it to continue execution from where it was previously interrupted.

```
void vTaskSwitchContext( void )
{
    if( uxSchedulerSuspended != ( UBaseType_t ) pdFALSE )
    {
        /* The scheduler is currently suspended - do not allow a
    context switch. */
        xYieldPending = pdTRUE;
    }
    else
    {
        xYieldPending = pdFALSE;
        traceTASK_SWITCHED_OUT();

        ...

        /* Select a new task to run using either the generic C or
    port optimised asm code. */

        taskSELECT_HIGHEST_PRIORITY_TASK();
        traceTASK_SWITCHED_IN();

        ...

    }
}
```

Listing 3.6: vTaskStartScheduler FreeRTOS API body function relevant instructions

The `traceTASK_SWITCHED_IN` as been already described in 3.6.1.
The `traceTASK_SWITCHED_OUT` is the same as `traceTASK_SWITCHED_IN` but referred to the task the scheduler is switching out.
The `taskSELECT_HIGHEST_PRIORITY_TASK` function is responsible for selecting the highest priority task that is ready to run. It scans the ready list to find the highest priority task that is ready to run typically selecting the task with the highest priority value. The Listing 3.7 shows the macro expansion.

```
#define taskSELECT_HIGHEST_PRIORITY_TASK()
                                \
{
                                \
    UBaseType_t uxTopPriority;
                                \

    portGET_HIGHEST_PRIORITY( uxTopPriority, uxTopReadyPriority );
                                \

    configASSERT( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[
    uxTopPriority ] ) ) > 0 ); \

    listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[
    uxTopPriority ] ) );    \
} \
```

Listing 3.7: `taskSELECT_HIGHEST_PRIORITY_TASK` function expansion

Line 3

`UBaseType_t` uxTopPriority declares a local variable uxTopPriority of type `UBaseType_t`, which represents the highest priority task's priority.

Line 5

`portGET_HIGHEST_PRIORITY`(uxTopPriority, uxTopReadyPriority) is a port-specific function or macro that determines the highest priority task that is ready to run. It typically scans the ready list or priority list to find the highest priority task and stores its priority value in uxTopPriority. uxTopReadyPriority is a port-specific variable that may be used to store additional information related to the highest priority task.

Line 7

configASSERT(`listCURRENT_LIST_LENGTH`(&(pxReadyTasksLists[uxTopPriority]))>0) declares a local variable uxTopPriority of type `UBaseType_t`, which represents the highest priority task's priority.

Line 9

`listGET_OWNER_OF_NEXT_ENTRY`(pxCurrentTCB, &(pxReadyTasksLists[uxTopPriority])) retrieves the next task from the ready list corresponding to the highest priority and stores its task control block (TCB) pointer in pxCurrentTCB. The task control block represents the state and context of the task.

3.7 FreeRTOS Ready list implementation

The Ready list is the list where are referenced tasks in the Ready state. It is used by the scheduler to chose which task to pick up during a context switch. In Listing 3.8 is shown is definition inside the task.c file of the FreeRTOS source code.

```
            PRIVILEGED_DATA static List_t pxReadyTasksLists[
    configMAX_PRIORITIES ];

```

Listing 3.8: FreeRTOS task prototype function

pxReadyTasksLists is declared as a static global array having as much entry as the maximum value of priority given by the macro `configMAX_PRIORITIES`. Declaring it static made it accessible only within the file where it is defined. Each entry of the array is of type `List_t`. The struct `List_t` is defined in file list.h as in Listing 3.9.

```
typedef struct xLIST
{

    listFIRST_LIST_INTEGRITY_CHECK_VALUE
    volatile UBaseType_t uxNumberOfItems;
    ListItem_t * configLIST_VOLATILE pxIndex;
    MiniListItem_t xListEnd;
    listSECOND_LIST_INTEGRITY_CHECK_VALUE

} List_t;
```

Listing 3.9: FreeRTOS task prototype function

Each item of the array is composed of :

- a variable pxIndex pointing to a `ListItem_t` type : this variable as an index to scan through the tasks;
- a xListEnd variable of type `MiniListItem_t` : it is used to mark the end of the list;
- the uxNumberOfItems which indicates the number of items present at the related index of the pxReadyTasksLists;

The struct `ListItem_t` is shown in Listing 3.10.

```
struct xLIST_ITEM
{

    listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE
    configLIST_VOLATILE TickType_t xItemValue;
    struct xLIST_ITEM * configLIST_VOLATILE pxNext;
    struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;
    void * pvOwner;
    struct xLIST * configLIST_VOLATILE pxContainer;
    listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE

};
typedef struct xLIST_ITEM ListItem_t;
```

Listing 3.10: FreeRTOS task prototype function

At each index of the pxReadyTasksLists are saved tasks in the Ready state. Tasks with a priority number equal to i can be found at index i of the Ready list. Each Ready list index is implemented as a Circular Doubly Linked List. From each item of the list you can access the previous item through the pxPrevious pointer and to the next one through pxNext. The void pointer pvOwner points to the TCB of the task. It is cast as void to avoid compilation error since the struct of the TCB is defined elsewhere and it is not included in list.h file. The pxContainer instead points to the list where the task is saved.

Chapter 4

Work implementation

4.1 Project goal

The aim of the work of this thesis is trying to improve the performance of the FreeRTOS Operative System modifying the default schedule's behaviour. The new scheduling algorithm make use of the hardware event counter stored in the HPCs to work properly. It can be enabled or disabled allowing to perform later analysis with respect to the original implementation. For all tasks executed the kernel keep tracks of the number of events occurring when the task is in the Running state. The values stored in the chosen performance counter registers are used to calculate a cost. The scheduler exploits this cost to decide which is the next task to pick up in each context switch.

4.2 Kernel configuration

To be able to implement the new algorithm scheme as optional it has been exploited the FreeRTOSConfig.h file described in 3.2.1. As showed in Listing 4.1 it has been added a macro named `configeUSE_PCER_SCHEDULING_POLICY` which allow to enable or disable the new policy by setting it respectively to 1 and 0.

```

#define configeUSE_PCER_SCHEDULING_POLICY 1


```

Listing 4.1: `configeUSE_PCER_SCHEDULING_POLICY` macro definition

In the FreeRTOS.h header file of the FreeRTOS kernel source code it is checked

if the `configeUSE_PCER_SCHEDULING_POLICY` macro has been already defined. If the macro was not declared and defined in the configuration file of the developer's custom application the default implementation adopted by the kernel is to set it to 0, thus using the default scheduling algorithm.
Then for each performance counter register a macro has been defined. A total of 17 macros allow enabling or disable a specific performance counter register by setting it to 1 or 0. It has been taken this approach because the application developer can choose the hardware events to monitor simply by modifying the value of the macro. In Listing 4.2 is shown the declaration of the macros.

```

#define PCRs_STATE_CSR_PCER_CYCLES          0
#define PCRs_STATE_CSR_PCER_INSTR           0
#define PCRs_STATE_CSR_PCER_LD_STALL        1
#define PCRs_STATE_CSR_PCER_JMP_STALL       0
#define PCRs_STATE_CSR_PCER_IMISS           1
#define PCRs_STATE_CSR_PCER_LD              0
#define PCRs_STATE_CSR_PCER_ST              0
#define PCRs_STATE_CSR_PCER_JUMP            0
#define PCRs_STATE_CSR_PCER_BRANCH          1
#define PCRs_STATE_CSR_PCER_TAKEN_BRANCH    1
#define PCRs_STATE_CSR_PCER_RVC             0
#define PCRs_STATE_CSR_PCER_LD_EXT          1
#define PCRs_STATE_CSR_PCER_ST_EXT          1
#define PCRs_STATE_CSR_PCER_LD_EXT_CYC      0
#define PCRs_STATE_CSR_PCER_ST_EXT_CYC      0
#define PCRs_STATE_CSR_PCER_TCDM_CONT       1
#define PCRs_STATE_CSR_PCER_CSR_HAZARD      1
```

Listing 4.2: FreeRTOS PCCRs associated macros

Additionally has been modified the body of the function `system_init` (Listing 4.3) which set the system at the start up. The new functions (described in Section 4.3) initialize and set the performance counters.

```
void system_init(void)
{
    ...

    vPerfInitialize();
    vPerfStopCounting();
    vPerfSetAllCounters(0);

    ...
}
```

