ESCOLA POLITÉCNICA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

MESTRADO EM SISTEMAS DE COMUNICAÇÃO

ALINE SCHRÖPFER FRACALOSSI

DEVELOPMENT OF AN INTELLECTUAL-PROPERTY CORE TO DETECT TASK SCHEDULING ERRORS IN RTOS-BASED EMBEDDED SYSTEMS

Porto Alegre

2021

Pontifícia Universidade Católica do Rio Grande do Sul – PUCRS

Graduate Program in Electrical Engineering

DEVELOPMENT OF AN INTELLECTUAL-PROPERTY CORE TO DETECT TASK SCHEDULING ERRORS IN RTOS-BASED EMBEDDED SYSTEMS

ALINE SCHRÖPFER FRACALOSSI

Dissertation presented as partial requirement for obtaining the degree of Master in Electrical Engineering at Pontifícia Universidade Católica do Rio Grande do Sul

Advisor: Prof. Dr. Fabian Luis Vargas

Co-advisor: Prof. Dr. César Augusto Missio Marcon

Porto Alegre

2021

Ficha Catalográfica

F797d   Fracalossi, Aline Schröpfer

Development of an Intellectual-Property Core to detect task scheduling errors in RTOS-Based Embedded Systems / Aline Schröpfer Fracalossi. – 2021.
60.
Dissertação (Mestrado) – Programa de Pós-Graduação em Engenharia Elétrica, PUCRS.

Orientador: Prof. Dr. Fabian Luis Vargas.
Co-orientador: Prof. Dr. César Augusto Missio Marcon.

1. Task scheduler. 2. Real-time operating system (RTOS). 3. Embedded system for critical application. 4. Multicore processor. 5. Fault-tolerance. I. Vargas, Fabian Luis. II. Marcon, César Augusto Missio. III. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

DEVELOPMENT OF AN INTELLECTUAL-PROPERTY CORE TO DETECT TASK SCHEDULING ERRORS IN RTOS-BASED EMBEDDED SYSTEMS

CANDIDATA: ALINE SCHROPFER FRACALOSSI

Esta Dissertação de Mestrado foi julgada para obtenção do título de MESTRE EM ENGENHARIA ELÉTRICA e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica do Rio Grande do Sul.

To my mother Regina, my husband Pedro and my little son Marcelo, for being always with me.

Acknowledgements

First of all, I thank God.

I thank my parents, Regina and Joacir, for always believing in me and for all the support, you make me more strong and confident. Especially my mother, who was always there for me when I needed it most. I also thank my husband Pedro and my son Marcelo, who were my inspiration in the final part of this work.

I thank my advisor, Fabian Vargas, and my co-advisor Cesar Marcon, for all the advice and help during this period. And also to my friends and teachers.

To CAPES for the financial support to carry out this research work.

"It is the time you have wasted for your rose
that makes your rose so important."
(The Little Prince)

Abstract

The employment of a Real-Time Operating System (RTOS) has become an attractive solution for designing critical real-time embedded systems that are part of our daily lives. For these systems, the correct functioning depends not only on the correct logical response, but also on the time at which the answer is given. In this regard, RTOS has emerged as an interesting solution for multiple processing cores. Furthermore, the market pressures to reduce the energy consumption that these multicore embedded systems need to operate. The main consequence of this pressure is the higher susceptibility to transient failures. This type of failure can affect the task scheduling process and change the system correct functioning. In this scenario, it is necessary to have a solution to improve the reliability of the scheduling process. Therefore, this dissertation develops and validates an Intellectual-Property Core (I-IP) able to monitor the Earliest Deadline First (EDF) scheduling algorithm running on a single core system. The I-IP performs passive monitoring of the system's task scheduling to detect failures. Described in Very-High-Speed Integrated Circuits Hardware Description Language (VHDL), the I-IP is connected to the processor address bus to perform system monitoring. The proposed technique was implemented in the HF-RISC softcore processor (namely Hellfire processor), which was running under the control of the HF-RISC Operating System (HellfireOS). Simulation results indicate that the proposed technique effectively detects faults that induce task scheduling malfunctioning at runtime, while incurring acceptable penalties, of low area overhead and negligible energy consumption increase.

Keywords: Task scheduler, Real-time operating system (RTOS), Embedded system for critical application, Multicore processor, Fault-tolerance.

Resumo

Sistemas embarcados críticos fazem cada vez mais parte do nosso dia e devido à essa criticidade, os sistemas operacionais de tempo real (RTOS) tornaram-se uma solução atrativa. Para estes sistemas, o correto funcionamento depende primeiramente do tempo no qual a resposta foi dada e então da resposta lógica correta. Juntamente com eles surgiu a necessidade de vários núcleos de processamento e também a necessidade de reduzir o consumo de energia destes sistemas. Em decorrência disso, o sistema tem maior suscetibilidade a falhas transientes. Basicamente, este tipo de falha pode afetar o escalonamento das tarefas, alterando o correto funcionamento do sistema. Surge então a necessidade de promover uma solução que garanta a confiabilidade do escalonamento das tarefas do sistema. E, esta dissertação aborda o desenvolvimento e validação de um Intellectual-Property Core (I-IP) para monitoramento do algoritmo de escalonamento Earliest Deadline First (EDF). Seu objetivo é detectar falhas no escalonamento de tarefas, devendo realizar o supervisionamento passivo do processo de escalonamento de tarefas. Descrito em Very-High-Speed Integrated Circuits Hardware Description Language (VHDL), o I-IP é conectado ao barramento de endereços de cada processador, para o monitoramento. A técnica prosposta foi implementada no processador softcore HF-RISC (nomeado, processador Hellfire), que estava rodando sob o controle do Sistema Operacional HF-RISC (HellfireOS). Os resultados da simulação indicam que a técnica proposta é muito eficaz para detectar falhas que induzem o mau funcionamento do escalonamento de tarefas enquanto incorrem em penalidades aceitáveis de baixa sobrecarga de área e aumento insignificante do consumo de energia.

Palavras-chaves: Escalonamento de tarefas, Sistemas Operacional de Tempo Real (RTOS), Sistemas embarcados para aplicações críticas, Processador Multicore, Tolerância a falhas.

List of Figures

List of Tables

List of abbreviations and acronyms

CAM	Content-Addressable Memory
Ci	Task Execution time
Di	Task Deadline
EDF	Earliest Deadline First
FSM	Finite State Machine
GSE	Embedded Systems Group
I-IP	Infrastructure-Intellectual Property
ISR	Interrupt Service Routines
LUT	Look Up Table
MIPS	Microprocessor without Interlocked Pipeline Stages
MIT	Minimum Inter Arrival Time
OS	Operating Systems
PC	Program Counter
RISC	Reduced Instruction Set Computing
RM	Rate Monotonic
RR	Round Robin
RTOS	Real Time Operating Systems
SRAM	Static Random Access Memory
Ti	Task Period
VHDL	VHSIC Hardware Description Language
WCET	Worst Case Execution Time

Contents

1 Introduction

Nowadays, various types of applications ranging from simple tasks to highly complex tasks are based on embedded systems. The development of real-time critical applications is a new technological trend within systems. Applications in the automotive and medical areas, air traffic control systems, telecommunications systems, industrial automation, and military systems can be cited as examples of critical applications for real-time embedded systems.

A Real-Time Operating System (RTOS) is defined as a computational system that reacts to external events while respecting exact time conditions. However, the proper behavior of this type of system depends not only on the computational value, but on the time after the results are produced (STANKOVIC; RAMAMRITHAM, 1988). Most safety-critical embedded systems are used for real-time applications, and these systems need to respect stringent timing constraints. According to (IGNAT et al., 2006b), the real-time systems must provide temporally correct results, not only logically correct results.

Single Event Upset (SEU) can cause transient faults in Real-time operating systems affecting the applications that are running (ARLAT et al., 2003)(IGNAT et al., 2006a). Some of the transient faults affecting the RTOS are responsible for generating scheduling dysfunctions and, because of that, could lead to incorrect system behavior (SILVA; BOLZANI; VARGAS, 2011).

Up to now, some solutions have been presented to treat the reliability problems of real-time systems (IZOSIMOV et al., 2005). One of the proposals made in (TARRILO; BOLZANI; VARGAS, 2009) was developed for operating systems with the Round-Robin algorithm without interrupt support. In order to solve this problem, an I-IP capable of manipulating interrupts was proposed in (SILVA; BOLZANI; VARGAS, 2011). However, none of the techniques support preemptive scheduling monitoring, later proposed by (OLIVEIRA; BOLZANI; VARGAS, 2015).

Through experiments, it was found that 34% of the transient faults led the RTOS to malfunction in the task scheduler and an additional 17% resulted in system crashes

(NICOLESCU et al., 2005). And about 21% of these propagated failures lead to application failures (IGNAT et al., 2006b). Because of this high rate of failure that propagates to RTOS and the application resulting in errors, the effort to increase system robustness is validated by detecting failures in the RTOS scheduler.

To detect task scheduling misbehavior, in this Thesis, we present a hardware-based approach. This approach proposes to be monitoring the scheduling of EDF to detect any violations during the scheduling process. To be more specific, the goal is to detect the faults that can shift the tasks' execution flow.

1.1 Goals

This Master's Thesis aims to propose a technique based on hardware to monitor the RTOS execution flow, capable of increasing the robustness of the system to detect the misbehavior of the scheduling. In more detail, the proposed approach provides fault detection in RTOS-based embedded systems that adopt preemptive scheduling, which can change the flow of task execution. The proposed technique is based on implementing an Infrastructure Intellectual-Property (I-IP) to detect failures through real-time monitoring of application tasks.

The main difference between the new I-IP and the one previously proposed in (TARRILO; BOLZANI; VARGAS, 2009), (SILVA; BOLZANI; VARGAS, 2011) and (OLIVEIRA; BOLZANI; VARGAS, 2015), is that the I-IP proposed in this work represents a broader solution since it can monitor a larger set of tasks that would have lost its deadline in previously scheduler algorithms. The present work improves the previous work, providing the I-IP capable of monitoring the scheduler algorithm with dynamic priorities: Earliest Deadline First.

1.2 Structure of this Document

To better present this thesis, we divided this document into four major parts. The first part presents a small introduction. The second part presents the theory behind this work. The third part presents the methodology used to implement the proposal and block diagrams of the architectures. The fourth part presents the test platform and the results obtained after the experiments carried out to analyze and validate the proposed technique, as well as the conclusion of this Master's Work.

2 Theoretical Foundations

Real-time Systems have critical time constraints to perform their tasks, making not only the logical results imperative but also the times at which these results are produced (STANKOVIC, 1988), (MICCO L.; VARGAS, 2020a), (MICCO L.; VARGAS, 2020b). The goal of real-time computing is to create ways to provide predictable temporal behavior in a system.

Essentially, the RTOS can be classified according to the time restrictions in soft and hard. The hard-RTOS represents the systems that have critical time restrictions; if the tasks cannot fail, the time is precious, and the schedule has to respect its deadlines. In contrast, the soft-RTOS can tolerate latencies and respond with decreased service quality (QING; YAO, 2003), (JUHáSZ; PLETL; MOLNAR, 2019).

2.1 Real-Time Operating System

An Operating System (OS) is a program that virtualizes the system's hardware resources. In this way, users using a system resource will be manipulating logical entities, and the OS converts these actions in operations with physical access to the hardware (COSTA, 2010).

The kernel of an operating system is one of the main parts. For example, through OS is that system calls, such as reading and writing to a file, access the hardware. The OS is also responsible for detecting and starting the devices indispensable for the operation of the machine. The main functionalities that an operating system must implement are:

- Process management;
- Memory management;
- Input and output management;
- Synchronization of processes;
- File management.

The performance of an OS has a significant influence on the performance of applications. The system resources simplify the design of real-time applications by offering native mechanisms to manage tasks, concurrency, memory, time as well as interrupts. The efficient use of the CPU is considered the more critical and the more important issue in RTOS (SILVA; BOLZANI; VARGAS, 2011).

As mentioned in (QING; YAO, 2003), an RTOS is a program that schedules the execution of an application at the correct time, manages system resources, and provides a basis for the development of application code, unlike a general-purpose operating system, an RTOS is characterized by having time to complete a task as a fundamental parameter. The kernel provides the basic services for the other parts of the OS. All OSs have a kernel that typically includes memory, process, file management, scheduler, and interrupt controller.

2.1.1 Scheduler

All kernels have a scheduler responsible for the choice of task and the processor context switch. The task scheduling needs allocating resources and time to meet specific performance requirements. The most widely researched topic within real-time systems is scheduling. This is the basic problem of making sure the tasks comply with their deadlines. The use of multicore processors becomes a challenge due to the sharing of different physical resources, such as cache memory (GRACIOLI; FROHLICH; PELLIZZONI, 2013), buses and peripherals (BETTI et al., 2013) and (BOYD-WICKIZER et al., 2010), which make it difficult to estimate the worst-case execution time of applications (ZHURAVLEV et al., 2012).

The scheduler consists of objects and services (Figure 1 adapted from the book Real-Time Concepts (QING; YAO, 2003)). Figure 1 represents two concepts, objects, and services. The objects represent tools that help the designer to develop their applications. The services represent operations that the kernel performs on objects; among these operations are timing, resource management and interrupt handling.

