UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITA' DEGLI STUDI DI MODENA E REGGIO EMILIA
Dipartimento di Scienze Fisiche, Informatiche e Matematiche

Corso di laurea in informatica

Integrating and Analysing a Head-up Display Application

Relatore:
Bertogna Marko

Secondo Relatore:
Rouxel Benjamin

Correlatore:
Bouquillon Fabien

Laureando:
Bondioli Davide

Anno Accademico 2024-2025

Abstract

Real-time systems are increasingly prevalent across a wide range of domains, including automotive, robotics, and embedded applications. These systems must meet strict timing constraints, which can vary based on the application, ranging from hard real-time requirements in safety-critical environments to softer constraints where safety is less of a concern. Regardless of the specific timing requirements, precise analyses are essential to ensure system stability and reliability. This is particularly crucial in scenarios where missing deadlines can lead to system failures or unsafe outcomes.

This thesis presents a research project conducted on an ARM platform within the field of real-time embedded systems, focusing on an augmented reality head-up display (AR-HUD) as a case study. We explore the development process, detailing the necessary code components to build and execute the system.

After building a working system we analyze the response times of the application and perform a schedulability analysis to evaluate its stability and ensure it meets the required safety and performance constraints.

Contents

INTRODUCTION *9*

BACKGROUND *13*

2.1. Tasks, jobs and deadline 13
2.1.1. *Tasks and jobs* *13*
2.1.2. *Non-Real-Time tasks* *14*
2.1.3. *Real-Time tasks* *15*
2.1.4. *Deadline* *16*
2.2. Introduction to Real-Time Systems 16
2.2.1. *Types of Real-Time Systems* *17*
2.2.2. *Certifications and standards* *18*
2.3. Worst-Case Execution Time 19
2.3.1. *Importance of the WCET* *19*
2.3.2. *WCET on real-time systems* *20*
2.3.3. *WCET analysis* *21*
2.4. Scheduling 24
2.4.1. *Scheduler* *25*
2.4.2. *Scheduling categories* *25*
2.4.3. *Schedulability analysis* *34*

YASMIN *39*

3.1. Design 41
3.2. Implementation 42
3.2.1. *Overview of YASMIN API* *43*
3.3. Functioning 44
3.3.1. *On-Line Scheduling and Task Management* *45*
3.3.2. *Scheduling Modes: Global vs. Partitioned* *45*
3.3.3. *Ready Queue Management* *47*
3.4. Cecile 48

PROJECT *51*

4.1. Hardware platform 52
4.2. ORB-SLAM 53
4.3. Head Pose Estimation 55

4.4. IMPLEMENTATION AND INTEGRATION 58
4.5. INITIALIZATION STEPS 58
4.6. TASK AND PERIOD IDENTIFICATION 59
4.7. YASMIN AND CECILE MANAGEMENT 59
4.8. HOPENET ALONE 61
4.8.1. WCRT estimation – Baseline metric 61
4.8.2. WCRT estimation – with schedulers 65
4.8.3. Checking the caching system 67
4.9. ORB-SLAM ALONE 69
4.9.1. WCRT estimation – Baseline metric 70
4.9.2. WCRT estimation – with schedulers 71
4.9.3. WCRT estimation – with scheduler and flushing 74
4.10. UNIFIED APPLICATION 75
4.11. SCHEDULABILITY ANALYSIS 78
4.11.1. Tool overview 78
4.11.2. Task model 79
4.11.3. File construction 79
4.11.4. Tool run and Results 80
CONCLUSION...83
5.1. FUTURE WORKS 83
BIBLIOGRAPHY...85