Listing 4.3: `pcer_v2.h` header file

4.3 Hardware Performance Counter

An important step of the work has been writing an API which allows the application writer to set up the performance counter through special registers and to perform a read or a write in one of the PCCRs, which holds the counter of the relative event. The developing of a functional API has started from the file `pcer_v2.h`, showed in Listing 4.4, which is already present in the pulp-freertos source code with some code written.

```

#ifndef _ARCHI_RISCV_PCER_V1_H
#define _ARCHI_RISCV_PCER_V1_H


#define CSR_PCER_CYCLES    0  /* Count the number of cycles the core was running */
#define CSR_PCER_INSTR     1  /* Count the number of instructions executed */
#define CSR_PCER_LD_STALL    2  /* Number of load use hazards */
#define CSR_PCER_JMP_STALL    3  /* Number of jump register hazards */
#define CSR_PCER_IMISS     4  /* Cycles waiting for instruction fetches. i.e. the number of instructions wasted due to non-ideal caches */
#define CSR_PCER_LD    5  /* Number of memory loads executed. Misaligned accesses are counted twice */
#define CSR_PCER_ST    6  /* Number of memory stores executed. Misaligned accesses are counted twice */
#define CSR_PCER_JUMP    7  /* Number of jump instructions seen, i.e. j, jr, jal, jalr */
#define CSR_PCER_BRANCH    8  /* Number of branch instructions seen, i.e. bf, bnf */
#define CSR_PCER_TAKEN_BRANCH 9  /* Number of taken branch instructions seen, i.e. bf, bnf */
#define CSR_PCER_RVC     10  /* Number of compressed instructions */
#define CSR_PCER_LD_EXT    11  /* Number of memory loads to EXT executed. Misaligned accesses are counted twice. Every non-TCDM access is considered external */
#define CSR_PCER_ST_EXT    12  /* Number of memory stores to EXT executed. Misaligned accesses are counted twice. Every non-TCDM access is considered external */
```

```
#define CSR_PCER_LD_EXT_CYC 13  /* Cycles used for memory loads to
    EXT. Every non-TCDM access is considered external */
#define CSR_PCER_ST_EXT_CYC 14  /* Cycles used for memory stores to
    EXT. Every non-TCDM access is considered external */
#define CSR_PCER_TCDM_CONT  15  /* Cycles wasted due to TCDM/log-
    interconnect contention */
#define CSR_PCER_CSR_HAZARD  16 /* Cycles wasted due to CSR access */

#define CSR_PCER_NB_EVENTS       17
#define CSR_PCER_NB_INTERNAL_EVENTS     17
#define CSR_NB_PCCR                  31


#define CSR_PCER_EVENT_MASK(eventId)  (1<<(eventId))
#define CSR_PCER_ALL_EVENTS_MASK  0xffffffff

#define CSR_PCMR_ACTIVE            0x1 /* Activate counting */
#define CSR_PCMR_SATURATE          0x2 /* Activate saturation */

static inline void pcerEventsSet(unsigned int eventMask)
{
    asm volatile ("csrw 0xCC0, %0" :: "r" (eventMask));
}

static inline unsigned int pcerEventsGet()
{
  unsigned int result = 0;
  asm volatile ("csrr %0, 0xCC0" : "=i" (result) );
  return result;
}

static inline void pcerModeSet(unsigned int confMask)
{
  asm volatile ("csrw 0xCC1, %0" :: "r" (confMask));
}

static inline void pcerModeGet(unsigned int confMask)
{
    unsigned int result = 0;
    asm volatile ("csrr %0, 0xCC1" : "=i" (result) );
}

static inline void pcerSetAllCounters(unsigned int value) {
  asm volatile ("csrw  0x79F, %0" :: "r" (value));
}

static inline unsigned int pcerGetValue(const unsigned int counterId)
     {
  unsigned int value = 0;
  switch(counterId) {
    case  0: asm volatile ("csrr %0, 0x780" : "=r" (value)); break;
    case  1: asm volatile ("csrr %0, 0x781" : "=r" (value)); break;
    case  2: asm volatile ("csrr %0, 0x782" : "=r" (value)); break;
    case  3: asm volatile ("csrr %0, 0x783" : "=r" (value)); break;
    case  4: asm volatile ("csrr %0, 0x784" : "=r" (value)); break;
    case  5: asm volatile ("csrr %0, 0x785" : "=r" (value)); break;
    case  6: asm volatile ("csrr %0, 0x786" : "=r" (value)); break;
    case  7: asm volatile ("csrr %0, 0x787" : "=r" (value)); break;
    case  8: asm volatile ("csrr %0, 0x788" : "=r" (value)); break;
    case  9: asm volatile ("csrr %0, 0x789" : "=r" (value)); break;
    case 10: asm volatile ("csrr %0, 0x78A" : "=r" (value)); break;
    case 11: asm volatile ("csrr %0, 0x78B" : "=r" (value)); break;
    case 12: asm volatile ("csrr %0, 0x78C" : "=r" (value)); break;
    case 13: asm volatile ("csrr %0, 0x78D" : "=r" (value)); break;
    case 14: asm volatile ("csrr %0, 0x78E" : "=r" (value)); break;
    case 15: asm volatile ("csrr %0, 0x78F" : "=r" (value)); break;
    case 16: asm volatile ("csrr %0, 0x790" : "=r" (value)); break;
  }
  return value;
}

#endif
```

Listing 4.4: `pcer_v2.h` header file

`CSR_PCER_EVENT_MASK`
This macro shift the value 1 as many position as the eventId value. It is useful to set the mask for the Performance Counter Event Register enabling the counting of the wanted PCCR;

pcerModeSet, pcerModeGet
These two functions allow respectively to write and read a value into the Performance Counter Mode Register (see Section 2.2.3);

pcerEventSet, pcerEventeGet
They allow to write and read a value in the Performance Counter Event Register (see Section 2.2.4);

pcerSetAllCounters
This function allow writing a value passed as parameter of the function in all the PCCRs, by writing in the PCCR named "ALL" (see Table 2.3);

pcerGetValue
: This function allow reading the value of a speficic performance counter counter register given its counterId. Each register is associated to an Id through the macro declared on top of the file;

On top of this it has been created a new file called `perf_API.h` which include `pcer_v2.h`. In Listing 4.5 are reported the main important functions:

vPerfInitialize
: This function create a mask that will be used as paramater of the function pcerEventSet. Thanks to the macro defined in the FreeRTOS.h header file will be created a mask only for those PCCR the application developer want to track. If a performance counter is enabled by the application developer the macro `CSR_PCER_EVENT_MASK` with the Id of the event and the result is ORED with the previous value of eventMask;

vPerfReadAllValues
: This function allow to read the value of the PCCRs enabled by the macro defined in FreeRTOS.h and store it into an array of unsigned integer which is passed as parameter;

usPcerGetNumActive
: It returns the number of PCCRs which are enabled by the macro declaration in FreeRTOS.h;

vPerfStartCounting
: This function make the PCCRs to start counting the occurences of their associated event;

vPerfStopCounting
: On the contrary of vPerfStartCounting it stop the counting of the events by writing in the PCM Register;

vPerfResetandStopCounting
: It performs the same operation of the vPerfStartCounting after having set all counters to 0;

```c
#include "archi/riscv/pcer_v2.h"
#include "FreeRTOS.h"
#include "printfCustom.h"
#include <stdio.h>

static inline void vPerfInitialize(void)
{

    unsigned int eventMask = 0;
    #if (PCRs_STATE_CSR_PCER_CYCLES == 1)
        eventMask = eventMask | CSR_PCER_EVENT_MASK(CSR_PCER_CYCLES);
    #endif
    #if (PCRs_STATE_CSR_PCER_INSTR == 1)
        eventMask = eventMask | CSR_PCER_EVENT_MASK(CSR_PCER_INSTR);
    #endif

    ...

    #if (PCRs_STATE_CSR_PCER_TCDM_CONT == 1)
        eventMask = eventMask | CSR_PCER_EVENT_MASK(
    CSR_PCER_TCDM_CONT);
    #endif
    #if (PCRs_STATE_CSR_PCER_CSR_HAZARD == 1)
        eventMask = eventMask | CSR_PCER_EVENT_MASK(
    CSR_PCER_CSR_HAZARD);
    #endif
    pcerEventsSet(eventMask);

}

static inline void vPerfResetandStartCounting(void)
{
    pcerSetAllCounters(0);

    pcerModeSet(CSR_PCMR_ACTIVE | CSR_PCMR_SATURATE);
}

static inline void vPerfStartCounting(void)
{
    pcerModeSet(CSR_PCMR_ACTIVE | CSR_PCMR_SATURATE);
}


static inline void vPerfStopCounting(void)
{
    pcerModeSet(0);
}

```

```

static inline void vPerfReadAllValues(unsigned int *pcrValues)
{
    unsigned int i = 0;

    #if (PCRs_STATE_CSR_PCER_CYCLES == 1)
        pcrValues[i] = pcerGetValue(CSR_PCER_CYCLES);
        i++;
    #endif

    ...

    #if (PCRs_STATE_CSR_PCER_CSR_HAZARD == 1)
        pcrValues[i] = pcerGetValue(CSR_PCER_CSR_HAZARD);
        i++;
    #endif
}

static inline  unsigned int usPcerGetNumActive()
{
    unsigned int count = 0;
        #if (PCRs_STATE_CSR_PCER_CYCLES == 1)
            count++;
        #endif

        ...

        #if (PCRs_STATE_CSR_PCER_CSR_HAZARD == 1)
            count++;
        #endif
    return count;
}
```

Listing 4.5: `perf_API.h` header file

4.4 Task

For what concern the tasks it has been implemented additional variables in the Task Control Block allowing the new scheduling algorithm to perform as well. These new structures are reported in Listing 4.6 and they are correctly initialized when the task is created, in the body of prvInitialiseNewTask function, as shown in Listing 4.7.

```

        unsigned int *PCRsValue;
        unsigned int *PCRsNum;
        unsigned int NumActivePCRs;
        #if (configeUSE_PCER_SCHEDULING_POLICY == 1)
            unsigned int cost;
        #endif

```

Listing 4.6: TCB new structures

It's important to notice that only the variable cost is declared and initialized if the macro `configeUSE_PCER_SCHEDULING_POLICY` is set to 1, letting the others available for any other possible future implementation. Additionally, the initialization of these new structures is not performed for the IDLE task because it will be a computational waste of resources and time.