The scheduler efficiency depends not only on the algorithm used but also on the nature of the application, for example, regarding the real-time requirement, that is, an event must be started and completed in a predefined time. The most common

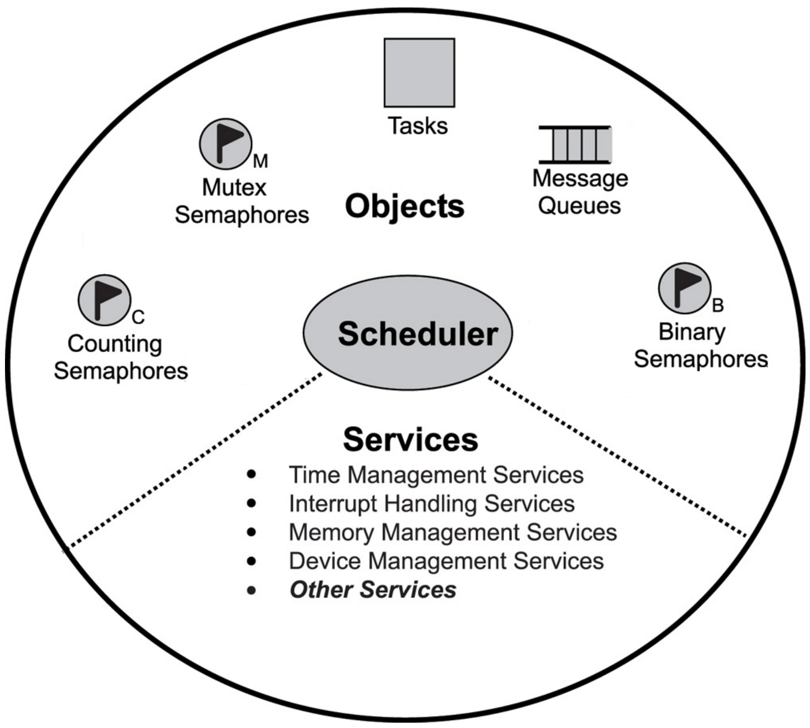


Figure 1 – Scheduler: Common components (source: [QING; YAO, 2003]).

objects in RTOS are the Tasks, Semaphores, and Message Queues. However, there may be other objects such as Timers and Pipes, can be used.

The **objects** in Figure 1 represent the tasks, semaphores, and the message queue, as will be better explained below.

- **Tasks**
  The main goal of the scheduler is to select the tasks to be executed. The tasks are classified according to their periodicity as mentioned in (DAVIS; BURNS, 2011):
  - **Periodic tasks:** have a period, that is, a fixed interval of time in which they execute their computation;
  - **Aperiodic tasks:** not periodic tasks. The tasks do not have definite temporal behavior and can execute at any moment;
  - **Sporadic tasks:** not follow a well-defined period, but have the guarantee that, from the beginning of task execution, the same task will not be

reintroduced for being scheduled for at least one Minimum Inter arrival Time (MIT).

Figure 2 represents the periodicity of tasks, adapted from the book Distributed Real-Time Systems (ERCIYES, 2019)).

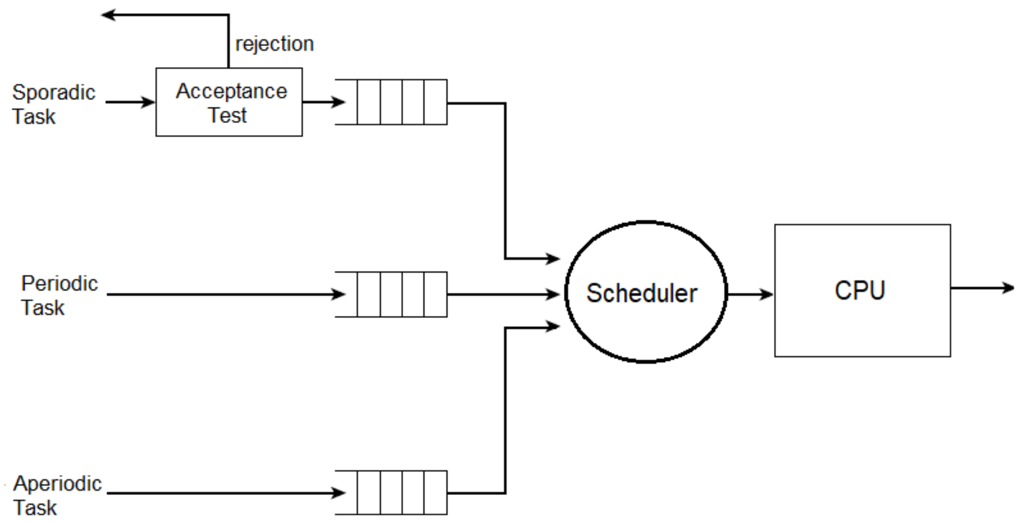


Figure 2 – Task types in operating systems (source: [ERCIYES, 2019]).

Each task also has a priority that determines the precedence of execution in relation to the other tasks that make up the system. The main properties of a task are:

- **Computing time:** time during which the task effectively executes;
- **Period:** the interval in which the task must replay the processor;
- **Deadline:** absolute time at which execution must end;
- **Priority:** importance value of the task in the system.

Figure 3 represents the properties of the task: computing time, period and deadline. The priority is defined according to the scheduler algorithm.

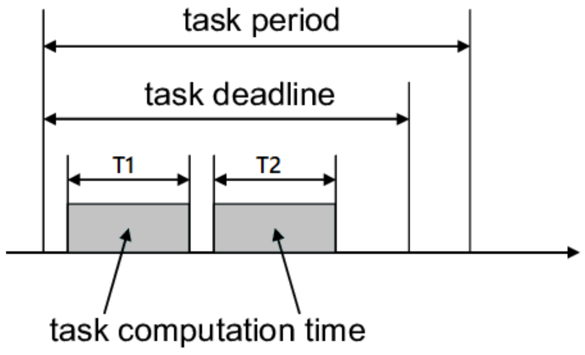


Figure 3 – Properties of the task (source: [IGNAT, 2008]).

For the control of the scheduling process, states have been created that are assigned to these tasks during the execution of the application. From the book Real-Time Concepts (QING; YAO, 2003), Figure 4 demonstrates these states and possible transitions.

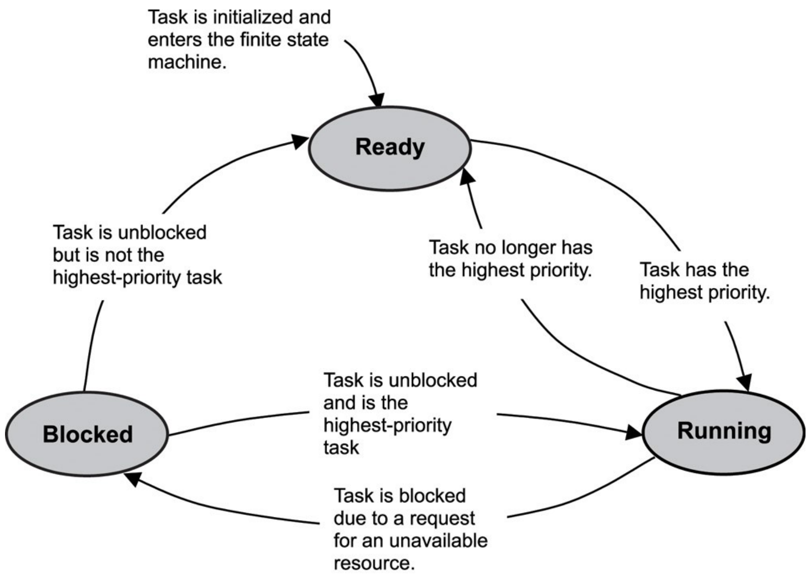

Task is initialized and enters the finite state machine.
Ready
Task is unblocked but is not the highest-priority task
Task no longer has the highest priority.
Task has the highest priority.
Blocked
Task is unblocked and is the highest-priority task
Running
Task is blocked due to a request for an unavailable resource.

Figure 4 – Task States (source: [QING; YAO, 2003]).

As demonstrated in Figure 4, Ready is the initial state of the tasks; in this state, the task waits for the scheduler execution. The blocked state is when a task requests some unavailable resource or is waiting for some semaphore. When the task has the highest priority and is not locked, the task assumes the running state, and the process starts to consume the processing time.

- **Semaphores**
  The semaphore is used to control access to shared resources in a multitasking environment. The semaphore is used for synchronization and mutual exclusion. The semaphore has to block the resource before it is used, and after using, the resource must be released. While the feature is in use, any other process that uses it should wait for the release.

- **Message Queue**
  The message queue is an RTOS object, as illustrated in Figure 1. The message queue is a buffer through which messages are sent from the sender to the receiver for communication and synchronization purposes.

Moreover, the main function of the Services in Figure 1 is to help in the development of the application. These services are usually for the interface between the application and hardware, such as Input/Output devices, in addition to facilitating time management and attendance of interruptions (QING; YAO, 2003).

All of the services and objects presented in Figure 1 are essential for the most important requisite of the real-time systems; the time conditions and the time requisition are guaranteed for the scheduling algorithms.

2.1.2 Scheduling Algorithms

The part of the OS that chooses which process to execute is called the scheduler, and the algorithm used is called the scheduling algorithm. The scheduling process affects system performance because it determines which processes to expect and which ones to progress.

Some of the most common algorithms (Round Robin, Rate Monotonic and Earliest Deadline First) are presented in (FARINES J.; FRAGA, 2000)); this Thesis addresses the EDF algorithm. It is essential to clarify the parameters that model each task (execution time (Ci), period (Ti) and deadline (Di)) and how these parameters influence the scheduling process. Moreover, those parameters are essential to determine if the tasks can be scheduled.

Regardless of the scheduler algorithm, the main goal of the scheduler process is to decide which process will run at each point in time. Usually, it is a preemptive scheduler, as demonstrated in Figure 5 from (QING; YAO, 2003), meaning that it can control the process by stopping, switching, moving back or forward the queue, or even initiating a new process.

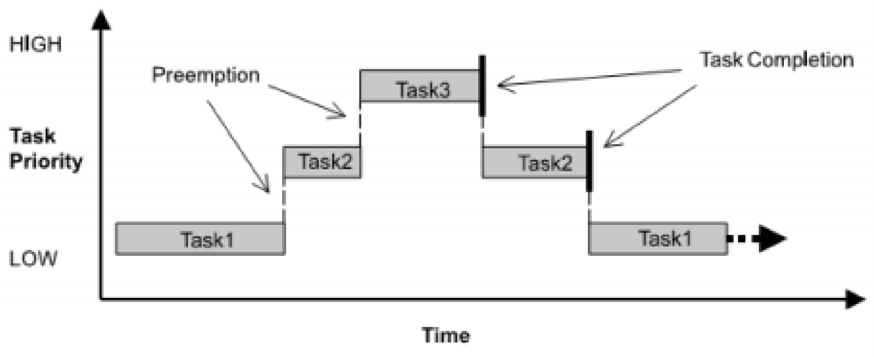


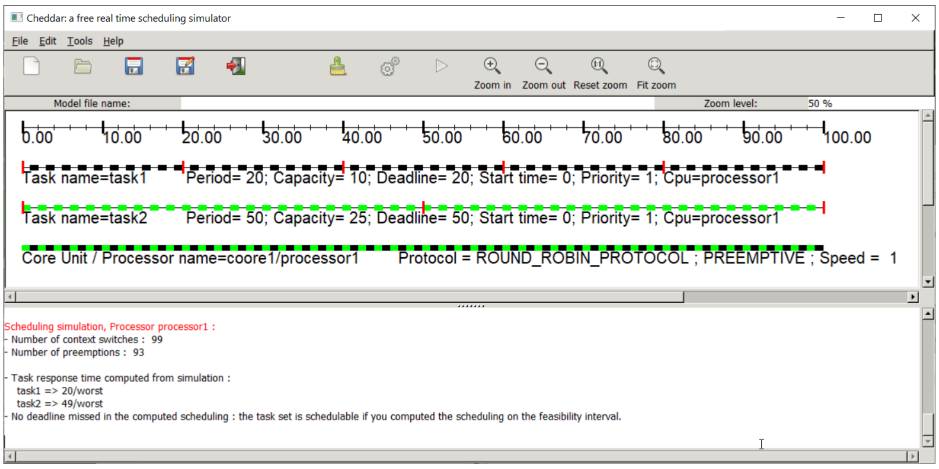
Figure 5 – Preemptive scheduling algorithm (source: [QING; YAO, 2003]).

In order to better demonstrate this scenario, Table 1 has two tasks that need to be scheduled. To represent the scheduler process with algorithms RR, RM, and EDF, a Real-Time scheduling analysis with Cheddar (SINGHOFF F.; LEGRAND, 2004) is demonstrated in Figure 6, Figure 7, and Figure 8.

Task	Ci	Ti	Di
T1	10	20	20
T2	25	50	50

Table 1 – Task parameters.

- Round Robin (RR) is a scheduler algorithm that seeks to be fair with all tasks; RR does not prioritize tasks. A periodic interruption exchanges task execution in the processor guaranteeing slices of time for all tasks. This scaling style is generally not used in real-time systems because of the lack of priority between tasks. If there is a critical task in the system, it will not have any privileges over the others. As demonstrated in Figure 6.



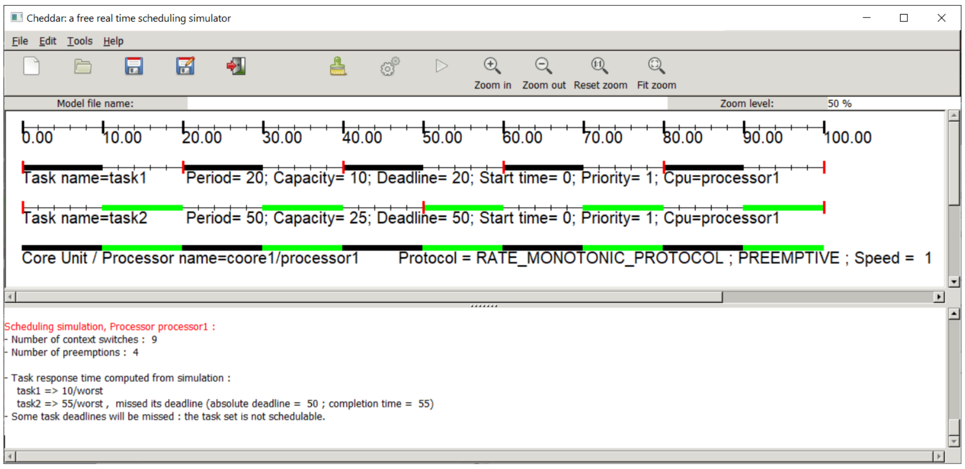
```
Task name=task1      Period= 20; Capacity= 10; Deadline= 20; Start time= 0; Priority= 1; Cpu=processor1
Task name=task2      Period= 50; Capacity= 25; Deadline= 50; Start time= 0; Priority= 1; Cpu=processor1
Core Unit / Processor name=coore1/processor1      Protocol = ROUND_ROBIN_PROTOCOL ; PREEMPTIVE ; Speed =  1
```

```
Scheduling simulation, Processor processor1 :
- Number of context switches :  99
- Number of preemptions :  93

- Task response time computed from simulation :
   task1 => 20/worst
   task2 => 49/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.
```

Figure 6 – Round Robin (source: Author).