List of figures

FIG. 1. TASK COMPOSITION 14
FIG. 2. NON-REAL-TIME PARAMETERS 14
FIG. 3. REAL-TIME PARAMETERS 15
FIG. 4. REAL-TIME SYSTEMS DISTRIBUTION 18
FIG. 5. EXECUTION TIME DISTRIBUTION 20
FIG. 6. SCHEDULING EXECUTED WITH FIFO 28
FIG. 7. SCHEDULING EXECUTED WITH RR 29
FIG. 8. RATE MONOTONIC SCHEDULING 30
FIG. 9. DEADLINE MONOTONIC SCHEDULING 31
FIG. 10. EARLIEST DEADLINE FIRST SCHEDULING 33
FIG. 11. DIFFERENCE BETWEEN LUB AND HB 36
FIG. 12. GLOBAL ONLINE SCHEDULING STRATEGY 46
FIG. 13. PARTITIONED ONLINE SCHEDULING STRATEGY 47
FIG. 14. COORDINATION COMPILER WORKFLOW 49
FIG. 15. AR HUD DIAGRAM 52
FIG. 16. ORIN SYSTEM-ON-CHIP (SOC) BLOCK DIAGRAM 53
FIG. 17. TASK GRAPH OF ORB-SLAM3 54
FIG. 18. TASK GRAPH OF HOPENET 56
FIG. 19. WCRT DEADLINE MISSES FOR THE BASELINE CONFIGURATION 62
FIG. 20. AVERAGE EXECUTION TIMES TREND 63
FIG. 21. WCRT DEADLINE MISSES GRAPH WITH FULL PERFORMANCE 64
FIG. 22. AVERAGE EXECUTION TIMES TREND ON MAXIMUM PERFORMANCE 64
FIG. 23. WCRT DEADLINE MISSES GRAPH COMPARISON WITH SCHEDULERS 65
FIG. 24. WCRT DEADLINE MISSES GRAPH WITH DM SCHEDULER 66
FIG. 25 AVERAGE EXECUTION TIMES TREND WITH DM 67
FIG. 26. WCRT DEADLINE MISSES GRAPH WITH DM SCHEDULER AND FLUSHING THE CACHES 68

FIG. 27. AVERAGE EXECUTION TIMES TREND WITH DM AND FLUSHING....69
FIG. 28. WCRT DEADLINE MISSES GRAPH....70
FIG. 29. AVERAGE EXECUTION TIMES TREND....71
FIG. 30. WCRT DEADLINE MISSES GRAPH COMPARISON WITH SCHEDULERS....72
FIG. 31. WCRT DEADLINE MISSES GRAPH WITH DM SCHEDULER....73
FIG. 32. AVERAGE EXECUTION TIMES TREND WITH DM....73
FIG. 33. WCRT DEADLINE MISSES GRAPH WITH DM AND FLUSHING....74
FIG. 34. AVERAGE EXECUTION TIMES TREND WITH DM AND FLUSHING....75
FIG. 35. WCRT DEADLINE MISSES GRAPH COMPARISON WITH SCHEDULERS....76
FIG. 36. WCRT DEADLINE MISSES GRAPH WITH DM....77
FIG. 37. AVERAGE EXECUTION TIMES TREND WITH DM....78
FIG. 38. HOPENET SCHEDULABILITY RATIO GRAPH....80
FIG. 39. ORB-SLAM SCHEDULABILITY RATIO GRAPH....81
FIG. 40. UNIFICATION SCHEDULABILITY RATIO GRAPH....81

Chapter 1

Introduction

As the world becomes increasingly connected and automated, the demand for embedded and real-time systems is growing rapidly across many sectors. These systems are at the heart of crucial industries such as automotive, aerospace, robotics, and the Internet of Things (IoT), where tasks must not only be completed correctly but also within strict timing constraints. This rising complexity in real-world applications, alongside the massive influx of data, requires systems to operate both efficiently and predictably, often with safety-critical requirements[16]. Embedded systems now must execute a diverse set of tasks simultaneously, many of which have varying levels of importance and criticality. These tasks can range from those with hard deadlines, such as deploying airbags in a car accident or stabilizing an aircraft mid-flight, to less critical functions, like adjusting air conditioning in a vehicle. The challenge is to ensure that tasks with higher criticality (e.g., life-saving functions) are guaranteed to meet their timing requirements, while other, less critical tasks are still efficiently handled.

Real-Time Scheduling and Task Management

The need to handle multiple tasks, often with different timing constraints, introduces the concept of mixed-criticality systems, where each task has specific performance and safety requirements. Tasks in a real-time system are generally divided into categories based on their timing requirements[17]:

- Hard real-time tasks: Missing a deadline can result in system failure or catastrophic consequences, such as in the automotive sector with braking systems or in avionics with flight control systems.

- Soft real-time tasks: Missing a deadline degrades system performance but doesn't cause total failure, like in multimedia streaming or infotainment systems in vehicles

These tasks are scheduled according to well-defined scheduling algorithms. In real-time systems, ensuring the schedulability of tasks (determining whether all tasks will meet their deadlines under a given scheduling strategy) is essential for system reliability[18].

Worst-Case Execution Time (WCET) and Predictability

One of the core concepts in real-time systems is the Worst-Case Execution Time (WCET). For a system to be certified as safe and reliable, especially in critical domains, it is necessary to accurately determine the maximum time a task will take to execute under the worst conditions[19]. This involves analyzing hardware performance (e.g., cache misses, interrupts) and software behavior (e.g., loops, conditional branches). Determining WCET is crucial for schedulability analysis since it ensures that tasks with strict deadlines will not overrun their allocated time, even under worst-case scenarios. Accurate WCET estimations and robust WCET analyses are key for designing systems where real-time guarantees are needed.