PCRsValue
It is an array of unsigned int allocated dynamically. After being correctly initialized it will contain as much entry as the number of performance counter registers enabled by the application writer through the macros of the FreeRTOS.h header file;

PCRsNum
It is a dynamically allocated array of unsigned int. As for the PCRsValue contains as much entry as the number of performance counter registers. Item at index i contains the Id of the performance counter whose value is stored at index i of PCRsValue;

NumActivePCRs
This variable holds the number of performance counter enabled by the macros;

cost
It hold the cost computed over the values holded in the performance counters. It is used to scheudle the task at every context switch;

```
if (strcmp(pcName, configIDLE_TASK_NAME)!=0){
        pxNewTCB->NumActivePCRs = usPcerGetNumActive();
        #if (configeUSE_PCER_SCHEDULING_POLICY == 1)
            pxNewTCB->cost = 0;
        #endif

        if ((pxNewTCB->PCRsValue = (unsigned int *)pvPortMalloc(
    sizeof(unsigned int)*pxNewTCB->NumActivePCRs)) == NULL)
             printfC("Errore nell'allocazione dell'array PCRsValue\n"
    );
        for(uint8_t i=0; i<pxNewTCB->NumActivePCRs; i++){
            pxNewTCB->PCRsValue[i] = 0;
        }
        if ((pxNewTCB->PCRsNum = (unsigned int *)pvPortMalloc(
    pxNewTCB->NumActivePCRs * sizeof(unsigned int))) == NULL)
             printfC("Errore nell'allocazione dell'array PCRsNum\n");
        unsigned int i = 0;
        #if (PCRs_STATE_CSR_PCER_CYCLES == 1)
            pxNewTCB->PCRsNum[i] = 0;
            i++;
        #endif

        ...

        #if (PCRs_STATE_CSR_PCER_CSR_HAZARD == 1)
            pxNewTCB->PCRsNum[i] = 16;
            i++;
        #endif

}
```

Listing 4.7: Initialization of the new structures of the TCB

4.5 New Scheduling Algorithm Implementation

In this section are described the changes applied in the body's functions which manage the behaviour of the scheduler. As in Listing 4.8 inside the body of the xPortStartScheduler function, right before the calling of xPortStartFirstTask which makes the first task execute, the performance counter registers start to count the occurrence of the events.

```
BaseType_t xPortStartScheduler( void )
{
extern void xPortStartFirstTask( void );
```

```

    ...

    vPerfStartCounting();

    xPortStartFirstTask();

    ...

}
```

Listing 4.8: xPortStartScheduling body function changes

The main work is focused in the function managing the context switch, called vTaskSwitchContext, showed in Listing 4.9. It is useless to perform all the added functionality to the IDLE task, so they will be executed after an if statement checking the current task isn't the IDLE one by its name. Additionally, the if statement filters out the tasks who has been deleted and need to be switched out yet, thus no more present in the ReadyList.
For what concern the task switching out, first of all the performance counter registers are stopped. Then by calling the vPerfReadAllValues function the values stored in these special registers are read and stored in a temporary array called values, which contains as many items as the number of performance counter registers enabled by macros. This array is used to update the total count of each event during the lifetime of the task and to compute the cost.
The computation of the cost is performed in line 18. A detailed description is given in Section 4.7.
Once the cost has been computed the scheduler rearranges the ReadyList using two possible different solutions (line 21, 22), as explained in Section 4.8. The list is reordered such that next task to be switched in is the one less penalized by the formulation of the cost.
It's important to notice that since the cost is initialized with the value 0, each of the created tasks are allowed to run in turn in the first round. Once the tasks have entered the Running state once during their lifetime, their next execution is totally ruled by the new policy. After a task has been switched in the scheduler reset the performance counters and make them starting count the events.
Two possible solution can be adopted in counting the events:

1. Saving the value of performance counter when the task is switched out during a context switch, restore them when that task is switched in and saves this values in a proper structure. When the task is switched out again the number of events are computed as the difference of the new values and the ones previously saved.

2. Each time a task is switched in during a context switch the values of the performance counter are not saved, instead they are reset to the 0 value. When the same task is switched out the performance counter are stopped and their value are read.

Of the two possible solutions the second one has been adopted.

```
void vTaskSwitchContext( void )
{
        ...

        xYieldPending = pdFALSE;
        traceTASK_SWITCHED_OUT();

        ...

        List_t * pxList = &(pxReadyTasksLists[pxCurrentTCB->
    uxPriority]);

        if (strcmp(pxCurrentTCB->pcTaskName, configIDLE_TASK_NAME)!=0
     && (pxCurrentTCB->xStateListItem.pxContainer == pxList) )
        {
                unsigned int values[pxCurrentTCB->NumActivePCRs];
                    vPerfStopCounting();

                    vPerfReadAllValues(values);
                    for(unsigned int i=0; i<pxCurrentTCB->
    NumActivePCRs; i++){
                        pxCurrentTCB->PCRsValue[i] += values[i];
                    }
            #if(configeUSE_PCER_SCHEDULING_POLICY == 1)
                pxCurrentTCB->cost = usTaskComputeCostMalus(
    pxCurrentTCB->PCRsValue, pxCurrentTCB->PCRsNum,  pxCurrentTCB->
    NumActivePCRs);

                if ((pxList->uxNumberOfItems)>1)
                {
                    vListInsertByCost1(pxList ,&(pxCurrentTCB->
    xStateListItem));
                    //vListInsertByCost2(pxList ,&(pxCurrentTCB->
    xStateListItem));
                }
            #endif
        }

        taskSELECT_HIGHEST_PRIORITY_TASK();
        traceTASK_SWITCHED_IN();

```

```
            ...


            if (strcmp(pxCurrentTCB->pcTaskName, configIDLE_TASK_NAME)
    !=0)
                vPerfResetandStartCounting();
        }

}
```

Listing 4.9: vTaskSwitchContext body function changes

4.6 Identification of the HPCs

In this section is explored the identification of the events occurring during the execution of the task sutibale in the formulation of the cost function. From the PCCRs listed in Table 2.3 have been selected: `JR_STALL`, `LD_STALL`, `I_MISS`, `JUMP`, `BRANCH`, `BRANCH_TAKEN` and `RVC`. These registers counts events having bad impact on the performance of a task in terms of execution time.

The `JR_STALL` counts the number of jump instructions which introduce a stall due to a Read after Write. This happens when there is a conditional branch where two operands need to be compared before making the decision, but one of the operand need to be written yet by a previous instruction. Typically, this operand is a register from which the instruction reads the value it is holding.

The `LD_STALL` counts the number of data hazard encountered when a load instruction is executed. Typically this happens when a Read-After-Write takes place, so when the load instruction is trying to read a register which have to be written by a previous instruction. To avoid reading a wrong value stalls are introduced in the pipeline.

The `I_MISS` counts the number of cycles wasted due to non-ideal caching of the instructions in memory. The cycle wasted refers to the fetch phase of the pipeline.

The `BRANCH` and the `BRANCH_TAKEN` are used to derive the number of untaken branches by computing the difference. In a pipelined processor the next instruction is fetched before the previous ones have been completed. This means that the CPU fetches the instruction of the branch as if it is taken before it is aware about taking it or not. An untaken branch translates in the next sequential instruction who has been fetched to be aborted, resulting in a waste of CPU time.

The JUMP register counts the number of unconditional jump family instructions executed. This leads to the same problem mentioned with the BRANCH and the BRANCH_TAKEN events.

Finally the RVC counts the number of compressed instructions executed. If the hardware has not been designed correctly the expansion of the instructions during the decode phase can lead to important overhead, thus drops in performance.