- Rate Monotonic (RM) is one of the real-time scheduling algorithms. This algorithm has the following assumptions:
  - Tasks are periodic and independent.
  - Computing time, period and deadline are known and constant.
  - Fixed priorities. In RM, the task priorities are inversely proportional to the period; thus, the task with the shortest period has the highest priority.

  When tasks from Table 1 are scheduled by the Rate Monotonic algorithm, it is possible to observe that Task 'T2' has its deadline missed. In time 50, Task 'T2' is available again, implying that, the period 'Ti' is reached; however, only in time 55, Task 'T2' completes the total execution time, as demonstrated in Figure 7.



Task name=task1 Period= 20; Capacity= 10; Deadline= 20; Start time= 0; Priority= 1; Cpu=processor1

Task name=task2 Period= 50; Capacity= 25; Deadline= 50; Start time= 0; Priority= 1; Cpu=processor1

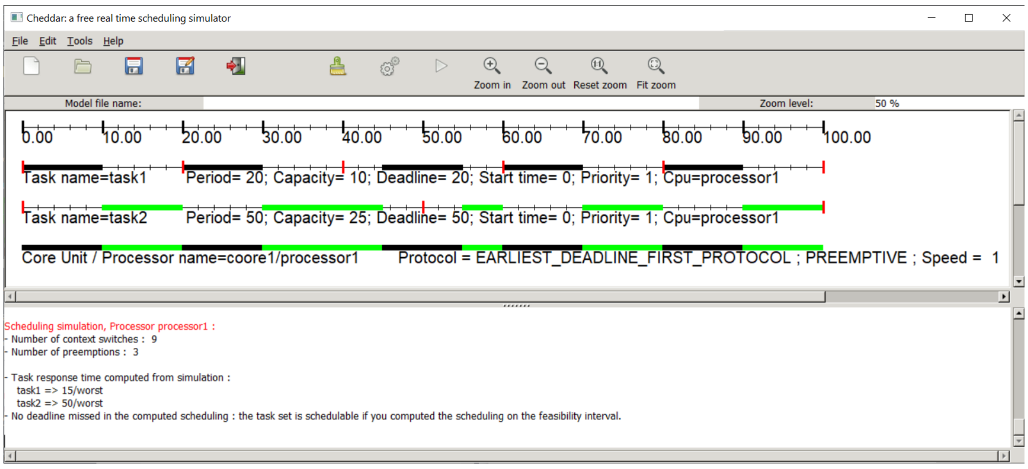
Core Unit / Processor name=coore1/processor1 Protocol = RATE_MONOTONIC_PROTOCOL ; PREEMPTIVE ; Speed = 1

```
Scheduling simulation, Processor processor1 :
- Number of context switches :  9
- Number of preemptions :  4

- Task response time computed from simulation :
   task1 => 10/worst
   task2 => 55/worst ,  missed its deadline (absolute deadline =  50 ; completion time =  55)
- Some task deadlines will be missed : the task set is not schedulable.
```

Figure 7 – Rate Monotonic (source: Author).

- Earliest Deadline First (EDF) is also one of the real-time scheduling algorithms. Has the premises very similar to the RM, being:
  - Tasks are periodic and independent.
  - Computing time, period and deadline are known and constant.
  - Has dynamic priorities and, task priorities are sorted according to the time remaining for the period to occur. The shorter the term, has the higher priority.

  When tasks from Table 1 are scheduled by the Earliest Deadline First algorithm, it is possible to observe that Task 'T2' does not have your deadline missed. Till time 40, RM and EDF have the same schedule; the main difference here is that, with the EDF algorithm, in time 40, task T2 continues executing. This happens because, in time 40, task T2 has a shorter term, so T2 has the highest priority and can continue running. None of the tasks from Table 1 have missed deadlines with EDF, as demonstrated in Figure 8.



```
Task name=task1    Period= 20; Capacity= 10; Deadline= 20; Start time= 0; Priority= 1; Cpu=processor1
Task name=task2    Period= 50; Capacity= 25; Deadline= 50; Start time= 0; Priority= 1; Cpu=processor1
Core Unit / Processor name=coore1/processor1    Protocol = EARLIEST_DEADLINE_FIRST_PROTOCOL ; PREEMPTIVE ; Speed =  1
```

```
Scheduling simulation, Processor processor1 :
- Number of context switches :  9
- Number of preemptions :  3

- Task response time computed from simulation :
   task1 => 15/worst
   task2 => 50/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.
```

Figure 8 – Earliest Deadline First (source: Author).

As mentioned before, the EDF algorithm is similar to RM, but it has a dynamic priority. The most important task is the one that has the deadline closest to the current time (FARINES J.; FRAGA, 2000). The assumptions of this scheduling algorithm are similar to those of RM. To guarantee that a set of tasks is scheduled with RM and have all deadlines met, the following Equation 2.1 is used.

$$U = \sum_{i}^{n} \frac{Ci}{Ti} \tag{2.1}$$

Knowing U (Utilization), inequality is used:

$$U <= n(2^{\frac{1}{n}} - 1)$$

If this condition is respected, it will be possible to schedule using Rate Monotonic without losing any deadline.

Applying Equation 2.1 for tasks from Table 1:

$$U = \sum_{i}^{n} \frac{Ci}{Ti}$$

$$U = \frac{C1}{T1} + \frac{C2}{T2}$$

$$U = \frac{10}{20} + \frac{25}{50}$$

$$U = 1$$

Then checking the schedulability with:

$$U <= n(2^{\frac{1}{2}} - 1)$$

$$1 <= 0.41$$

It is possible to observe that some tasks will have the deadline missed.

For scheduling a set of tasks with EDF and have all deadlines met, the following test is used Equation 2.2.

$$U = \sum_{i}^{n} \frac{Ci}{Ti} \tag{2.2}$$

Then, the utilization is checked using:

$$U <= 1$$

If U is greater than 1, then the EDF algorithm cannot schedule the task set; if U is less or equal than 1, it is possible to schedule using Earliest Deadline First.

Applying Equation 2.2 for tasks from Table 1, it is possible to observe that none tasks will have the deadline missed.

$$U = \sum_{i1}^{n} \frac{Ci}{Ti} <= 1$$

$$U = \frac{C1}{T1} + \frac{C2}{T2}$$

$$U = \frac{10}{20} + \frac{25}{50}$$

Then checking the schedulability with:

$$1 <= 1$$

The algorithms Rate Monotonic and Earliest Deadline First were designed specifically for systems with more rigid time constraints, thus meeting real-time system requirements. For real-time systems, fault tolerance is also an important matter.

2.2 Fault Tolerance

The goal of fault tolerance is to achieve dependability. The term dependability indicates the quality of service provided by a given system and the trust placed in the service provided. Fault tolerance and dependability are not properties of a system to which numerical values can be assigned directly. However, all attributes of dependability correspond to numerical measures. According (PRADHAN, 1996) and (ANSARI et al., 2019), key attributes of dependability are reliability, availability, safety, security, maintainability, testability, and performability.

Reliability and availability are increasingly desirable in computing systems as day by day increases the dependency of society on automated and computerized systems. Whether in the control of terrestrial and aerial traffic or power plants, in the maintenance of sensitive data on the lives and finances of citizens and companies, in telecommunication and international commercial transactions of all kinds, computers are active and continuous.

Failures are inevitable, but the consequences of failures, like system collapse, service interruption and data loss, can be avoided by the proper use of feasible and easy-to-understand techniques. So, fault tolerance is the ability to identifying a failure in one component or system and decide from this what to do to avoid more damages, for example, restart the system (LAPRIE, 1985), (SCHERRER; STEININGER, 2003).

It is easy to imagine that defects in these systems can lead to major catastrophes. The high complexity of the software also acts as a villain in this scenario. However, defects can be avoided using fault-tolerance techniques in two approaches, hardware or software (TARRILO; BOLZANI; VARGAS, 2009).

- Software Approach: Software techniques use redundant instructions, flow control through checkpoints, or even both. The performance reduction is around 40% by fault detection (SHYE et al., 2009). Because these techniques are based on the insertion of instructions, thus they cause delays; this effect can be diminished through profiling. In other words, the algorithm is protected only in the most critical regions; however, the delay cannot be eliminated.
- Hardware Approach: Watchdog timer is a module responsible for monitoring some function in the system and when this function was violated, this module can reset the system, for example. The watchdog timer is the most common I-IP used in embedded systems for fault-tolerant purposes (JACK, 2004).

Moreover, directly connected with fault tolerance, it is necessary to explain three concepts, failure, error, and defect, and explain how these concepts are connected.

2.3 Fault, Error and Defect

According (PRADHAN, 1996), there is a relationship between failure, error, and defect; its definitions are presented below:

- Failure: Internal and external natural phenomena can origin failures, as well, accidental or intentional human actions. A failure may be generated due to interference or aging components. The fault just is considered active when it produces an error. The failures occur in the physical universe;
- Error: The error evidences the defect; for example, when there is a difference between the value obtained and the expected value, it is an error. The error can be propagated; that is, one error is transformed into another error. The errors occurs in the information universe;
- Defect: Occurs when there is a deviation from the project specifications. The defect occurs when an error is propagated. The defect occur in the user universe.

Figure 9 from the book Fault-tolerant computer system design (PRADHAN, 1996) demonstrates the relation among failure, error, and defect.

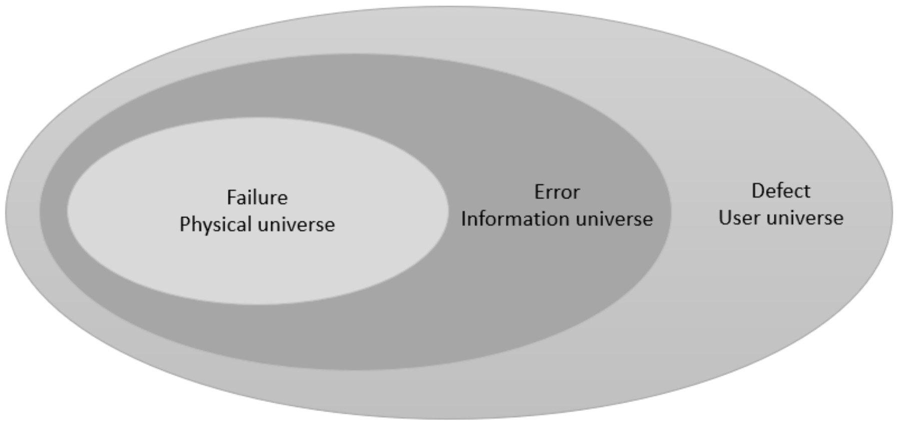


Figure 9 – Relations among failure, error and defect (source: [PRADHAN, 1996]).

2.4 Related Research

Over the years, and due to the increasing complexity of applications and the demand for low-power devices, simply increasing the frequency of processors has become inappropriate. The evolution in manufacturing technology impacts directly in the complexity of the designs for high-frequency processors has greatly increased.

It is necessary to highlight that all the techniques have a common goal: detecting transient or permanent failures during the execution of the application itself. To date, no technique has been found in the literature that has the purpose of detecting failures during the execution of the operating system, more specifically, the execution of the task scheduling process with dynamic priorities.

The existing techniques demonstrated promising results, but just for algorithms like Round Robin, with static priorities. Even the preemptive scheduler did not have dynamic priorities. For critical real-time systems, loss of performance should be avoided at all costs, as a timing failure can compromise the system or cause severe damage.

The first one proposed by (TARRILO; BOLZANI; VARGAS, 2009) was developed for operating systems with the Round-Robin algorithm that does not support interrupts. In order to solve this problem, an I-IP capable of manipulating interrupts and with support for preemptive operating systems was proposed by (SILVA; BOLZANI; VARGAS, 2011).

However, due to the increasing use of multicore technology in modern systems, it was proposed by (OLIVEIRA; BOLZANI; VARGAS, 2015), the development of a hardware-based technique. The central idea of the proposal is focused more specifically on the process of scheduling tasks, being able to detect more failures that the native protection of the operating systems can detect and with less latency.

3 Operating System (HellfireOS)

In this chapter, the Hellfire Real-Time operating system (HellfireOS) will be introduced. Scheduling options, system characteristics, memory and system calls will also be exposed.

HellfireOS is a preemptive real-time system with dynamic task scheduling and system calls to deal with a missed deadline, context switch, processor capacity and memory management. Moreover, HellfireOS is open source, and it is available in (FILHO S. J.; AGUIAR, 2007) and it is illustrated by Figure 10.

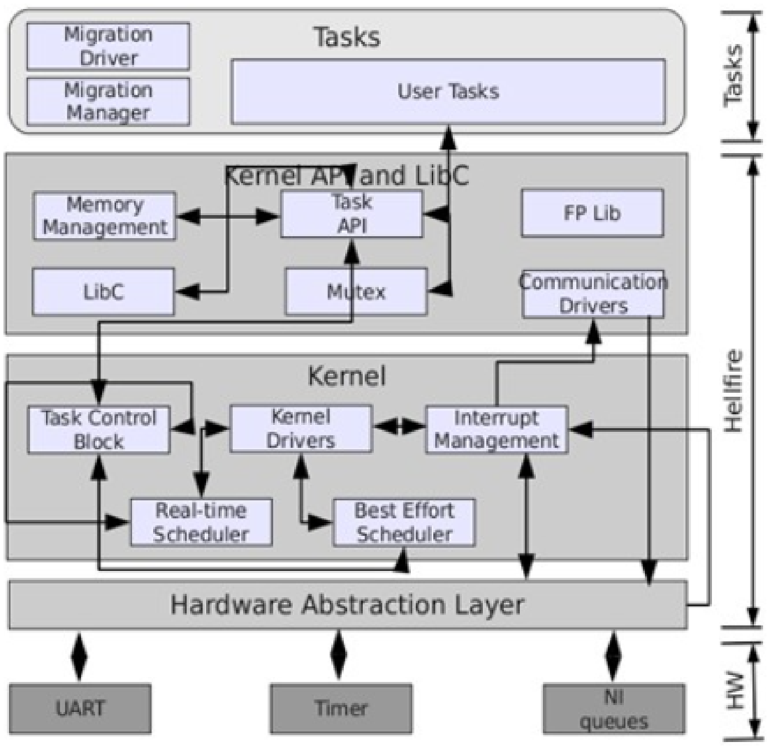


Figure 10 – Hellfire block diagram (source: [JOHANN, 2007]).