4.7 Cost's formulation

The cost, one for each task, is a parameter used by the scheduler to reorder the ReadyList, thus crucial in the choice of the next task to pick up for execution after a tick interrupt has occurred.
The formulation used in this work sees the cost as a penalty parameter for the task. The task associated to a lower cost will be the next one to be picked up from the ReadyList by the scheduler during a context switch.
The cost is computed as the sum of the weighted events identified in Section 4.6.

$$\begin{aligned} cost = W_1 * \texttt{JMP\_STALL\_EVENTS} + W_2 * \texttt{LD\_STALL\_EVENTS} \\ +W_3 * \texttt{IMISS\_EVENTS} + W_4 * \texttt{BRANCH\_EVENTS} \\ -W_4 * \texttt{TAKEN\_BRANCH} + W_5 * \texttt{JUMP\_EVENTS} \\ +W_6 * \texttt{RVC\_EVENTS} \end{aligned} \tag{4.1}$$

The weights associated to the events must follow this restriction :

$$\sum_{i=1}^{7} W_i = 1$$

The value of the weights must be set correctly by hand in the configuration header file. If one of the selected performance counter registers in Section 4.6 is disabled by the macro the cost will not include it automatically, so attention must be paid in updating the weights according to the restriction.
In Listing 4.10 is shown how the cost is computed after the proper resources has been identified while in Listing 4.11 is shown the definition of the weights in the FreeRTOS.h kernel configuration source header file.

```
unsigned int usTaskComputeCostMalus(unsigned int *PCRValues, unsigned
    int *NumPCR, unsigned int n){
   unsigned int cost = 0;

```

```
    for(unsigned int i=0;i< n;i++){
            switch (NumPCR[i])
            {
            case CSR_PCER_JMP_STALL:
                cost += W1 * PCRValues[i];
            break;
            case CSR_PCER_LD_STALL:
                cost += W2 * PCRValues[i];
            break;
            case CSR_PCER_IMISS:
                cost += W3 * PCRValues[i];
            break;
            case CSR_PCER_BRANCH:
                cost += W4 * PCRValues[i];
            break;
            case CSR_PCER_TAKEN_BRANCH:
                cost -= W4 * PCRValues[i];
            break;
            case CSR_PCER_JUMP:
                cost += W5 * PCRValues[i];
            break;
            case CSR_PCER_RVC:
                cost += W6 * PCRValues[i];
            break;
            default:
            break;
            }
        }
    cost = cost/60;


    return  cost;
}
```

Listing 4.10: usTaskComputeCostMalus function body

```

#define JMP_STALL 10
#define LD_STALL 10
#define IMISS 10
#define BRANCH_UNTAKEN 10
#define JUMP 10
#define RVC 10

#define W1 JMP_STALL
#define W2 LD_STALL
#define W3 IMISS
#define W4 BRANCH_UNTAKEN
```

```
#define W5 JUMP
#define W6 RVC
```

Listing 4.11: Weights definition in FreeRTOS.h

A new cost is computed in every time windows which start from the task being switched in and the moment in which the same task it is switched out.

4.8 Ready List Rearrangement

In this section is explored the way in which the priority list, used by the scheduler to choose the task to be switched in during a context switch, is reordered. Two implementation as been developed. In the first solution the rearrangement is performed by calling the vListInsertByCost1 function placed in the list.c source kernel file. This function order the task in the ReadyList in increasing order of cost. This means the task which have a lower cost is always placed as the "first" item and will be the one switched in during a context switch. In Listing 4.12 is showed the implementation. In the following section this implementation is denoted as 'Solution 1'.
However, this solution can lead to an important issue. If there is a task for which the calculated cost in a context switch is always the lowest among all, then it will be always the one selected by the scheduler and executes until the task itself is deleted. For this reason another approach has been followed and implemented by the function vListInsertByCost2, listed in Listing 4.13. In the following section this implementation is denoted as 'Solution 2'.
This method sees tasks executing in rounds. In each round every task in the ReadyList are executed once and the costs computed in each round determines the order of execution for the next round, placing them always in crescent order of cost. To keep trace of how many tasks have been executed during a round a counter has been associated to the ReadyList. The counter allow to place the task to be rescheduled in the list after those tasks which haven't been executed yet in the actual round . Every time the counter reaches the number of items in the list it means the round has finished (all the tasks performed their execution in that round), thus the counter is reset to 0.

```

void vListInsertByCost1(List_t * const pxList, ListItem_t * const
    pxNewListItem ) {

    listTEST_LIST_INTEGRITY( pxList );
    listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );


    ListItem_t *  pxIndexEnd = (ListItem_t *) &(pxList->xListEnd);
    uxListRemove(pxNewListItem);


    if(usTaskGetCost(pxIndexEnd->pxPrevious->pvOwner)<= usTaskGetCost
   (pxNewListItem->pvOwner)){
        vListInsertBeforeEndMarker(pxList, pxNewListItem);
    }
    else{
        ListItem_t * pxIterator;
        for( pxIterator = pxIndexEnd; usTaskGetCost(pxIterator->
   pxNext->pvOwner) <= usTaskGetCost(pxNewListItem->pvOwner);
   pxIterator = pxIterator->pxNext );
        pxNewListItem->pxNext = pxIterator->pxNext;
        pxNewListItem->pxPrevious = pxIterator;

        pxIterator->pxNext->pxPrevious = pxNewListItem;
        pxIterator->pxNext = pxNewListItem;

        pxNewListItem->pxContainer = pxList;
        ( pxList->uxNumberOfItems )++;
    }
}
```

Listing 4.12: vListInsertByCost1 body function

```

void vListInsertByCost2(List_t * const pxList, ListItem_t * const
   pxNewListItem ){


    BaseType_t found = pdFALSE;

    listTEST_LIST_INTEGRITY( pxList );
    listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );

    pxList->pxIndex = (ListItem_t *) &(pxList->xListEnd);


    if ((pxList->counter) == (pxList->uxNumberOfItems))
        (pxList->counter) == 0;

    uxListRemove(pxNewListItem);
    pxList->pxIndex = (ListItem_t *) &(pxList->xListEnd);

    if ((pxList->uxNumberOfItems) == 2)
    {
        vListInsertBeforeEndMarker(pxList, pxNewListItem);
    }
    else{
        if (pxList->counter == 0){
            vListInsertBeforeEndMarker(pxList, pxNewListItem);
        }
        else{
            ListItem_t * pxIter;
            ListItem_t * const pxIndexEnd = (ListItem_t *) &(pxList->xListEnd);
            if(usTaskGetCost(pxIndexEnd->pxPrevious->pvOwner) <= usTaskGetCost(pxNewListItem->pvOwner)){
                vListInsertBeforeEndMarker(pxList, pxNewListItem);
            }
            else{
                pxIter = pxIndexEnd;
                for(unsigned int i=0;i<pxList->counter;i++){
                    pxIter = pxIter->pxPrevious;
                }
                if (usTaskGetCost(pxIter->pvOwner) >= usTaskGetCost(pxNewListItem->pvOwner))
                {
                    pxNewListItem->pxNext = pxIter;
                    pxNewListItem->pxPrevious = pxIter->pxPrevious;

                    pxIter->pxPrevious->pxNext = pxNewListItem;
                    pxIter->pxPrevious = pxNewListItem;

                    pxNewListItem->pxContainer = pxList;
                    ( pxList->uxNumberOfItems )++;
                }
                else{
                    for(;usTaskGetCost(pxIter->pxNext->pvOwner) > usTaskGetCost(pxNewListItem->pvOwner);pxIter=pxIter->pxNext){};

                    pxNewListItem->pxNext = pxIter->pxNext;
                    pxNewListItem->pxPrevious = pxIter;

                    pxIter->pxNext->pxPrevious = pxNewListItem;
                    pxIter->pxNext = pxNewListItem;

                    pxNewListItem->pxContainer = pxList;
                    ( pxList->uxNumberOfItems )++;

                }
            }
        }
        (pxList->counter)++;
```

```
            if((pxList->counter) == (pxList->uxNumberOfItems)){
                (pxList->counter) = 0;
            }
        }
}
```

Listing 4.13: vListInsertByCost2 body function

Chapter 5

Results and performance analysis

5.1 Benchmarks selection

The analysis of the performance reached by the new implementation of the scheduler have been carried out with the benchmarks offered by the Tacle-Bench repository. It has been used this collection of benchmarks because as reported in [14] the programs have been developed by different research groups, they don't rely on system-specific header files through #include directives or on an operating system for their execution and are written in C. The input data for all benchmarks is embedded directly within the C source code itself. Additionally, any functions that might be needed from mathematical libraries are provided within the C source code as well. All these features make this collection suitable for applications in general embedded or bare-bone systems.
Among all the benchmarks, priority in the selection was given to the ones picked up from MiBench repository [15], specifically built for the embedded domain. The selected benchmarks have been divided in two groups. The first group, showed in Table 5.1, is a collection of tasks which should be rewarded by the implementation of the new scheduler. Tasks belonging to this category are called 'Critical'.

Name	Description	Code Size	Origin
`rijndael_dec`	Rijndael AES decryption	820	MiBench
`rijndael_enc`	Rijndael AES encryption	734	MiBench
`lift`	A lift controller	361	MartinSchoeberl[16]
`bitonic`	Bitonic sorting network	117	MiBench

Table 5.1: Group 1 : Critical tasks

On the contrary, the second group identifies the 'Non-Critical' tasks which should be penalized. The tasks belonging to this group are listed in Table 5.2.

Name	Description	Code Size	Origin
`gsm_enc`	GSM provisional standard encoder	1491	MediaBench[17]
`minver`	Floating point matrix inversion	141	SNU-RT [18]
`dijkstra`	All pairs shortest path	117	MiBench

Table 5.2: Group 2 : Non-critical tasks

From these benchmark tests several use cases have been defined. The performance analysis has been performed over them. Each use case is composed of a different set or subset of tasks belonging from the Critical and Non-critical group, thus adding diversity and making the results more valuable in the analysis. The use cases are reported in Table 5.3.

USE CASES				
	Use Case 1	Use Case 2	Use Case 3	Use Case 4
rijndael_dec	✓		✓	✓
rijndael_enc	✓	✓		✓
lift	✓	✓		✓
bitonic	✓	✓		✓
gsm_enc	✓		✓	
dijkstra	✓	✓	✓	✓
minver	✓	✓	✓	

Table 5.3: Benchmarks for each use case

5.2 Benchmarks environment set up

This section describes how the environment has been set up to run the benchmarks. Each benchmark is executed in FreeRTOS as a standalone task. The FreeRTOS created tasks execute a function consisting of a loop iterated N times, where in each iteration is called the main function of the benchmark you want to execute. When the loop ends the task delete itself calling the API function vTaskDelete passing the NULL parameter. To make the main function of the test benchmarks available you need to export the main benchmark function in your main file and all the source *.c and *.h files must be linked by the compiler exploiting the makefile. In Listing 5.1 is shown an example.

```c

#include <FreeRTOS.h>
#include <task.h>

/* c stdlib */
#include <stdio.h>

#include "system.h"
#include "timer_irq.h"

extern void mainBitCount();

void vApplicationMallocFailedHook( void );
void vApplicationIdleHook( void );
void vApplicationStackOverflowHook( TaskHandle_t pxTask, char *
    pcTaskName );
void vApplicationTickHook( void );

#define N 100

```

```

void taskBitonic(void * pvParameters){

    unsigned int time = xTaskGetTickCount();
    for ( unsigned int i = 0; i<N ;i++ )
    {
        mainBitonic();
    }
    time = xTaskGetTickCount() - time;
    printf("Bitonic time  : %u\n", time);
    vTaskDelete(NULL);
}

int main( void )
{
    prvSetupHardware();

    xTaskCreate(taskBitonic,
                "Bitonic",
                configMINIMAL_STACK_SIZE*2,
                (void *)NULL,
                configMAX_PRIORITIES-1,
                NULL);
    vTaskStartScheduler();
    return 1;
}
```

Listing 5.1: c source code example

Once the simulated FreeRTOS is launched it will never stop even if all the created tasks are delete, because there will be always at least one task executing: the IDLE task. To force the exiting from the execution of the OS a call to xPortEndScheduler() is performed once the IDLE task is switched in. This event occurs only after all tasks have finished executing their function, thus after they have been all deleted.
The selected benchmarks have different size in therms of line of code. This translates in a significant difference in lifetime execution which can produce unreliable results. The execution time of each task has been made less or more equal with a margin of error by fixing the number N of iteration in each task function of a multiplication factor (see Table 5.4).
Additionally, in the vTaskDelete API function it has been added some code which performs the following operations : first it update the total count of events and second prints their value on stdout.

Name	Multiplication Factor
`rijndael_dec`	3
`rijndael_enc`	3
`lift`	25.5
`bitonic`	800
`gsm_enc`	4
`dijkstra`	1/2
`minver`	525

Table 5.4: Multiplication factors of tasks' loop

```c

void vTaskDelete( TaskHandle_t xTaskToDelete )
{
    TCB_t *pxTCB;
    taskENTER_CRITICAL();
    {
       pxTCB = prvGetTCBFromHandle( xTaskToDelete );

       unsigned int values[pxTCB->NumActivePCRs];
       if (pxTCB == pxCurrentTCB){
              vPerfStopCounting();
            vPerfReadAllValues(values);
            for(unsigned int i=0; i<pxTCB->NumActivePCRs; i++){
                pxTCB->PCRsValue[i] += values[i];
            }
       }
       vTaskPrintPerformanceValue(pxTCB->PCRsValue, pxTCB->PCRsNum, pxTCB->NumActivePCRs);

    ...

}
```

Listing 5.2: c vTaskDelete added line code

In Listing 5.3 is shown the custom application configuration used to obtain the results.

```
#include <stddef.h>
#ifdef __PULP_USE_LIBC
    #include <assert.h>
#endif

#if defined( __GNUC__ )
    #include <stdint.h>
#endif

#define configCLINT_BASE_ADDRESS          0
#define configUSE_PREEMPTION              1
#define configUSE_TIME_SLICING            1
#define configUSE_IDLE_HOOK               1
#define configUSE_TICK_HOOK               1
#define configCPU_CLOCK_HZ                DEFAULT_SYSTEM_CLOCK
#define configTICK_RATE_HZ                ( ( TickType_t ) 100 )
#define configMAX_PRIORITIES              ( 3 )
#define configMINIMAL_STACK_SIZE          ( ( unsigned short ) 256 )
#define configAPPLICATION_ALLOCATED_HEAP 1
#define configTOTAL_HEAP_SIZE             ( ( size_t ) ( 128 * 1024 )
    )
#define configMAX_TASK_NAME_LEN           ( 20 )
#define configUSE_TRACE_FACILITY          1
#define configUSE_16_BIT_TICKS            0
#define configIDLE_SHOULD_YIELD           0
#define configUSE_MUTEXES                 1
#define configQUEUE_REGISTRY_SIZE         8
#define configCHECK_FOR_STACK_OVERFLOW    2
#define configUSE_RECURSIVE_MUTEXES       1
#define configUSE_MALLOC_FAILED_HOOK      1
#define configUSE_APPLICATION_TASK_TAG    0
#define configUSE_COUNTING_SEMAPHORES     1
#define configGENERATE_RUN_TIME_STATS     0

#define configUSE_NEWLIB_REENTRANT 1
#define configUSE_CO_ROUTINES           0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

#define configUSE_TIMERS                0
#define configTIMER_TASK_PRIORITY       ( configMAX_PRIORITIES - 1 )
#define configTIMER_QUEUE_LENGTH        4
#define configTIMER_TASK_STACK_DEPTH    ( configMINIMAL_STACK_SIZE )

#ifndef uartPRIMARY_PRIORITY
    #define uartPRIMARY_PRIORITY        ( configMAX_PRIORITIES - 3 )
#endif

#define INCLUDE_vTaskPrioritySet            1
#define INCLUDE_uxTaskPriorityGet           1
```

```
#define INCLUDE_vTaskDelete                 1
#define INCLUDE_vTaskCleanUpResources       1
#define INCLUDE_vTaskSuspend                1
#define INCLUDE_vTaskDelayUntil             1
#define INCLUDE_vTaskDelay                  1
#define INCLUDE_eTaskGetState               1
#define INCLUDE_xTimerPendFunctionCall      0
#define INCLUDE_xTaskAbortDelay             1
#define INCLUDE_xTaskGetHandle              1
#define INCLUDE_xSemaphoreGetMutexHolder    1

#ifdef __PULP_USE_LIBC
    #define configASSERT( x ) assert ( x )
#else
    #define configASSERT( x ) do { if( ( x ) == 0 ) { 
   taskDISABLE_INTERRUPTS(); for( ;; ); } } while ( 0 )
#endif
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
#define configKERNEL_INTERRUPT_PRIORITY 7
#define configeUSE_PCER_SCHEDULING_POLICY 1
#endif /* FREERTOS_CONFIG_H */
```

Listing 5.3: FreeRTOSConfig.h header file

The process of data collection has been automatized with a bash script and a python one. The bash script create the txt file passed as parameter and in a loop repeated 15 times run an instance of GVSoC through the makefile waiting the process to finish. All the output printed during each execution of GVSoC is redirected and appended to the created .txt file. The python script instead runs the bash script waiting its end and then process the .txt file, passed as a parameter in the command line when launching the script, filtering the revelant information.