Figure 10 demonstrate the three layers in the system:

- Tasks;
- Hellfire;

- Hardware.

The first one, Tasks, has the tasks and parameters created by the user. The second one, Hellfire, has the scheduler algorithms in the 'Real-time Scheduler' block, has the 'Interrupt Management' block with the management of the IRQ's and 'Task Control Block' with the management of the queues, also, memory management and Mutex. And, the third one, the Hardware layer.

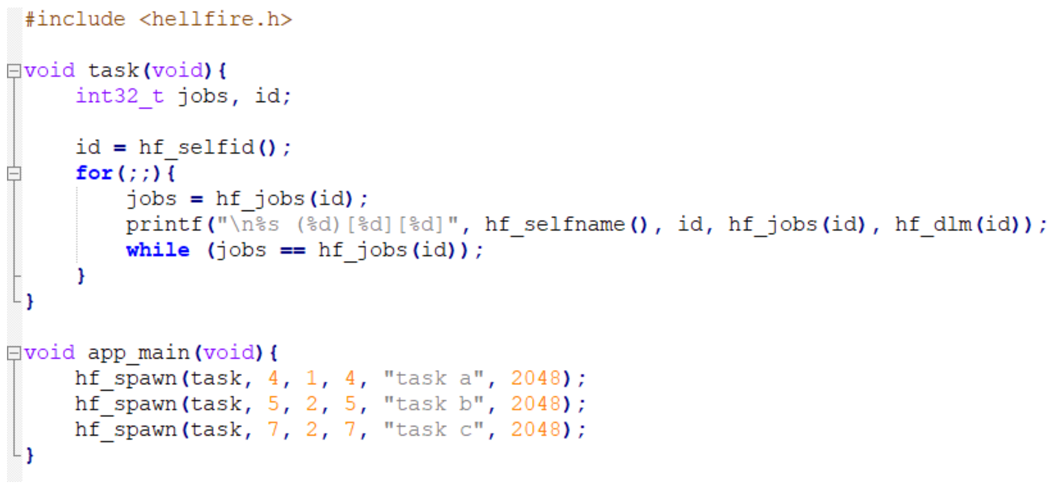
3.1 HellfireOS Scheduling Algorithms

HellfireOS counts with the following scheduling algorithms:

- Round Robin: take a task from the run queue, copy its entry and put it back at the tail of the run queue. If the task is in the blocked state, it is put back at the tail of the run queue and the next task is picked up.
- Rate Monotonic: sort the queue of tasks by period, update real-time information (remaining deadline and capacity) of the whole task set. If the task at the head of the queue fits the requirements to be scheduled, then register the task to be scheduled.
- Earliest Deadline First: sort the queue of tasks by period, update real-time information (remaining deadline and capacity) of the whole task set. If the task at the head of the queue fits the requirements to be scheduled (not blocked, has jobs to execute and no task with higher priority according to EDF was selected), then register the task to be scheduled.

3.2 Task Functions

Functions that monitor the task behavior are defined in a specific block of code of the OS. They are included in the system during the initialization or execution. A simple task creation is illustrated in Figure 11.



```c
#include <hellfire.h>

void task(void){
    int32_t jobs, id;

    id = hf_selfid();
    for(;;){
        jobs = hf_jobs(id);
        printf("\n%s (%d)[%d][%d]", hf_selfname(), id, hf_jobs(id), hf_dlm(id));
        while (jobs == hf_jobs(id));
    }
}

void app_main(void){
    hf_spawn(task, 4, 1, 4, "task a", 2048);
    hf_spawn(task, 5, 2, 5, "task b", 2048);
    hf_spawn(task, 7, 2, 7, "task c", 2048);
}
```

Figure 11 – Hellfire task example (source: Author).

To create a task set to be executed by the HellfireOS, it is needed to create a new file, include the file hellfire.h. It is needed to call function hf_spawn() to pass the task parameters (period, capacity, deadline, name and, stack_size) for each task. And also, function hf_selfid() to get the task id of the current task. Function hf_jobs is also called to get the number of executed jobs of a task. Function hf_selfname() is called to get the current task name and, the last one, hf_dlm is called to get the number of deadline misses of a task.

For the control of the scheduling process, states have been created that are assigned to these tasks during the execution of the application. Figure 12 demonstrates these states and possible transitions for the HellfireOS.

All tasks are started on state 'Not executed yet'. After the first execution, if the task is not blocked or waiting for a semaphore, it will be in Ready state; in this state, the task is waiting for the scheduler to select the task to be executed by the processor. Tasks on Blocked state are waiting for an unavailable resource. Tasks on Waiting status are waiting for a semaphore. And, the task on Running state is being executed by the processor. If at the state Ready are no tasks to be executed, the Idle task will assume the processor. The Idle task is always on Ready or Running state; it cannot be on Blocked or Waiting state.

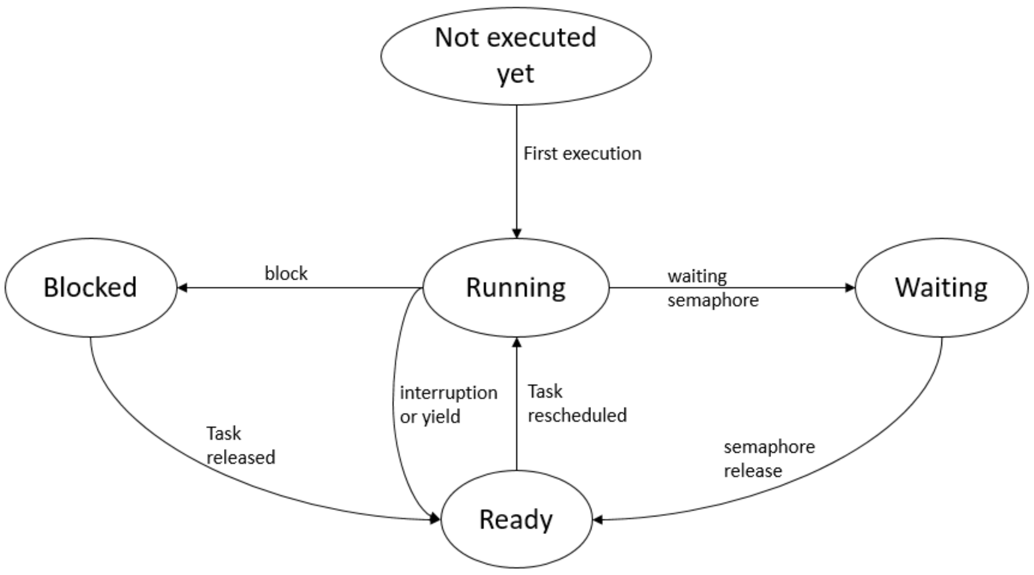


Figure 12 – Hellfire task states (source: [JOHANN, 2007]).

Moreover, communication among tasks occurs by shared memory, using mutexes and semaphores. Still, in the context of task management, HellfireOS has several functions to manage the tasks, as demonstrated in Table 2.

HellfireOS	**Format**	**Description**
hf_id	int32_t hf_id(int8_t *name);	Get a task id by its name
*hf_name	int8_t *hf_name(uint16_t id);	Get a task name by its id.
hf_selfid	uint16_t hf_selfid(void);	Get the current task id.
*hf_selfname	int8_t *hf_selfname(void);	Get the current task name.
hf_state	int32_t hf_state(uint16_t id);	Get the current state of a task.
hf_jobs	int32_t hf_jobs(uint16_t id);	Get the number of executed jobs of a task.
hf_dlm	int32_t hf_dlm(uint16_t id);	Get the number of deadline misses of a task.
hf_spawn	int32_t hf_spawn (void (*task)(), uint16_t period, uint16_t capacity, uint16_t deadline, int8_t *name, uint32_t stack_size);	Spawn a new task. - task is a pointer to a task function / body. - period is the task RT period (in quantum / tick units). - capacity is the amount of work to be executed in a period (in quantum / tick units). - deadline is the task deadline to complete the work in the period (in quantum / tick units). - name is a string used to identify a task. - stack_size is the stack memory to be allocated for the task.
hf_yield	void hf_yield(void);	Yields the current task. The current task gives up execution.
hf_block	int32_t hf_block(uint16_t id);	Blocks a task.
hf_resume	int32_t hf_resume(uint16_t id);	Resumes a blocked task.
hf_kill	int32_t hf_kill(uint16_t id);	Kills a task.
hf_delay	int32_t hf_delay (uint16_t id, uint32_t delay);	Delays a task for an amount of time.

Table 2 – Hellfire: task functions.

Equally important, HellfireOS has several functions to manage the scheduling process, as demonstrated in Table 3.

HellfireOS	Description
dispatch_isr	The job of the dispatcher is responsible for: - save the current task context; - update its state to ready; - invoke the real-time scheduler; - update the scheduled task state to running and restore the context of the task.
sched_edf	The scheduling algorithm Earliest Deadline First: - Sort the queue of RT tasks by the remaining deadline; - Update real-time information (remaining deadline and capacity) of the whole task set. - If the task at the head of the queue fits the requirements to be scheduled (not blocked, has jobs to execute and no task with higher priority according to EDF was selected), then register the task to be scheduled.
hf_semwait	Wait on a semaphore. The semaphore count is decremented and the calling task is blocked and queued on the semaphore if the count reaches a negative value. If not, the task continues its execution.
hf_sempost	Signal a semaphore. The semaphore count is incremented, and the task from the head of the semaphore queue is unblocked if the count is less than or equal to zero.

Table 3 – Hellfire: scheduling functions.

3.3 Fault detection of HellfireOS

HellfireOS counts with the following native function to detect faults during the execution of the task scheduling algorithm:

- Missed deadline
  HellfireOS counts with a deadline missed verification; if a task has your deadline missed, function hf_dlm, get the number of deadline misses of a task during all the execution time. This is illustrated by Figure 13; task b has your deadline

missed when trying to execute the time unit 21 (RM Scheduling algorithm, the same example from Figure 7.

HellfireOS also counts with a scheduler function named rt_schedule() that verifies if the task that will be executed by the processor has met their deadline; if so, it verifies if the task has completed the capacity. If the task was has the capacity fully executed, no deadline is missed. Otherwise, the deadline is missed. This process is executed for all tasks and function hf_dlm stores how many times the deadline was missed during the HelfireOS execution.

The HellfireOS scheduling results are logged in an output file that can be observed in Figure 13. This output file contains information regarding the architecture, clock, heap size, and max number of tasks. Later, once the tasks are loaded, we can check the tasks that are planned to be executed. In this example, there are two tasks (a and b) created by the user and an idle task created by the system to assume the processor once the processor is idle. It is also logged the task parameters:

- Task name;
- Task id;
- Task period;
- Task capacity;
- Task deadline.

For example, these parameters can be observed on "KERNEL: [task a], id:1, p:20, c:10, d:20".

Once the HellfireOS is up and the task execution starts, at the output, it is possible to check the execution time of each task.

- Task name;
- Task id;
- Task capacity;
- Number of deadline misses.

For example, the results of the execution can be observed on "task a (1)[1][0]".

Checking the results from Figure 13, it is possible to check that task a was the first to be executed and has the capacity completely executed.

"task a (1)[1][0]" to "task a (1)[10][0]"

After that, task b assumes the processor and it is executed till task a be available again.

"task b (2)[1][0]" to "task b (2)[10][0]"

After that, task a assumes the processor and is fully executed again.

"task a (1)[11][0]" to "task a (1)[20][0]"

After that, task b assumes the processor and it is executed till task a be available again.

"task b (2)[11][0]" to "task b (2)[20][0]"

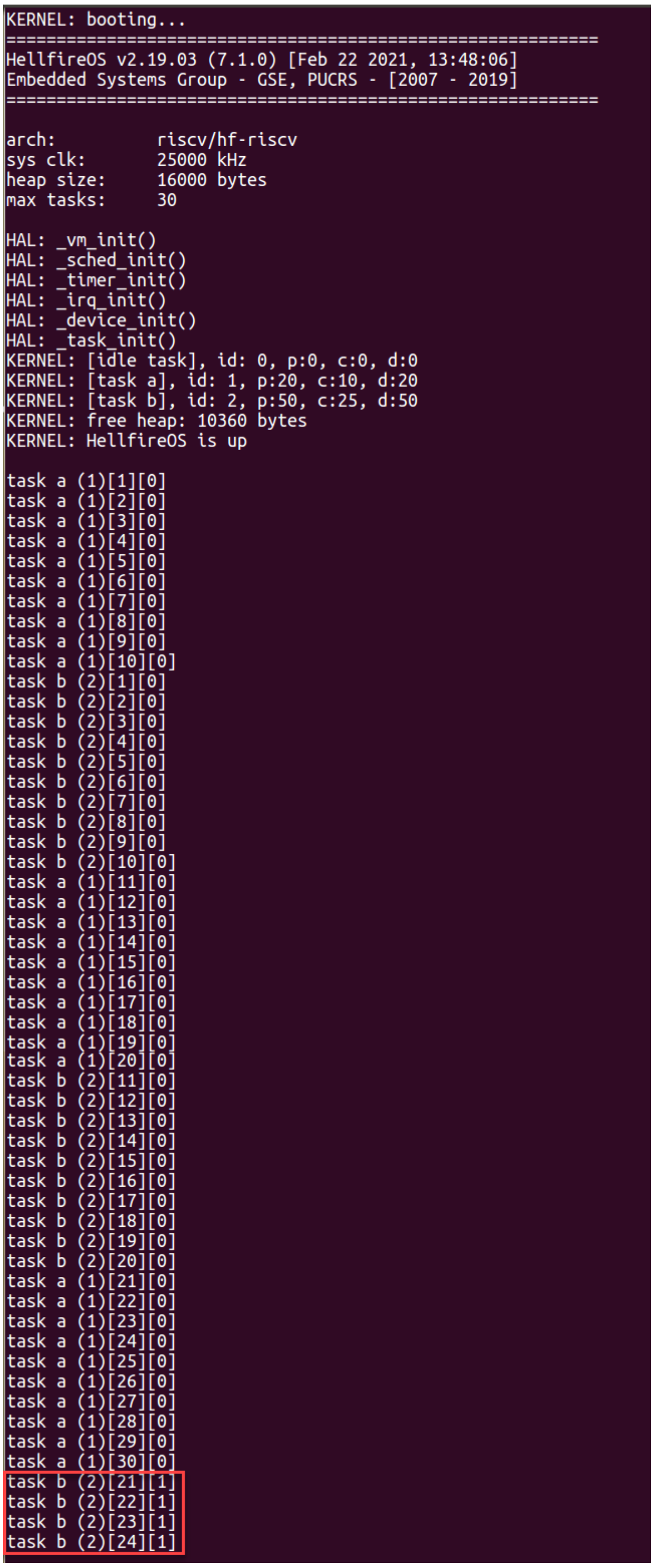
After that, task a assumes the processor and is fully executed again.