5.3 Results and analysis

This section explores the results obtained by this work and cover an analysis on them. The turnaround time and the total count of the events identified in Section 4.6 are the metrics target of the analysis. The main focus is given to the turnaround time since the goal of the work is to speed up the performance of the OS.
By taking a look at the log .txt file it has been noticed that in each of the 15 repetition the value of the turnaround time and the final count of the values stored in the performance counter registers are always the same. This is possible since the scheduler of a real-time system is deterministic. Thus, it is useless to investigate further in the analysis performing measures like the average, the median, the variance and the standard deviation.
In Figures 5.1, 5.2, 5.3, 5.4 is represented the turnaround time of tasks in each

use case for all the implementations, the default one and the news created for this work.
The 'Solution 1' solution brings a big improvement to the performance of the system. The task belonging to the Critical group, thus the one which should be rewarded by this solution, finish their execution earlier with respect to the default implementation: their turnaround time is improved up to almost 64%. For what concern the Non-Critical tasks we can see that in 5.1, 5.2, 5.3, there is a small percentage of improvement with an exception for the minver task having a turnaround time slightly higher. In 5.4 instead the Non-Critical task `dijkstra`, has performed imperceptibly worse with respect the the default scheduling implementation.
Even if few tasks had a drop of the performance in terms of turnaround time, we can see from Figure 5.5 that the total execution time (the sum of the turnaround time of each task) in each use case is lower compared with the 'Default' solution. More specifically in Use Case 1 we have an upgrade of 23%, in Use Case 2 it is of 22%, in Use Case 3 is of 15% and finally for Use Case 4 we have a 21% improvement in terms of total turnaround time.
Tables 5.5, 5.6, 5.7, 5.8 shows the total costs calculated for each task during its lifetime in each scenario. From Table 5.9 to Table 5.29 are reported the total value of the HPCs for each task in each use case. For the two new solutions adopted the total cost is generally higher. This behaviour is possible since the way in which the task are scheduled is different and all the environment is simulated. However based on the cost of its relative implementation, 'Solution 1' solution demonstrates an upgrade in performance because a task producing a low cost doesn't have to wait to be executed again due to the time slicing property of the default algorithm. They way in which the priority list is reordered allows the turnaround time of the rewarded tasks to be lower compared to the original solution because a task will be the only one executing until the moment a tick interrupt occurs the computed cost is lower than the other tasks' one.
For what concern the 'Solution 2' solutions we can see the turnaround time is imperceptibly higher with respect to the Default one. This implementation avoids a task to wait indefinitely due to a cost of some other tasks which is always lower, but the way in which the list is reordered and the way in which the next task to be switched is chosen doesn't produce an effect considerably different from the original solution. For this reason the results obtained are practically identical, indeed the Default one is preferable.

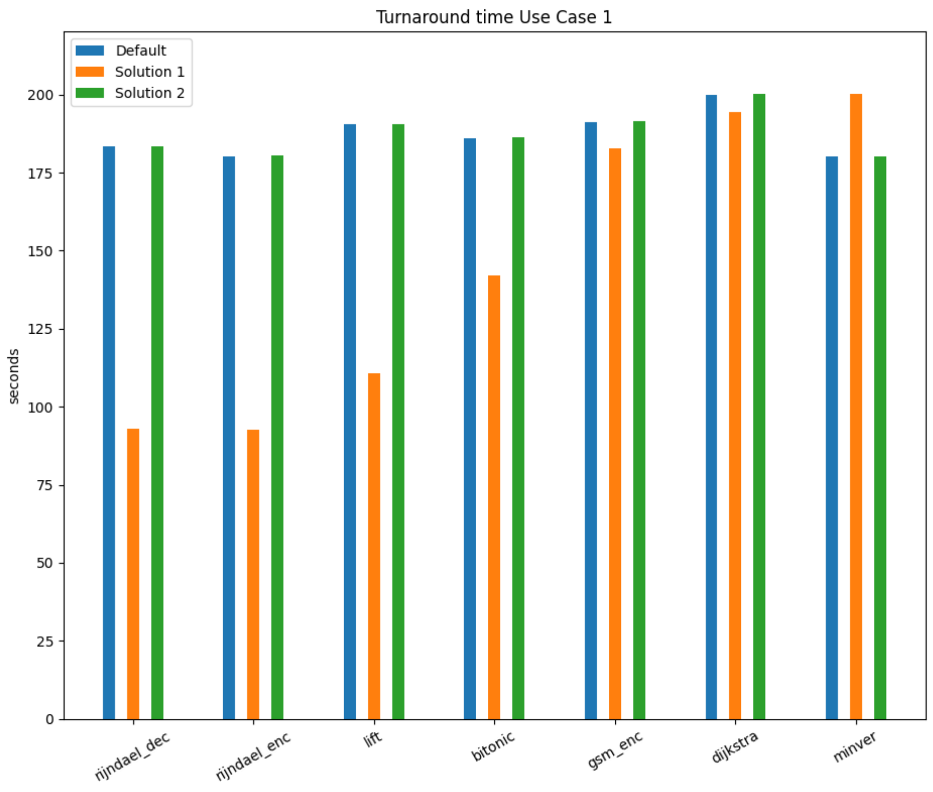

Figure 5.1: Turnaround time use case 1

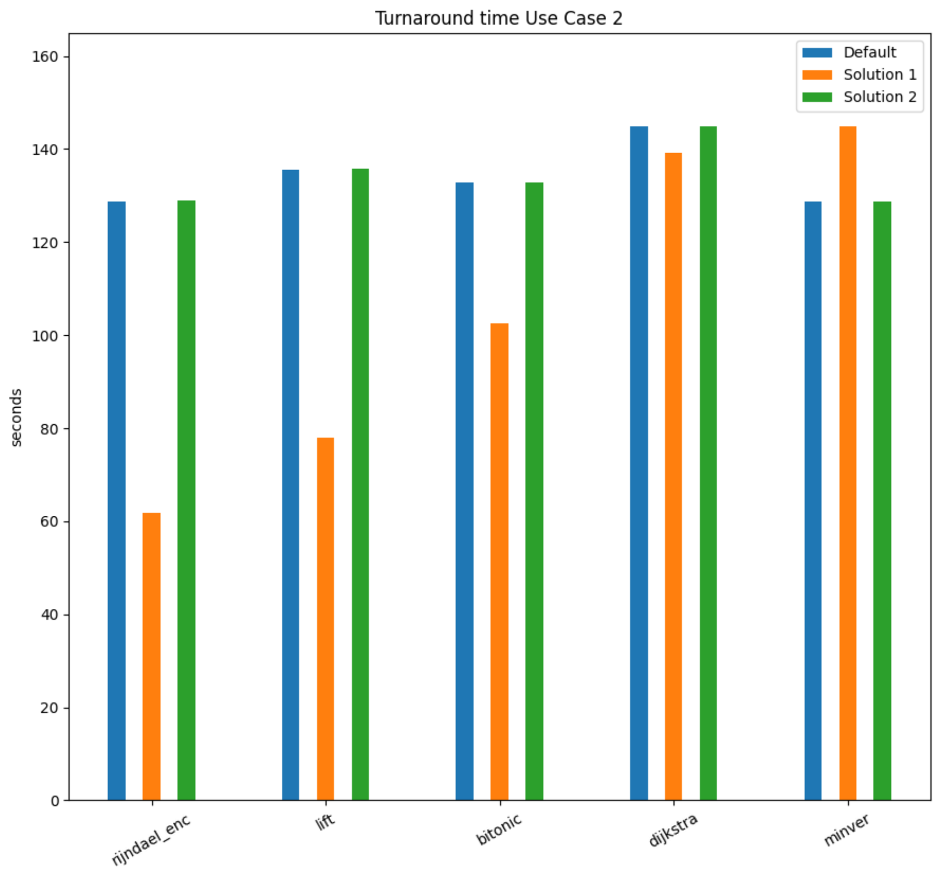

Figure 5.2: Turnaround time use case 2

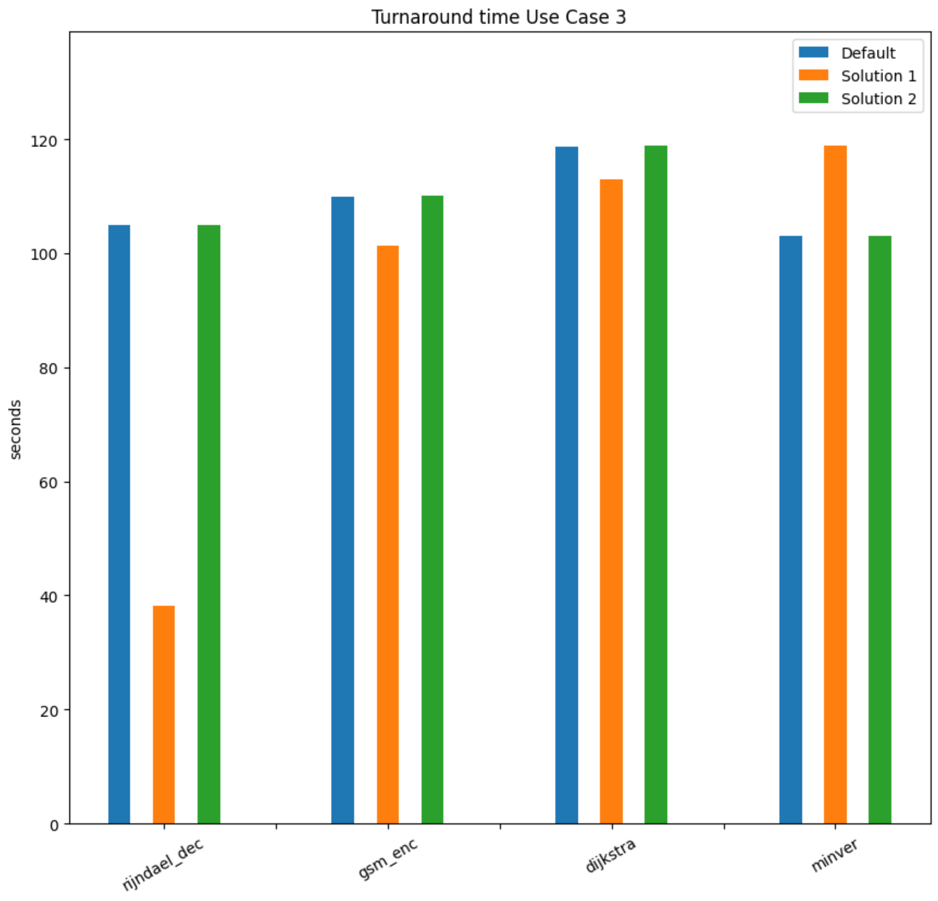

Figure 5.3: Turnaround time use case 3

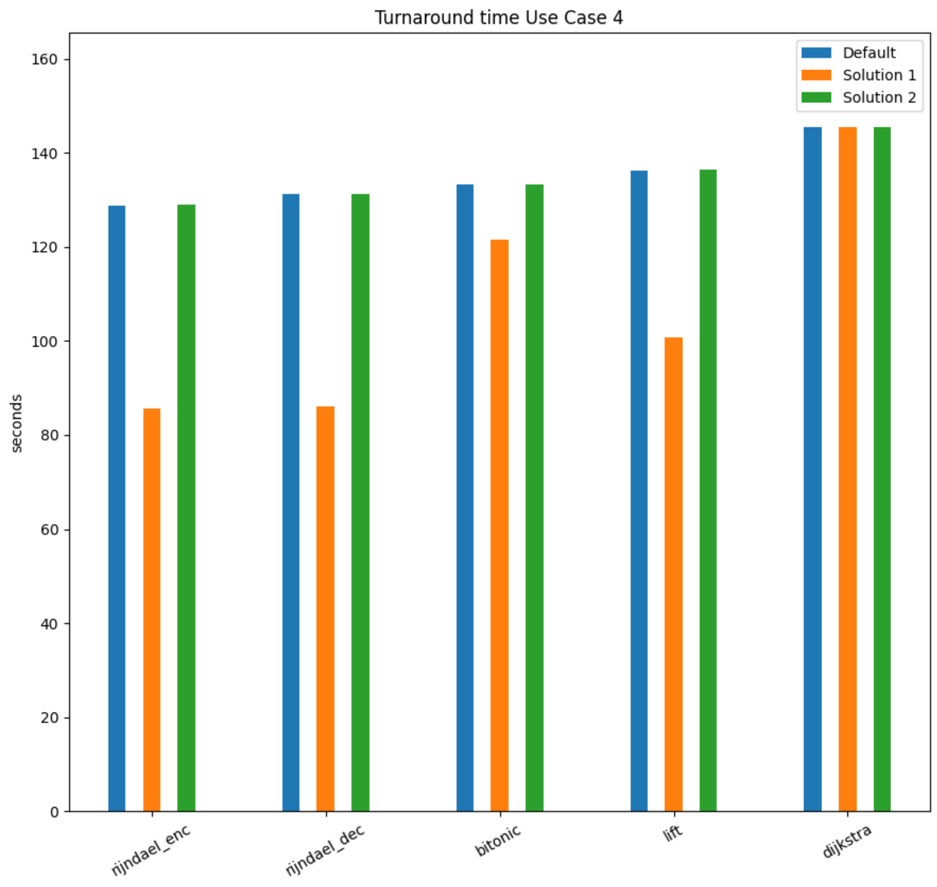

Figure 5.4: Turnaround time use case 4

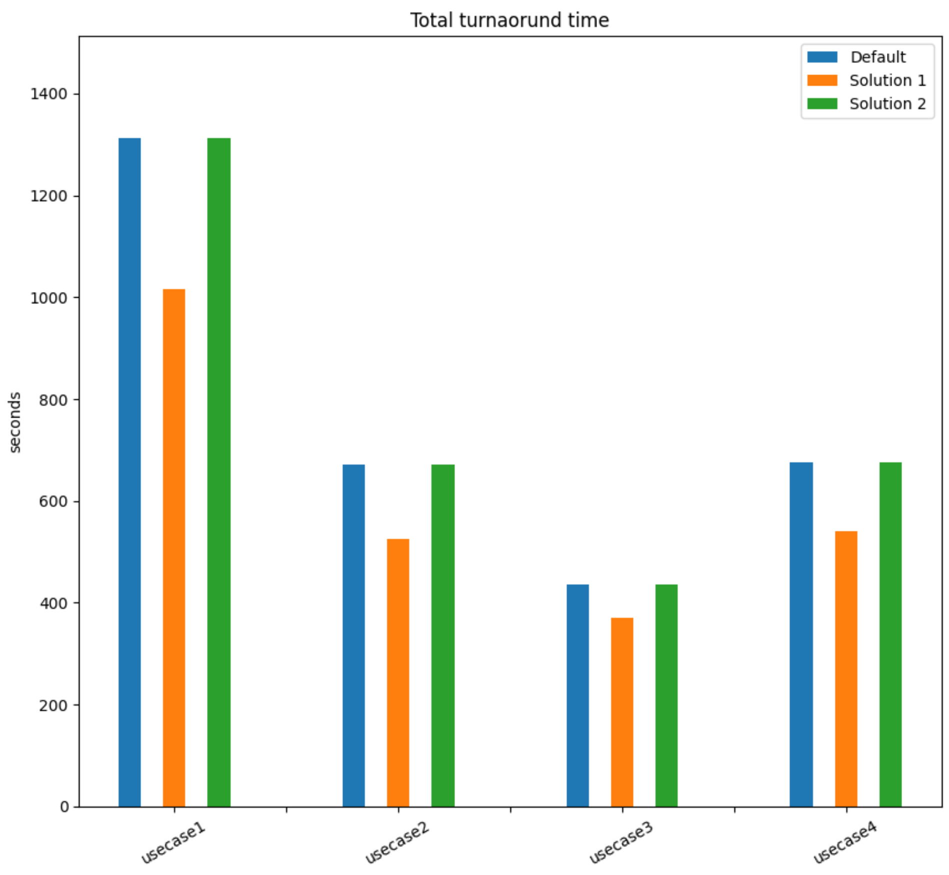

Figure 5.5: Total turnaround time

Use case 1 : Total cost

	Default	Solution 1	Solution 2
`rijndael_dec`	50102396	50102489	50102520
`rijndael_enc`	49733203	49733234	49733327
`lift`	71823527	71823603	71823603
`bitonic`	146649573	146649683	146649683
`gsm_enc`	320584630	320597411	320575863
`dijkstra`	401741839	401764781	401741953
`minver`	518833045	518818089	518833169

Table 5.5: Use case 1 total cost

Use case 2 : Total cost

	Default	Solution 1	Solution 2
`rijndael_enc`	49733203	49733234	49733327
`lift`	71823417	71823448	71823603
`bitonic`	146649573	146649697	146649635
`dijkstra`	401740850	401764780	401740912
`minver`	518833045	518818125	518833169

Table 5.6: Use case 2 total cost

Use case 3 : Total cost			
	Default	Solution 1	Solution 2
`rijndael_dec`	50102397	50102428	50102521
`gsm_enc`	320604347	320591464	320581255
`dijkstra`	401741886	401764842	401741952
`minver`	518833045	518818053	518833169

Table 5.7: Use case 3 total cost

Use case 4 : Total cost			
	Default	Solution 1	Solution 2
`rijndael_enc`	49733251	49733282	49733282
`rijndael_dec`	50102459	50102428	50102569
`lift`	71823478	71823602	71823602
`bitonic`	146649511	146649697	146649635
`dijkstra`	401740863	401702451	401742001

Table 5.8: Use case 4 total cost

Use case 1 : rijndael_enc			
	Default	Solution 1	Solution 2
`LD_STALL`	21825355	21825371	21825381
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	24814822	24814841	24814850
`BRANCH`	32544186	32544219	32544238
`BRANCH_TAKEN`	29503933	29503946	29503955
`RVC`	52773	52749	52813
`Total cost`	49733203	49733234	49733327

Table 5.9: Use case 1 : rijndael_enc HPCs total values

Use case 1 : rijndael_dec

	Default	Solution 1	Solution 2
`LD_STALL`	22196934	22196963	22196960
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	24683362	24683395	24683390
`BRANCH`	33354246	33354305	33354298
`BRANCH_TAKEN`	30186119	30186143	30186141
`RVC`	53973	53969	54013
`Total cost`	50102396	50102489	50102520

Table 5.10: Use case 1 : rijndael_dec HPCs total values

Use case 1 : bitonic

	Default	Solution 1	Solution 2
`LD_STALL`	32115337	32115353	32115353
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	84838126	84838147	84838147
`BRANCH`	71910742	71910778	71910778
`BRANCH_TAKEN`	42269973	42269989	42269989
`RVC`	53973	55394	55394
`Total cost`	146649573	146649683	146649683