"task a (1)[21][0]" to "task a (1)[30][0]"

Then, task b assumes the processor; however, the deadline is missed because a new period starts and, the capacity of task b was not completely executed. In this case, the number of deadline misses is increased to 1.

"task b (2)[21][1]"

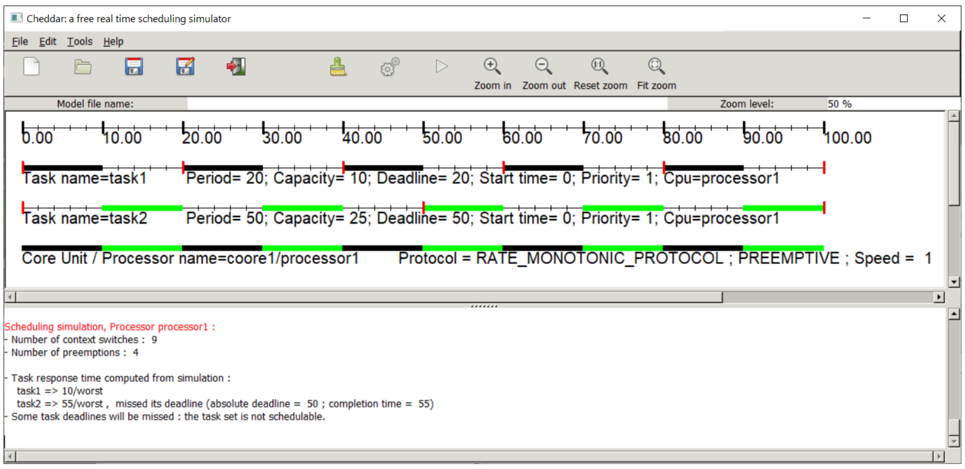
It is possible to observe the same, that task b has the deadline missed when tries to execute the computational unit of time 21 in the red block (Figure 13).



```
KERNEL: booting...
============================================================
HellfireOS v2.19.03 (7.1.0) [Feb 22 2021, 13:48:06]
Embedded Systems Group - GSE, PUCRS - [2007 - 2019]
============================================================

arch:          riscv/hf-riscv
sys clk:       25000 kHz
heap size:     16000 bytes
max tasks:     30

HAL: _vm_init()
HAL: _sched_init()
HAL: _timer_init()
HAL: _irq_init()
HAL: _device_init()
HAL: _task_init()
KERNEL: [idle task], id: 0, p:0, c:0, d:0
KERNEL: [task a], id: 1, p:20, c:10, d:20
KERNEL: [task b], id: 2, p:50, c:25, d:50
KERNEL: free heap: 10360 bytes
KERNEL: HellfireOS is up

task a (1)[1][0]
task a (1)[2][0]
task a (1)[3][0]
task a (1)[4][0]
task a (1)[5][0]
task a (1)[6][0]
task a (1)[7][0]
task a (1)[8][0]
task a (1)[9][0]
task a (1)[10][0]
task b (2)[1][0]
task b (2)[2][0]
task b (2)[3][0]
task b (2)[4][0]
task b (2)[5][0]
task b (2)[6][0]
task b (2)[7][0]
task b (2)[8][0]
task b (2)[9][0]
task b (2)[10][0]
task a (1)[11][0]
task a (1)[12][0]
task a (1)[13][0]
task a (1)[14][0]
task a (1)[15][0]
task a (1)[16][0]
task a (1)[17][0]
task a (1)[18][0]
task a (1)[19][0]
task a (1)[20][0]
task b (2)[11][0]
task b (2)[12][0]
task b (2)[13][0]
task b (2)[14][0]
task b (2)[15][0]
task b (2)[16][0]
task b (2)[17][0]
task b (2)[18][0]
task b (2)[19][0]
task b (2)[20][0]
task a (1)[21][0]
task a (1)[22][0]
task a (1)[23][0]
task a (1)[24][0]
task a (1)[25][0]
task a (1)[26][0]
task a (1)[27][0]
task a (1)[28][0]
task a (1)[29][0]
task a (1)[30][0]
task b (2)[21][1]
task b (2)[22][1]
task b (2)[23][1]
task b (2)[24][1]
```

Figure 13 – HellfireOS: Deadline missed example (source: Author).

The results of Figure 13 can be compared with the results of Figure 14.



Task name=task1 Period= 20; Capacity= 10; Deadline= 20; Start time= 0; Priority= 1; Cpu=processor1

Task name=task2 Period= 50; Capacity= 25; Deadline= 50; Start time= 0; Priority= 1; Cpu=processor1

Core Unit / Processor name=coore1/processor1 Protocol = RATE_MONOTONIC_PROTOCOL ; PREEMPTIVE ; Speed = 1

```
Scheduling simulation, Processor processor1 :
- Number of context switches :  9
- Number of preemptions :  4

- Task response time computed from simulation :
  task1 => 10/worst
  task2 => 55/worst ,  missed its deadline (absolute deadline =  50 ; completion time =  55)
- Some task deadlines will be missed : the task set is not schedulable.
```

Figure 14 – Rate Monotonic (source: Author).

4 HF-RISC Softcore Processor (Hellfire)

In this chapter, the HF-RISC softcore processor will be introduced. HF-RISC was selected because it is an open-source system, and in this way, it is possible to make adjustments to processor files to include the proposed I-IP.

4.1 HF-RISC Architecture

HF-RISC is a small 32-bit, in-order, 3-stage pipelined softcore processor designed at the Pontifical Catholic University of Rio Grande do Sul - PUCRS. All registers / memory accesses are synchronized to the rising edge of the clock. The core can be easily integrated into several applications, and interfaces directly to standard synchronous memories. It was implemented with the C programming language, having some architectural features:

- Memory is accessed in little endian-mode.
- No unaligned loads/stores.
- No co-processor is implemented and all peripherals are memory-mapped.
- Loads and stores take 3 cycles. The processor datapath is organized as a Von Neumann machine; therefore, only one memory interface is shared between code and data accesses.
- Interrupts are handled using memory-mapped VECTOR, CAUSE, MASK, STATUS, and EPC registers:
  - The VECTOR register is used to hold the address of the default (non-vectored) interrupt handler.
  - The CAUSE register is read-only, and peripheral interrupt lines are connected to this register.
  - The MASK register is read/write and holds the interrupt mask for the CAUSE register.
  - The interrupt STATUS register is automatically cleared on interrupts and is set by software when returning from interrupts.

– The EPC register holds the program counter when the processor is interrupted. As an interrupt is accepted, the processor jumps to the VECTOR address, where the first level of irq handling is done. A second-level handler (in C) implements the interrupt priority mechanism and calls the appropriate ISR for each interrupt.

4.2 HF-RISC Organization

As illustrated by (JOHANN S. F.; MOREIRA, 2016), Figure 15 depicts the stages of the HF-RISC pipeline and the tasks executed at each of these stages.

- In the fetch stage, memory is accessed and an instruction becomes available in one cycle. In this same cycle, the PC is updated.
- In the decode stage, an instruction is fed into the decoding and control logic, so values are registered for the next stage. Pipeline bubble insertion is performed in this stage for memory and branch operations.
- In the execute stage, the register file is accessed, and the ALU calculates the result of the operation. Address and data are put on the data bus (on store operations), or data are copied to the register file (on load operations). On logic/arithmetic operations, the ALU result is written to the register file. Branch outcomes are computed in this stage. Multiply operations write the result to HI and LO registers.

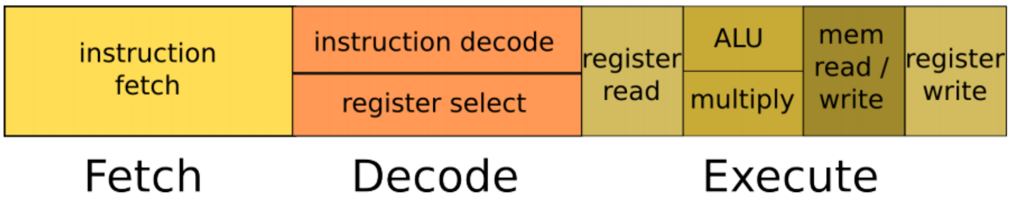


Figure 15 – HF-RISC 3-Stage pipeline and the stage tasks (source: [JOHANN, 2016]).

4.3 HF-RISC + HellfireOS

HellfireOS needs to be connected to HF-RISC to generate a complete system. In order to archive this scenario, it is needed to compile HellfireOS to generate file code.txt and also compile HF-RISC to generate file boot.txt. After that, it will be possible to use the Modelsim simulator to execute the whole system. The simulation output is driven to the Output.txt file. See Figure 16 for details.

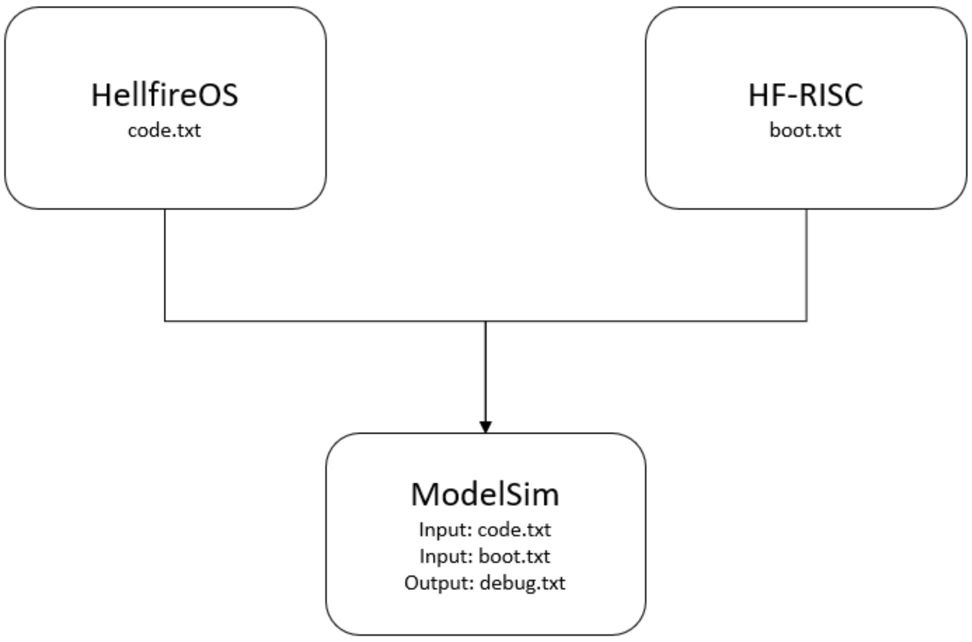


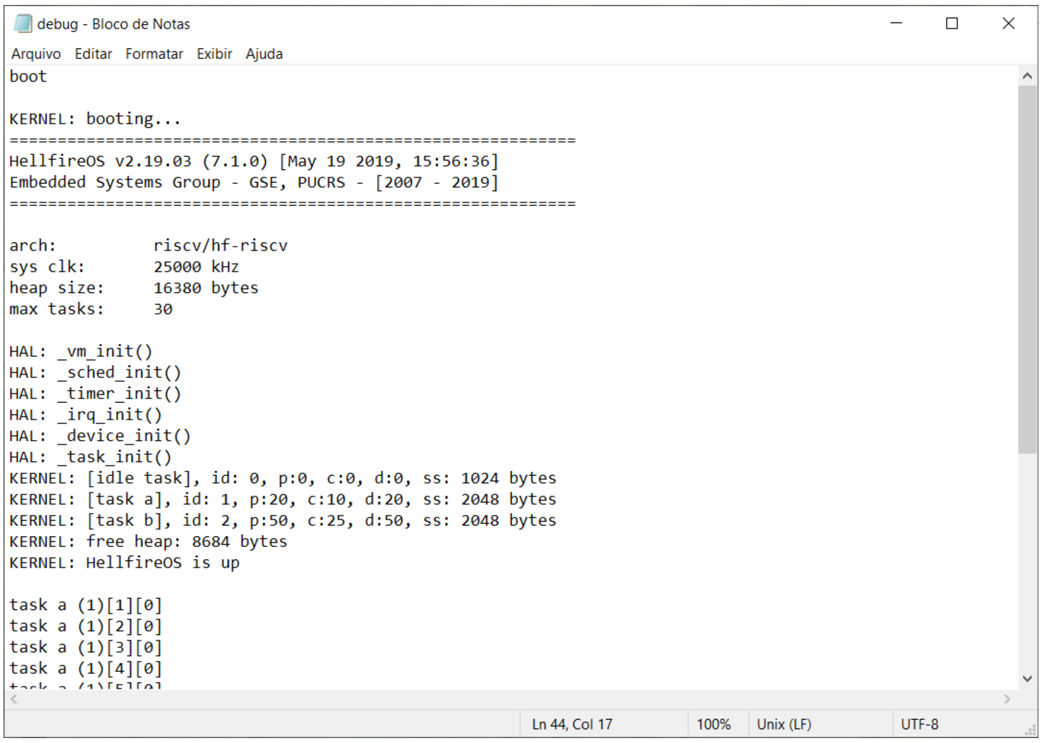
Figure 16 – HellfireOS + HF-RISC (source: Author).

The program (Figure 11) responsible for loading the tasks is located in folder '../app', and it is needed to map this path in the makefile program located in 'APP=hellfireos/platform/single-core'. This folder also needs to set the architecture that will be used in the simulation 'ARCH=riscv/hf-riscv'. In the same file, it is possible to set parameters like heap size, in this case 'HEAP_SIZE=1600' and also, the kernel logs. After setting these parameters, it is needed to compile the HellfireOS + HF-RISCV, and to do that, the command make is used.

After integrating hardware and software, in order to simulate the whole system, it will be needed to create a new project in Modelsim tool. Once the new project was created the following files (FILHO S. J.; AGUIAR, 2007) are needed:

1. from folder ../riscv/core_rv32i:
    a) alu.vhd - The Arithmetic Logic Unit (ALU) stores the logic and arithmetic operations such as addition, subtraction, multiplication, division, etc.
    b) bshifter.vhd - The Barrel Shifter is responsible for shifting a data word by the needed number of bits.
    c) control.vhd - This file describes how the HF-RISCV signals are managed and sent to the datapath.
    d) cpu.vhd - The Central Processing Unit file is responsible for handling the input and output signals for the datapath and ALU.
    e) datapath.vhd - It is responsible for describing the how the data will flow inside the CPU.
    f) int_control.vhd - As part of the ALU, it handles all the operations.
    g) reg_bank.vhd - The Register Bank file describes how and wherein the CPU the data will be temporarily stored.
2. from folder ../riscv/sim:
    a) ram.vhd - The Random Access Memory file is responsible for describing the how and where the data loaded will be temporarily stored.
    b) boot_ram.vhd - This file loads the data in the boot.txt file to the memory
    c) hf_riscv_tb.vhd - The Testbench file is responsible for setting the workspace to simulate the hardware.
3. from folder ../devices/peripherals:
    a) minimal_soc.vhd - The System on a chip file is a support for the whole system operation. It is responsible for: counters, real-time timers, etc.
4. will be needed to add the files generated by the HellfireOS:
    a) boot.txt - This file loads the HF-RISCV, along with other system basic settings.
    b) code.txt - The code.txt file is the HellFireOS operating system code, already compiled.

After including all the necessary files, it will be possible to compile all data and start the simulation. Once the simulation finishes, it is possible to check the output file (output.txt), as illustrated by Figure 17.



```
boot

KERNEL: booting...
============================================================
HellfireOS v2.19.03 (7.1.0) [May 19 2019, 15:56:36]
Embedded Systems Group - GSE, PUCRS - [2007 - 2019]
============================================================

arch:           riscv/hf-riscv
sys clk:        25000 kHz
heap size:      16380 bytes
max tasks:      30

HAL: _vm_init()
HAL: _sched_init()
HAL: _timer_init()
HAL: _irq_init()
HAL: _device_init()
HAL: _task_init()
KERNEL: [idle task], id: 0, p:0, c:0, d:0, ss: 1024 bytes
KERNEL: [task a], id: 1, p:20, c:10, d:20, ss: 2048 bytes
KERNEL: [task b], id: 2, p:50, c:25, d:50, ss: 2048 bytes
KERNEL: free heap: 8684 bytes
KERNEL: HellfireOS is up

task a (1)[1][0]
task a (1)[2][0]
task a (1)[3][0]
task a (1)[4][0]
```

Figure 17 – Output file from HellfireOS + HF-RISC (source: Author).

5 Methodology

Real-time systems are classified according to temporal requirements. They can be categorized as soft real-time systems, firm real-time systems and hard real-time systems. If a hard real-time system fails to meet a time requirement, it can result in catastrophic consequences, economically and in human lives. In a firm real-time, a missed deadline can be tolerated, and for a soft real-time system degraded performance is accepted. Figure 18, adapted from the paper (RAHEJA R.; CHENG, 2009), demonstrates the time curves for firm, hard, soft real-time adaptations.

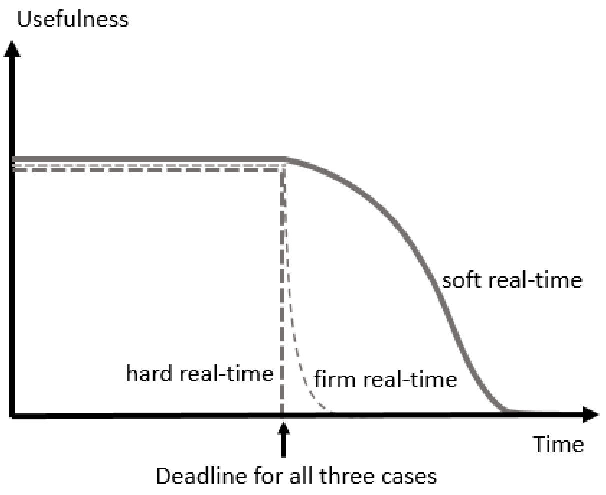


Figure 18 – Relations among firm, hard, soft real-time systems (source: [RAHEJA R.; CHENG, 2009]).

The hard real-time system is our focus in this document. This section addresses the specification and implementation of the I-IP.

5.1 Specification

The I-IP module must perform the passive monitoring of the process of scheduling tasks of the operating system. Passive monitoring cannot interfere with the normal operation of the processor and, cannot reduce the performance of the system.

The solution to be proposed should generate hardware with the smallest possible area compared to the processor area to reduce the probability of transient faults reaching the I-IP itself. Besides that, the proposed technique is based on an I-IP implemented in hardware connected to the address bus between the processor and the memory. One of the main advantages of this approach, compared to the approaches based on software solutions, where specific functions are embedded in the OS kernel, is that the proposed approach does not imply system performance degradation due to the addition of the monitoring functions. Moreover, as it will be presented in the validation chapter, the fault detection latency is just a few processor clock cycles, in contrast to thousands of clock cycles for software-based approaches. The technique is based on a known set of tasks with dynamic priorities.

To develop the I-IP, it is necessary to know the scheduling routines of the operating system; the addresses in which the compiler allocates the scheduling functions; the parameters of the tasks; and if it is the case, the core constraints for system execution (i.e., the set of tasks allocated to each core, if a multicore processor is considered, the hardware resources allowed to be accessed by each task, etc.)

Figure 19 depicts the basic connections between the I-IP, processor, memory and address bus.

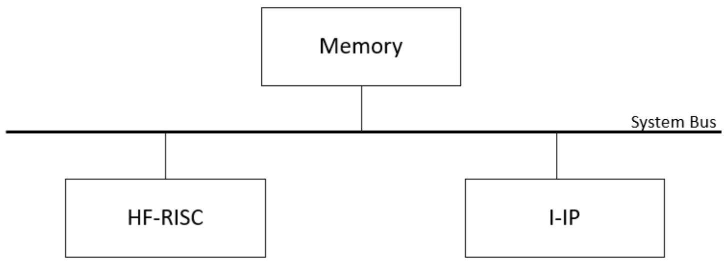


Figure 19 – Overview of the proposed approach (source: Author).

The I-IP is connected to the CPU address bus in order to monitor the scheduling process and inform if an error occurs. I-IP is connected to the CPU address, as shown in Figure 19.

Furthermore, it is possible to have n cores; in this case, the same n number of I-IP is required to perform the core monitoring. In other words, one I-IP is required for each core. Each I-IP will indicate the occurrence of the scheduling in each core. It means identifying the running task is performed by monitoring the addresses accessed by the processor. Each task is associated with an address previously defined and known in memory, and for each task, there is a previously allocated address space in memory.

I-IP will receive the task information and will check if the scheduling process was not violated. The address accessed in memory by the CPU is compared to the address range for each task. If the address accessed is not the same as the address of task, the control unit will indicate an error. The errors to be detected by the I-IP are summarized below:

- Running task is not in the I-IP list;
- Deadline missed;
- Running task is not the one with the highest priority of the ready to-do list;
- A rescheduling event occurred, but the tasks were not rescheduled (and there is at least one task ready to be executed with higher priority than the running one at the moment of rescheduling).

As illustrated by Figure 20, the I-IP is connected to the CPU bus of the embedded system and receive: an interruption signal and the addresses. And, to notify a scheduling error, there is an output signal in the I-IP.

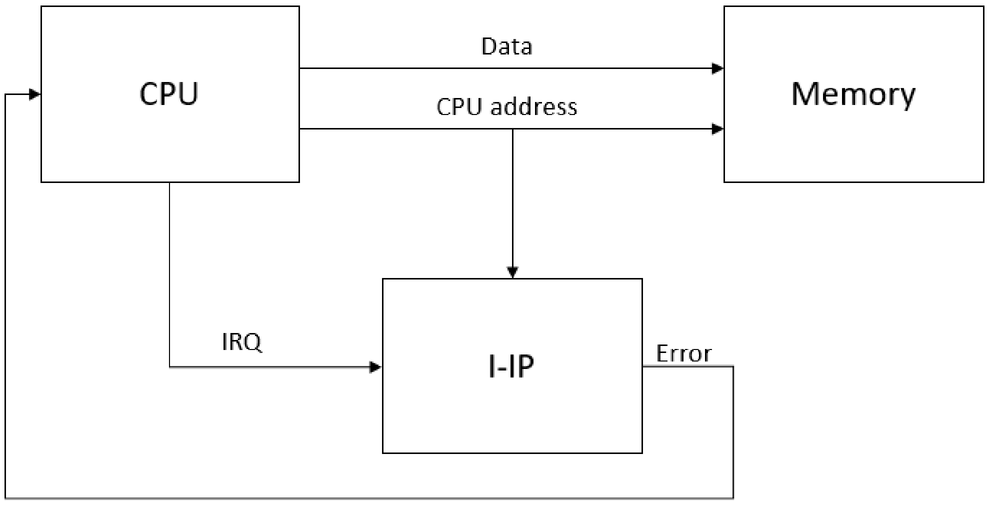


Figure 20 – External architecture of HF-RISC plus I-IP (source: Author).

The I-IP generates an error signal once a scheduling issue is identified. This error signal is called 'MISS'.

5.2 Implementation

In order to implement an I-IP able to detect scheduling issues, HellFireOS was selected because it is an open-source code. Other reasons that contributed to this choice are the fact that the RTOS of the HellFireOS is written in C language, supports semaphores, mutex, message queue, and has a real-time scheduler based on priorities. In HellFireOS, each task can be in one of the five main states: "not executed yet","running", "blocked", "waiting" and "ready", where tasks are organized according to their priorities.

Another important point is knowing the scheduling functions that are implemented in the operating system. Equally important is to describe the FSM (Finite State Machine) that the I-IP should follow (Figure 21). Once the I-IP was developed, it can be integrated into the address bus of the HF-RISC system. Furthermore, it can be validated by simulation.

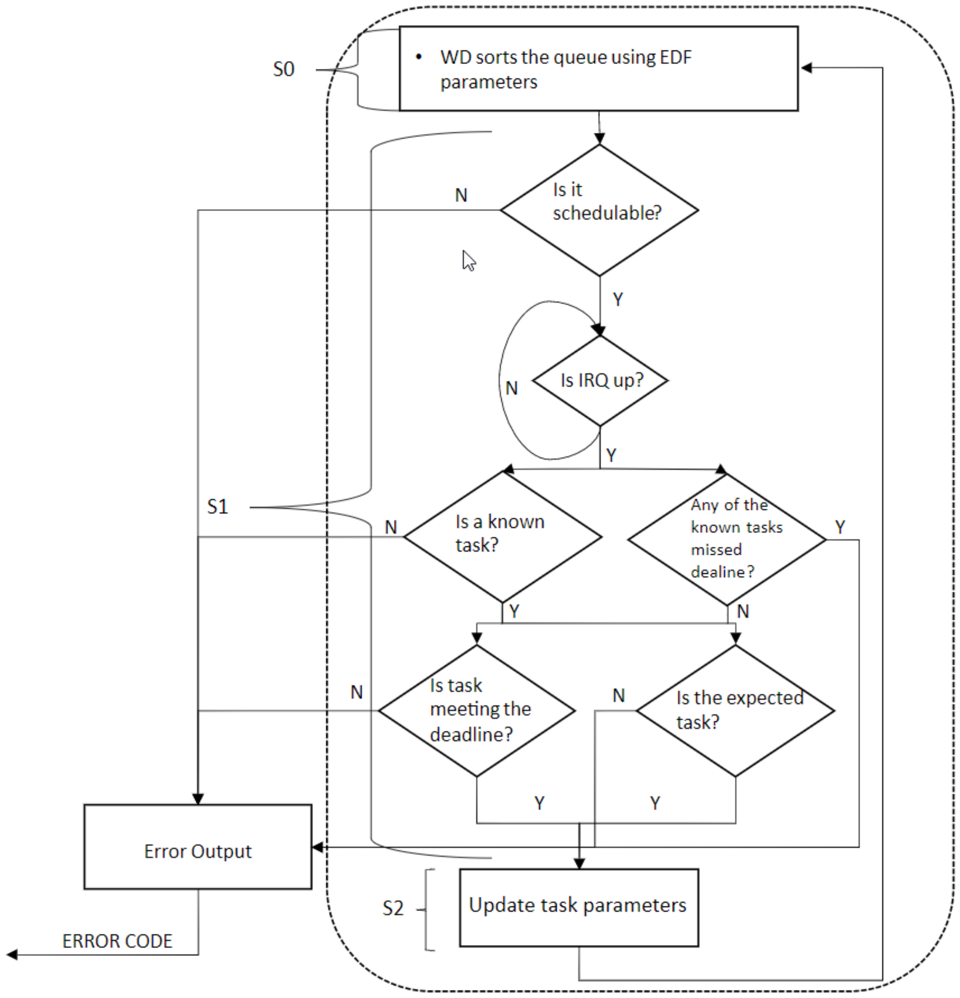


Figure 21 – I-IP Finite State Machine (source: Author).

As demonstrated by Figure 21, the FSM contains three states:

- S0: is the initial state where the queue is sorted by the system.
- S1: illustrate the conditions and validations to be performed by the I-IP.
- S2: update the task parameters. In each execution cycle, the task priority can be changed following the EDF conditions.

The proposed I-IP will be implemented in VHDL, and the module will be located in the processor bus. The I-IP needs to monitor the task scheduler process in order to inform any violations in the scheduling process.