Table 5.11: Use case 1 : bitonic HPCs total values

Use case 1 : lift

	Default	Solution 1	Solution 2
`LD_STALL`	28117845	28117868	28117868
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	30714134	30714155	30714155
`BRANCH`	133186073	133186115	133186115
`BRANCH_TAKEN`	120354846	120354863	120354863
`RVC`	160321	160328	160328
`Total cost`	71823527	71823603	71823603

Table 5.12: Use case 1 : lift HPCs total values

Use case 1 : gsm_enc

	Default	Solution 1	Solution 2
`LD_STALL`	276981220	276981295	276981118
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	28867393	28881800	28858424
`BRANCH`	78262052	78257253	78264635
`BRANCH_TAKEN`	63585048	63582010	63587367
`RVC`	59013	59073	59053
`Total cost`	320584630	320597411	320575863

Table 5.13: Use case 1 : gsm_enc HPCs total values

Use case 1 : minver

	Default	Solution 1	Solution 2
`LD_STALL`	13762324	13760030	13762350
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	38649804	38648092	38649832
`BRANCH`	103465562	103458654	103465614
`BRANCH_TAKEN`	50612149	50610431	50612171
`RVC`	413567504	413561744	413567544
`Total cost`	518833045	518818089	518833169

Table 5.14: Use case 1 : minver HPCs total values

Use case 1 : dijkstra

	Default	Solution 1	Solution 2
`LD_STALL`	245174464	245177997	245174485
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	4543350	4546004	4543370
`BRANCH`	320599728	320610314	320599778
`BRANCH_TAKEN`	168643552	168646203	168643569
`RVC`	67849	76669	67889
`Total cost`	401741839	401764781	401741953

Table 5.15: Use case 1 : dijkstra HPCs total values

Use case 2 : dijkstra

	Default	Solution 1	Solution 2
LD_STALL	245174311	245177996	245174324
JR_STALL	0	0	0
I_MISS	0	0	0
JUMP	4543236	4546004	4543250
BRANCH	320599272	320610314	320599298
BRANCH_TAKEN	168643438	168646203	168643449
RVC	67469	76669	67489
Total cost	401740850	401764780	401740912

Table 5.16: Use case 2 : dijkstra HPCs total values

Use case 2 : minver

	Default	Solution 1	Solution 2
LD_STALL	13762324	13760039	13762350
JR_STALL	0	0	0
I_MISS	0	0	0
JUMP	38649804	38648103	38649832
BRANCH	103465562	103458668	103465614
BRANCH_TAKEN	50612149	50610439	50612171
RVC	413567504	413561754	413567544
Total cost	518833045	518818125	518833169

Table 5.17: Use case 2 : minver HPCs total values

Use case 2 : rijndael_enc

	Default	Solution 1	Solution 2
LD_STALL	21825355	21825371	21825381
JR_STALL	0	0	0
I_MISS	0	0	0
JUMP	24814822	24814841	24814850
BRANCH	32544186	32544219	32544238
BRANCH_TAKEN	29503933	29503946	29503955
RVC	52773	52749	52813
Total cost	49733203	49733234	49733327

Table 5.18: Use case 2 : rijndael_enc HPCs total values

Use case 2 : bitonic

	Default	Solution 1	Solution 2
`LD_STALL`	32115337	32115363	32115350
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	84838126	84838154	84838140
`BRANCH`	71910742	71910794	71910768
`BRANCH_TAKEN`	42269973	42269995	42269984
`RVC`	55341	55381	55361
`Total cost`	146649573	146649697	146649635

Table 5.19: Use case 2 : bitonic HPCs total values

Use case 2 : lift

	Default	Solution 1	Solution 2
`LD_STALL`	28117829	28117845	28117868
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	30714113	30714132	30714155
`BRANCH`	133186037	133186070	133186115
`BRANCH_TAKEN`	120354830	120354843	120354863
`RVC`	160268	160244	160328
`Total cost`	71823417	71823448	71823603

Table 5.20: Use case 2 : lift HPCs total values

Use case 3 : minver

	Default	Solution 1	Solution 2
`LD_STALL`	13762324	13760021	13762350
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	38649804	38648081	38649832
`BRANCH`	103465562	103458640	103465614
`BRANCH_TAKEN`	50612149	50610423	50612171
`RVC`	413567504	413561734	413567544
`Total cost`	518833045	518818053	518833169

Table 5.21: Use case 3 : minver HPCs total values

Use case 3 : rijndael_dec

	Default	Solution 1	Solution 2
`LD_STALL`	22196935	22196951	22196961
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	24683362	24683381	24683390
`BRANCH`	33354246	33354279	33354298
`BRANCH_TAKEN`	30186119	30186132	30186141
`RVC`	53973	53949	54013
`Total cost`	50102397	50102428	50102521

Table 5.22: Use case 3 : rijndael_dec HPCs total values

Use case 3 : gsm_enc

	Default	Solution 1	Solution 2
`LD_STALL`	276981259	276981230	276981102
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	28883892	28871707	28862397
`BRANCH`	78258448	78261909	78264035
`BRANCH_TAKEN`	63578265	63582435	63585332
`RVC`	59013	59053	59053
`Total cost`	320604347	320591464	320581255

Table 5.23: Use case 3 : gsm_enc HPCs total values

Use case 3 : dijkstra

	Default	Solution 1	Solution 2
`LD_STALL`	245174466	245178009	245174484
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	4543357	4546018	4543370
`BRANCH`	320599738	320610340	320599778
`BRANCH_TAKEN`	168643557	168646214	168643569
`RVC`	67882	76689	67889
`Total cost`	401741886	401764842	401741952

Table 5.24: Use case 3 : dijkstra HPCs total values

Use case 4 : rijndael_enc			
	Default	Solution 1	Solution 2
LD_STALL	21825358	21825374	21825381
JR_STALL	0	0	0
I_MISS	0	0	0
JUMP	24814829	24814848	24814850
BRANCH	32544196	32544229	32544238
BRANCH_TAKEN	29503938	29503951	29503955
RVC	52806	52782	52813
Total cost	49733251	49733282	49733327

Table 5.25: Use case 4 : rijndael_enc HPCs total values

Use case 4 : rijndael_dec			
	Default	Solution 1	Solution 2
LD_STALL	22196948	22196951	22196961
JR_STALL	0	0	0
I_MISS	0	0	0
JUMP	24683376	24683381	24683390
BRANCH	33354272	33354279	33354298
BRANCH_TAKEN	30186130	30186132	30186141
RVC	53993	53949	54013
Total cost	50102459	50102428	50102521

Table 5.26: Use case 4 : rijndael_dec HPCs total values

Use case 4 : bitonic			
	Default	Solution 1	Solution 2
LD_STALL	32115324	32115363	32115350
JR_STALL	0	0	0
I_MISS	0	0	0
JUMP	84838112	84838154	84838140
BRANCH	71910716	71910794	71910768
BRANCH_TAKEN	42269962	42269995	42269984
RVC	55321	55381	55361
Total cost	146649511	146649697	146649635

Table 5.27: Use case 4 : bitonic HPCs total values

Use case 4 : lift

	Default	Solution 1	Solution 2
`LD_STALL`	28117841	28117867	28117870
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	30714127	30714155	30714162
`BRANCH`	133186063	133186115	133186125
`BRANCH_TAKEN`	120354841	120354863	120354868
`RVC`	160288	160328	160361
`Total cost`	71823478	71823602	71823650

Table 5.28: Use case 4 : lift HPCs total values

Use case 4 : dijkstra

	Default	Solution 1	Solution 2
`LD_STALL`	245174306	245168393	245174325
`JR_STALL`	0	0	0
`I_MISS`	0	0	0
`JUMP`	4543232	4538793	4543250
`BRANCH`	320599268	320581542	320599298
`BRANCH_TAKEN`	168643435	168638999	168643449
`RVC`	67492	52722	67489
`Total cost`	401740863	401702451	401740913

Table 5.29: Use case 4 : dijkstra HPCs total values

Chapter 6

Conclusion and future works

In this work is presented a new scheduling algorithm to be adopted in real-time operating system which has been developed trying to improve the performance of the OS itself in therms of tasks' execution time. The new implementation exploits the values stored in some special purpose registers called hardware performance counter registers. The values hold by these registers are the occurrences of specific hardware related events. The analysis has been performed on results obtained using benchmarks located in the Tacle-bench Github repository. The data showed that one of the two solution developed actually improved the performance of the OS. Considering the sum of the turnaround time for each task in each selected scenario we have an average improvement of 20.25%.
Even if the solution proposed demonstrates to bring an improvement to the OS it has some gaps. Some possible future works, built upon the actual one, can be to consider some of the events used in the formulation of the cost in relation to others hardware occurrences. An example could be to consider the number of stall as a percentage related to the number of instruction executed and not as an absolute value. Furthermore the analysis can be extended by changing the weights assigned to each event and see how the scheduler performance changes giving more or less importance to specific events. Moreover these weights can be made dynamic. They can be defined as static global variable inside the task.c source kernel file and the scheduler can change their value depending on the type of events occurring more frequently when tasks are executing, potentially adopting some algorithm based on Artificial Intelligence.