5.2.1 I-IP

The flow starts when HellfireOS + HF-RISCV + I-IP data structures are initialized. After this initialization, interrupt handlers are registered and enabled. At this point, initial tasks are added to the system and execution begins. The system is on hold until an interruption event occurs. At this point, the interrupt service routine is called, the basic processor context is saved, and an interrupt handler is invoked according to the source of the interrupt. In a timer event, the interrupt handler for scheduling (IRQ) is called, the task context is saved, and the scheduler is invoked. After scheduling, the context of the chosen task is restored, and its execution is continued. During the scheduling process, the bus addresses are checked, and if necessary, an error is raised. So, the I-IP module keeps monitoring the scheduling process to identify violations during the RTOS operation.

As illustrated in Figure 20, the I-IP will monitor the CPU address. Once a relevant IRQ occurs, the I-IP module will compare the task address from Hellfire with the task address from the I-IP simulation. If the address is not the expected one, I-IP should raise an error flag. Otherwise, the I-IP must continue running and monitoring.

5.2.2 I-IP Coverage

During the execution time, the I-IP should monitor the scheduling process. Here, four cases of error must be identified, as shown in Figure 22. The first error to be considered is if a task set can be scheduled without losing its deadline. This error is checked by Equation 2.2. Then, the other three errors can be monitored, mainly the deadline missed, then the unknown task being executed, and scheduling issues, such as priority inversion.

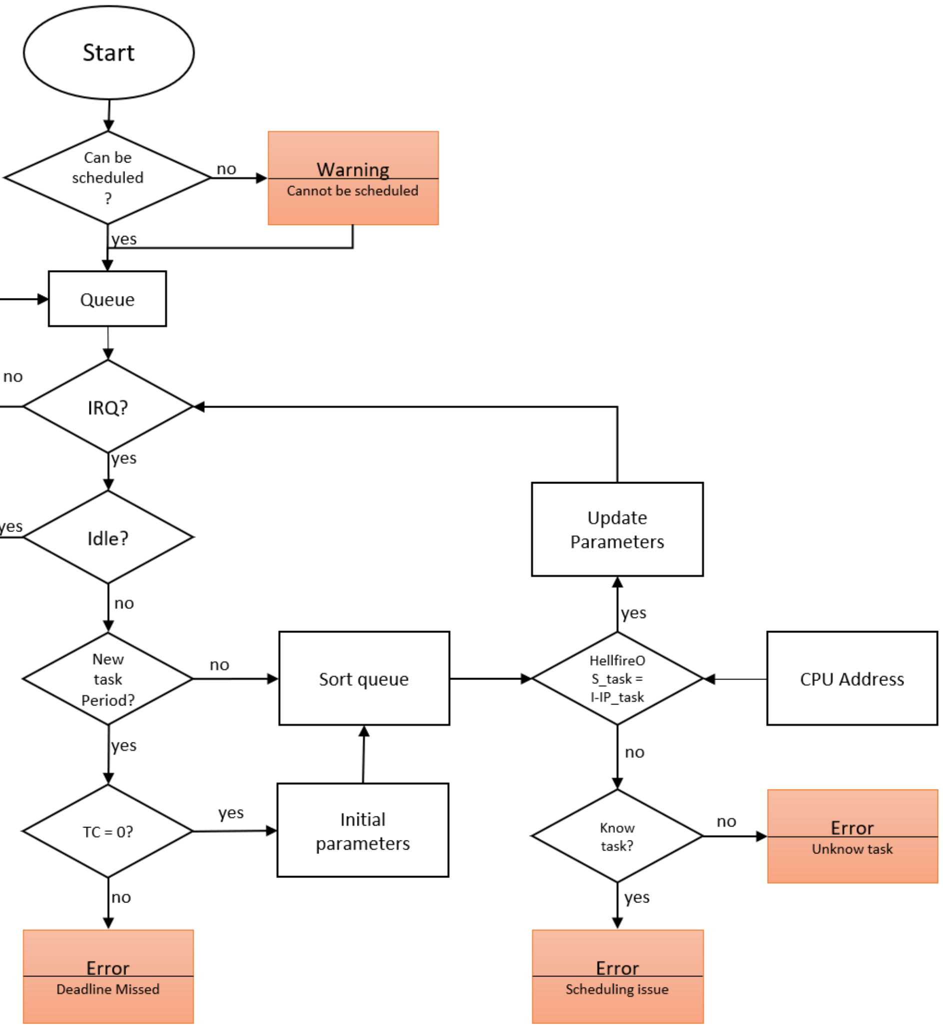


Figure 22 – I-IP: Error detection (source: Author).

As demonstrated in Figure 22, once the execution starts, the I-IP is available and ready to monitor the scheduling process. First, I-IP checks if the task set can be scheduled without missing the deadline, this validation is made by Equation 2.2. If it cannot be scheduled without deadline violation, the system will raise an exception to inform the user that a task will have the deadline violated. This is represented by 'Error: Cannot be scheduled'. This message is just a "warning", so even the task set cannot be scheduled without losing some deadline, the execution will continue.

Subsequently, it will be checked if the IRQ occurs; if not, I-IP will stay in the loop waiting for the IRQ. Once the IRQ occurs, it will be checked if it is the Idle task that is in execution. If so, no action from the I-IP is needed. If it is not the Idle task, the Task period will be checked, if the period is reached and the capacity is not completely executed, the deadline is missed and the message 'Error: Deadline Missed' will be raised. However, if the capacity is fully executed (TC=0), the task receives the initial parameters and the I-IP queue is sorted. Otherwise, if the Task period is not reached, the I-IP queue is sorted. After sorting the queue, the sorted address from I-IP is compared with the address bus from the HellfireOS. If the address is not the same, it is verified if the task that HellfireOS is executing is known; if so, the message 'Error: Scheduling Issue' is raised, informing that some scheduling issue occurred. If the address from HellfireOS is not recognized by the I-IP, the message 'Error: Unknown task' is raised. But if the I-IP address and HellfireOS address are the same, the task parameters are updated, and the I-IP waits for the next interruption.

The I-IP generates an error signal once a scheduling issue is identified. This error signal is called 'MISS' and will identify the following scenarios:

- Error: Cannot be scheduled (001)
- Error: Deadline Missed (010)
- Error: Scheduling Issue (011)
- Error: Unknown task (100)

6 Validation

In this chapter, the experiments used to analyze and validate the proposed technique are presented. Some faults will be simulated in the execution of the task scheduling process to notice if these faults will be detected by the I-IP as expected.

6.1 I-IP Coverage

The implemented I-IP performs the procedure of monitoring the task scheduling activity of the RTOS. The verification performed by I-IP is described in detail below. The whole verification is done using Figure 23, because all task addresses are checked with the I-IP list of addresses to validate any violation and, if it is the case, generate an error. As demonstrated by Figure 23 this approach can be used for multiple cores. In this case, each core will be monitored individually by the I-IP and will receive an error message. For this work, one core validation will be considered.

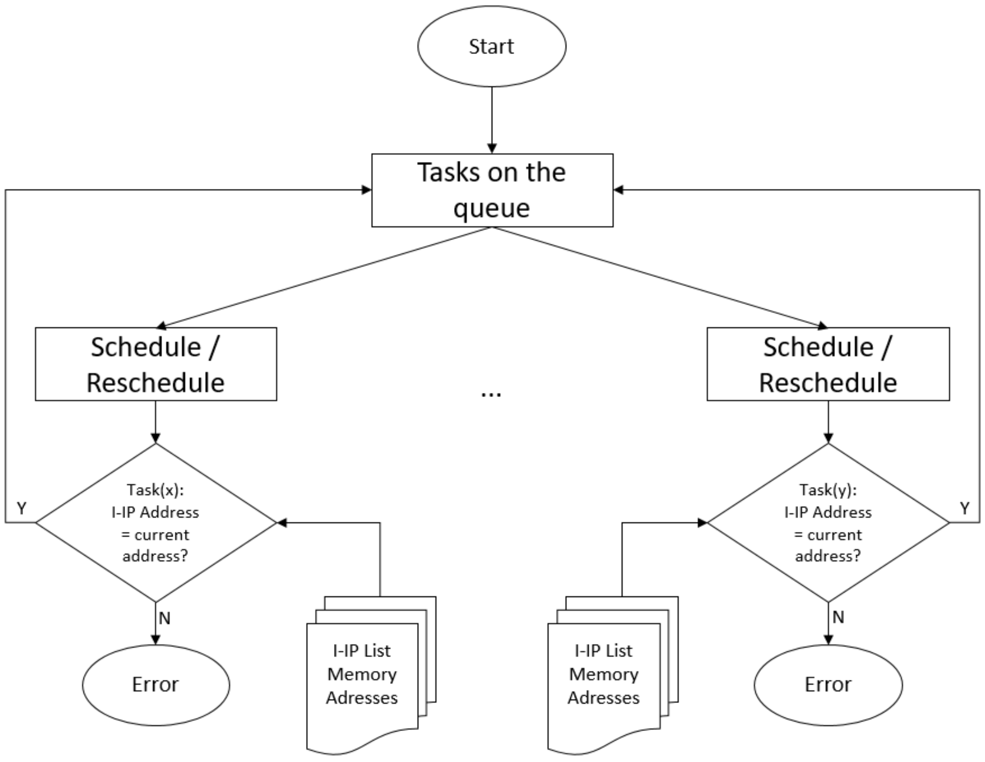

Start
Tasks on the queue
Schedule / Reschedule
...
Schedule / Reschedule
Task(x): I-IP Address = current address?
Task(y): I-IP Address = current address?
Y
N
Y
N
Error
I-IP List Memory Adresses
I-IP List Memory Adresses
Error

Figure 23 – I-IP Checks (source: Author).

To validate I-IP coverage, three test programs were developed. These programs were developed in order to analyze and validate the behavior of I-IP.

6.1.1 Non identified task running

Figure 24 presents the first test set; In this test was implemented five tasks in HellfireOs that should execute without losing their deadline; and, in I-IP just four tasks are mapped. The main goal here is to validate the Error: Unknown task.

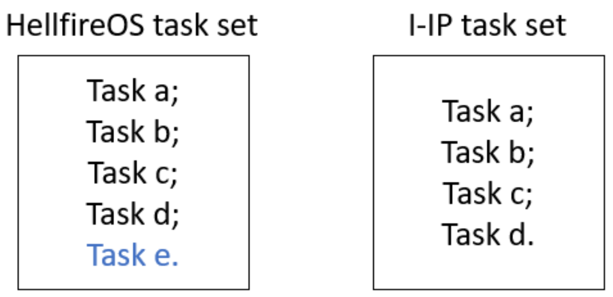


Figure 24 – Test set 1 (source: Author).

This check compares the running task with the expected tasks in I-IP. If the current task is not in the list of tasks, one error is generated to indicate that one unexpected task is in execution.

In this case, a new task was included in the HellfireOS, but was not mapped in the I-IP module. Once the unexpected task assumes the processor, the I-IP raises the MISS signal (100), meaning that some unknown task is being executed.

6.1.2 Deadline missed

This check verifies if some task has its deadline violated; if the task could not complete the execution time before the period arises its deadline is missed. The MISS signal shows this message error. This case considers the task set from Table 4:

Task	Ci	Ti	Di
T1	10	20	20
T2	35	50	50

Table 4 – Task parameters.

Applying Equation 2.2 for tasks from Table 4, and checking the utilization (U <= 1), it is possible to observe that some task will have the deadline missed.

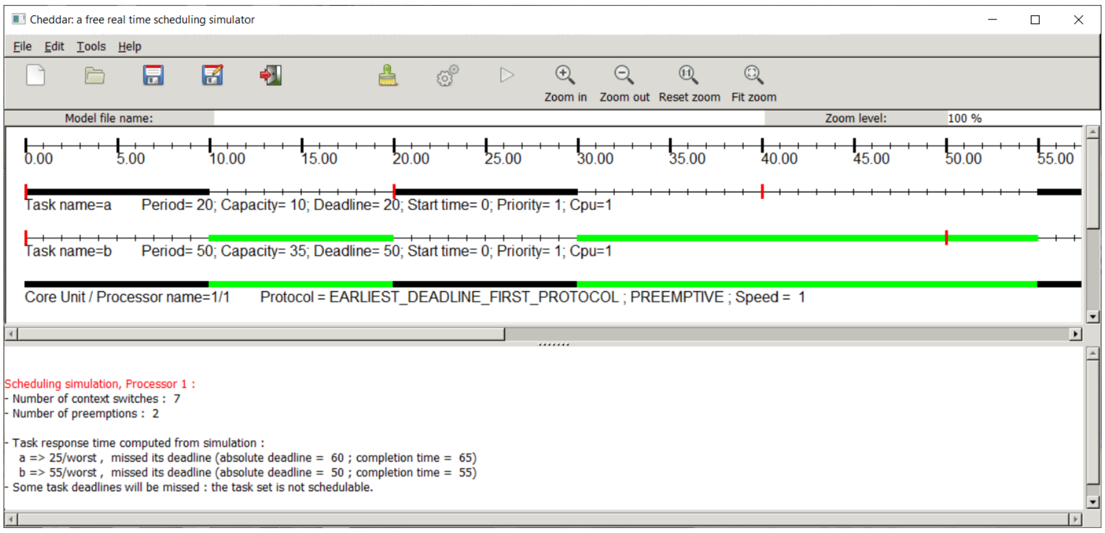
$$U = \sum_{i1}^{n} \frac{Ci}{Ti}$$

$$U = \frac{C1}{T1} + \frac{C2}{T2}$$

$$U = \frac{10}{20} + \frac{35}{50}$$

$$1.2 <= 1$$

When tasks from Table 4 are scheduled by the Earliest Deadline First algorithm, it is possible to observe that Task 'a' and 'b' will miss the deadline. In time 50, Task 'b' is available again; consequently, the period 'Ti' is reached. However, only in time 55, the task completes the total execution time. Also, for this task set, Task 'a' will lose the deadline. Since Task 'a' has 'Ti' = 20, in time 60 Task 'a' is available for the third time, which means, the period 'Ti' is reached again. However, only in time 65 the Task 'a' will complete the total execution time, as demonstrated in Figure 25.



```
Task name=a      Period= 20; Capacity= 10; Deadline= 20; Start time= 0; Priority= 1; Cpu=1
Task name=b      Period= 50; Capacity= 35; Deadline= 50; Start time= 0; Priority= 1; Cpu=1
Core Unit / Processor name=1/1      Protocol = EARLIEST_DEADLINE_FIRST_PROTOCOL ; PREEMPTIVE ; Speed = 1

Scheduling simulation, Processor 1 :
- Number of context switches :  7
- Number of preemptions :  2

- Task response time computed from simulation :
   a => 25/worst ,  missed its deadline (absolute deadline =  60 ; completion time =  65)
   b => 55/worst ,  missed its deadline (absolute deadline =  50 ; completion time =  55)
- Some task deadlines will be missed : the task set is not schedulable.
```

Figure 25 – Task set of Table 3 scheduled by Cheddar (source: Author).

Assuming the task set from Table 4, tasks a and b will have the deadline missed and, the I-IP raises the MISS signal (010).

6.1.3 Scheduling Error

Other errors that can occur during the operating system execution can be rescheduling or priority inversion, for example. In this case, a fault occurs either if rescheduling takes place, but the highest priority task does not assume the processor or a scheduling event does not occur and, unexpectedly, another task assumes the execution. Additionally, the I-IP verifies if scheduling events that occurred were executed, that is, that the execution task was indeed switched (unless there is no other task in the list of tasks ready to be executed with priority higher than the one under execution). This check may return an error for two reasons: (i) a task assumed the processor without a rescheduling event, or (ii) a rescheduling event occurred, but the task did not release the processor.

I-IP also checks whether the current task is the highest priority available. That means, I-IP checks the priority of the tasks ready with the current task, in order to guarantee that the current task is the expected one; if not, the MISS signal will also yield an error indication for this case of priority inversion.

In this case, to simulate a priority inversion, after calling the function sched_edf() a lower priority task was called. Once the task with low priority assumes the processor, the I-IP raises the MISS signal (011), meaning that a task with low priority is being executed.

6.1.4 Earlier Scheduling Error detection

As demonstrated in Chapter 2, for a set of tasks to be scaled with EDF and have all deadlines met, Equation 2.2 should be respected; otherwise, the deadline will be missed. In this scenario, I-IP checks Equation 2.2 to identify if any task has its deadline missed before the OS boot. If so, the MISS signal is raised (001), identifying that a task will have the deadline missed. This test was performed using task set from Table 4.

The verification of Equation 2.2 yields a "warning" indication. Nevertheless, it does not prevent the operating system from executing all tasks.

6.2 Error Latency Detection, Area Overhead, and Power Dissipation

The error latency detection depends on the points where the error is generated and also, the number of tasks that will have to be compared. For example, if I-IP has to check the status of two tasks, the process will take six clock cycles to raise the 'MISS' signal in case of one of the deadlines is missed.

Table 5 demonstrates an FPGA area overhead of 7,83% when adding I-IP to the system.

	HF_RISCV	HF_RISCV + I-IP	Overhad
Total Number of 4 input LUTs	2,208	2,381	7,83%

Table 5 – "I-IP: FPGA area overhead.

Furthermore, Table 6 demonstrates that adding I-IP to the system increase 1,21% the FPGA power dissipation.

	HF_RISCV	HF_RISCV + I-IP
Power dissipation	82mW	83mW

Table 6 – I-IP: Power consumption.

6.3 Results

I-IP can detect the missed deadline during the OS execution and identify scheduling errors. Comparing with the validation present in HellFireOS, I-IP has the highest coverage; once HellFireOS just cover the deadline misses.

Additionally, I-IP checks if the task set that the OS will execute can be scheduled without violating deadlines; this validation occurs right before the OS boot.

Therefore, if some task has a deadline missed or some scheduling error happens during the execution, I-IP will detect and update the MISS signal. In addition, before this happens, it is possible to know that some tasks can have the deadline missed and, this will be informed before the execution starts.

7 CONCLUSION

This work described a hardware-based technique to detect faults occurring during the operating system execution in real-time embedded systems. More precisely, these faults occur during the execution of the task scheduling algorithm. The greatest motivation of this work is due to the increasing use of embedded systems in the daily routine and the gap in the monitoring of scheduling failures when it comes to dynamic priority scheduling algorithms for real-time embedded systems.

The main contribution of this work is the robustness increase of embedded systems since the I-IP detects errors that are not covered by the native functions of the RTOS during the task scheduling analysis. Moreover, dynamic scheduling algorithms can scale some sets of tasks that would previously lose deadline, thus having a higher coverage.

The study was developed for the HF-RISC microprocessor and its HellFireOS operating system. In conclusion, the I-IP can detect if any task will have the deadline missed earlier than HellFireOS. I-IP can detect this condition before the processor starts running. It is worth noting that the I-IP can detect in real-time, i.e., with negligible delay, the missed deadline for each task, in addition to detecting scheduling failures.

8 FUTURE WORK

As a suggestion for future work, tests can be carried out on the FPGA board with radiation tests. Also, the main limitation of this I-IP module is the impossibility of identifying the resources that are being released or blocked. Identifying such RTOS resources will increase the detection capability of I-IP, as it will be possible to identify which resource each task is waiting for. Then, when a resource is released, it can be identified which task will be unlocked. Another suggestion for future work is implementing I-IP module to monitor different operating systems and the performance of respective fault injection experiments to verify the module's reliability in these new contexts.

Bibliography

ANSARI, M. et al. Peak power management to meet thermal design power in fault-tolerant embedded systems. *IEEE Transactions on Parallel and Distributed Systems, DOI: 10.1109/TPDS.2018.2858816*, v. 30, n. 1, p. 161–173, 2019.

ARLAT, J. et al. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computer, DOI: 10.1109/TC.2003.1228509*, 2003.

BETTI, E. et al. Real-time i/o management system with cots peripherals. *IEEE Trans. Comput., IEEE Computer Society, DOI: 10.1109/TC.2011.202*, v. 62, n. 1, p. 45–58, 2013.

BOYD-WICKIZER, S. et al. An analysis of linux scalability to many cores. in: Proceedings of the 9th usenix conference on operating systems design and implementation. *9th USENIX conference on Operating systems design and implementation, DOI: 10.5555/1924943.1924944*, p. 1–8, 2010.

COSTA, C. M. da. *Sistemas Operacionais – Programação concorrente com Pthreads*. 1. ed. Porto Alegre: EDIPUCRS, 2010.

DAVIS, R. I.; BURNS, A. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv., DOI: 10.1145/2379776.2379780*, 2011.

ERCIYES, K. *Uniprocessor-Independent Task Scheduling. In: Distributed Real-Time Systems. Computer Communications and Networks.* 1. ed. Switzerland: Springer, Cham, DOI: 10.1007/978-3-030-22570-4, 2019.

FARINES J.; FRAGA, J. O. R. *Sistemas de tempo real*. Florianópolis: [s.n.], 2000.

FILHO S. J.; AGUIAR, A. M. F. G. L. O. H. F. *HellfireOS Realtime Operating System*. 2007. <https://github.com/sjohann81/hellfireos>. [Online; accessed 19-May-2019].

GRACIOLI, G.; FROHLICH, A.; PELLIZZONI, R. Implementation and evaluation of global and partitioned scheduling in a real-time os. *University Of Waterloo*, 2013.

IGNAT, N. et al. Analysis of real-time systems sensitivity to transient faults using microc kernel. *IEEE Transactions on Nuclear Science, DOI: 10.1109/TNS.2006.880940*, v. 53, n. 4, 2006.

IGNAT, N. et al. Soft-error classification and impact analysis on real-time operating systems. *IEEE Design, Automation and Test in Europe, DOI: 10.1109/DATE.2006.244063*, 2006.

IZOSIMOV, V. et al. Design optimization of time- and cost constrained fault-tolerant distributed embedded systems. *IEEE Desgin Automation and Test in Europe, DOI: 10.1109/DATE.2005.116*, p. 864–869, 2005.

JACK, G. Great watchdogs. *Gaanssel Group*, 2004.

JOHANN S. F.; MOREIRA, M. T. C. N. L. V. H. F. P. The hf-risc processor: Performance assessment. *LASCAS, DOI: 10.1109/LASCAS.2016.7451018*, 2016.

JUHáSZ, T. D.; PLETL, S.; MOLNAR, L. A method for designing and implementing a real-time operating system for industrial devices. p. 149–154, 2019.

LAPRIE, J. C. Dependable computing and fault-tolerance: Concepts and terminology. *IEEE Proceedings, DOI: 10.1109/FTCSH.1995.532603*, 1985.

MICCO L.; VARGAS, F. F. P. Guest editorial special issue on embedded systems. *IEEE Latin America Transactions, Vol. 18, Issue 02, Feb. 2020: Special Issue on Embedded Systems, p. 180-187, ISSN 1548-0992, DOI: 10.1109/TLA.2020.9085270*, 2020.

MICCO L.; VARGAS, F. F. P. A literature review on embedded systems", ieee latin america transactions. *IEEE Latin America Transactions, Vol. 18 , Issue 02, Feb. 2020: Special Issue on Embedded Systems, p. 188-205. ISSN 1548-0992, DOI: 10.1109/TLA.2020.9085271*, 2020.

NICOLESCU, N. et al. Sensitivity of real-time operating systems to transient faults: A case study for microc kernel. *RADECS Radiation and Its Effects on Components and Systems, DOI: 10.1109/RADECS.2005.4365596*, p. 19–23, 2005.

OLIVEIRA, C.; BOLZANI, L.; VARGAS, F. A hardware-scheduler for fault detection in rtos-based embedded systems. *In Digital System Design, Architectures, Methods and Tools, 12th Euromicro Conference, DOI: 10.5555/1674636*, p. 10–13, 2015.

PRADHAN, D. K. Fault tolerant computer system design. *Prentice-Hall, DOI: 10.5555/230303*, 1996.

QING, L.; YAO, C. Real-time concepts for embedded systems. *CMP Books*, 2003.

RAHEJA R.; CHENG, S. G. D. S. B. Improving architecture-based self-adaptation using preemption. *SOAR, DOI: 10.1007/978-3-642-14412-7*$_2$, 2009.

SCHERRER, C.; STEININGER, A. Dealing with dormant faults in an embedded fault-tolerant computer system. *IEEE Transactions on Reliability, DOI: 10.1109/TR.2003.821943*, v. 52, n. 4, p. 512–522, 2003.

SHYE, A. et al. Plr: A software approach to transient fault tolerance for multicore architectures," ieee transactions on dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing, DOI: 10.1109/TDSC.2008.62*, p. 135–148, 2009.

SILVA, D.; BOLZANI, L.; VARGAS, F. An intellectual property core to detect task scheduling-related faults in rtos-based embedded systems. *IEEE 17th Int. On-Line Testing Symposium (IOLTS), DOI; 10.1109/IOLTS.2011.5993805*, p. 19–24, 2011.

SINGHOFF F.; LEGRAND, J. N. L. M. L. Cheddar : a flexible real time scheduling framework. *ACM SIGAda Ada Letters, DOI: 10.1145/1046191.1032298*, 2004.

STANKOVIC, J.; RAMAMRITHAM, K. Tutorial on hard real-time systems. *IEEE Computer Society Press*, 1988.

STANKOVIC, J. A. Misconceptions about real-time computing. *IEEE Computer, DOI: 10.1109/2.7053*, v. 21, 1988.

TARRILO, J.; BOLZANI, L.; VARGAS, F. On-chip watchdog to monitor rtos activity in mpsoc exposed to noisy environment. *10th International Workshop on the Electromagnetic Compatibility of Integrated Circuits (EMC Compo), DOI: 10.1109/MEMC.0.7543957*, p. 341–347, 2009.

ZHURAVLEV, S. et al. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys, DOI: 10.1145/2379776.2379780*, n. 4, 2012.

A CPU.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity processor is
port (          clk_i:       in std_logic;
                rst_i:       in std_logic;
                stall_i:     in std_logic;
                addr_o:      out std_logic_vector(31 downto 0);
                data_i:      in std_logic_vector(31 downto 0);
                data_o:      out std_logic_vector(31 downto 0);
                data_w_o:    out std_logic_vector(3 downto 0);
                extio_in:    in std_logic_vector(7 downto 0);
                extio_out:   out std_logic_vector(7 downto 0)
        );
end processor;

architecture arch_processor of processor is
        signal irq_cpu, irq_ack_cpu, exception_cpu, data_b_cpu, data_h_cpu,
  data_access_cpu: std_logic;
        signal irq_vector_cpu, address_cpu, data_in_cpu, data_out_cpu:
  std_logic_vector(31 downto 0);
        signal data_w_cpu: std_logic_vector(3 downto 0);
begin
        -- HF-RISC core
        core: entity work.datapath
        port map(        clock => clk_i,
                        reset => rst_i,
                        stall => stall_i,
                        irq_vector => irq_vector_cpu,
                        irq => irq_cpu,
                        irq_ack => irq_ack_cpu,
                        exception => exception_cpu,
                        address => address_cpu,
                        data_in => data_in_cpu,
                        data_out => data_out_cpu,
                        data_w => data_w_cpu,
                        data_b => data_b_cpu,
                        data_h => data_h_cpu,
                        data_access => data_access_cpu
        );

        -- interrupt controller
        int_control: entity work.interrupt_controller
        port map(
                clock => clk_i,
                reset => rst_i,
                stall => stall_i,
                irq_vector_cpu => irq_vector_cpu,
                irq_cpu => irq_cpu,
                irq_ack_cpu => irq_ack_cpu,
                exception_cpu => exception_cpu,
                address_cpu => address_cpu,
                data_in_cpu => data_in_cpu,
                data_out_cpu => data_out_cpu,
                data_w_cpu => data_w_cpu,
                data_access_cpu => data_access_cpu,
                addr_mem => addr_o,
                data_read_mem => data_i,
                data_write_mem => data_o,
                data_we_mem => data_w_o,
                extio_in => extio_in,
                extio_out => extio_out
        );

        --watchdog
        watchdog: entity work.watchdog
        port map(
                clk => clk_i,
                reset => rst_i,
                addr_from_cpu => address_cpu,
                data_access_cpu => data_access_cpu,
                irq_to_cpu => irq_cpu
        );
end arch_processor;
```