Increasing Demand for High-Performance and Real-Time Systems

The growing complexity of applications, coupled with the rapid expansion of automation, autonomous vehicles, and connected devices, has driven the demand for high-performance embedded systems that can meet real-time deadlines. Modern systems must process increasingly large datasets and make real-time decisions, whether in an autonomous vehicle navigating a busy road or a robot performing tasks in a factory.

The proliferation of heterogeneous architectures, such as Systems-on-Chip (SoCs) that combine CPUs, GPUs, and other hardware accelerators, allows for the efficient handling of multiple workloads. These systems are designed to optimize both performance and energy efficiency, making them ideal for embedded real-time applications. However, with the co-location of multiple workloads on shared resources, there is an increased risk of resource contention (e.g., competition for memory or I/O access), which can affect the timing behavior of critical tasks. This further complicates the real-time scheduling problem, as the system must carefully manage these resources to prevent interference.

Challenges and the Road Ahead

As embedded and real-time systems continue to evolve, ensuring the schedulability, predictability, and reliability of tasks in systems with varying criticality levels is one of the biggest challenges facing system designers today. Achieving this balance requires advanced scheduling techniques, accurate WCET analysis, and effective partitioning of system resources. At the same time, systems must also meet the stringent requirements of industry standards and certifications to guarantee safety in critical applications.
The future will likely see the rise of autonomous systems and cyber-physical systems, further increasing the demand for real-time, high-performance embedded computing. This will push the boundaries of current scheduling and real-time analysis techniques, requiring innovative solutions to manage the complexity of these systems in safety-critical environments.

Outline

This thesis aims to challenge the analysis and evaluation of the stability of different algorithms and to evaluate the schedulability and feasibility of the taskset.
First, we introduce some details about the real-time world. The second part introduces some useful details about the used tools. The last part discusses about the project itself, describing the hardware and software used, and passing from the beginning on building the system to the evaluation of the stability and the schedulability of the taskset.

Chapter 2

Background

In this chapter we introduce the real-time systems and describe what is the Worst-Case Execution Time (WCET) and the importance of this kind of estimation on systems that have timing constraints.
We give a rapid overview of the sectors related to Hard Real-Time, the certifications and standards.
We also introduce some techniques to improve the WCET estimation and some classifications to compute bounds on the execution time.

2.1. Tasks, jobs and deadline

When we speak about the real-time world, the first things we need to introduce are the tasks and the jobs. One of the most important components for real-time systems.

2.1.1. Tasks and jobs

A task is a sequence of instructions that in absence of other activities is continuously executed by the processor until completion.
A task is the composition of several sequences of instances called jobs, each job represents a component that contributes to the completion of a task.

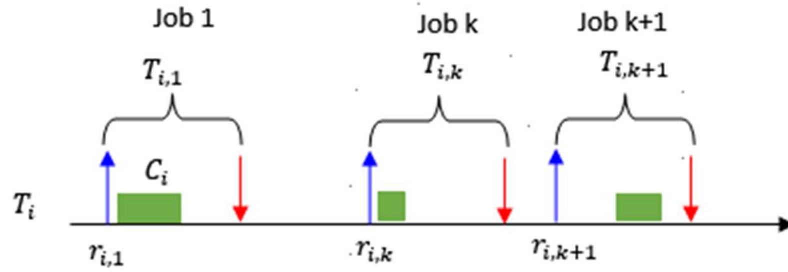

Fig. 1. Task composition

Figure 1 shows the jobs that are composing a single task, and that need to be executed to complete a task.

Here we can see T_i that is the task i, composed of $k + 1$ jobs, each computing for a time in green C_i and arriving at the time $r_{i,k}$.
There are two kinds of tasks:

- Non-real-time tasks: tasks which are not associated with the timing constraint, they don't have time bounds.
- Real-time tasks: tasks scheduled to finish all the computation events into timing constraints.

2.1.2. Non-Real-Time tasks

Non-Real-Time tasks are tasks which are not associated with the timing constraint and are not described by timing expressions.
They are not associated with any time bounds and can be completed at any time without serious consequences.

Fig. 2. Non-Real-Time parameters

On Figure 2, we can see the parameters for a Non-Real-Time task.

They have three parameters:

- Arrival time (r_i): when the task arrives and is ready to execute.

- Start time (s_i): when the task begins to execute.
- Finishing time (f_i): when the task completes all its jobs.

2.1.3. Real-Time tasks

Real-time tasks are tasks associated with the quantitative expression of time. This quantitative expression of time describes the behavior of the real-time tasks.
They have strict timing requirements and must be executed within specific time limits.

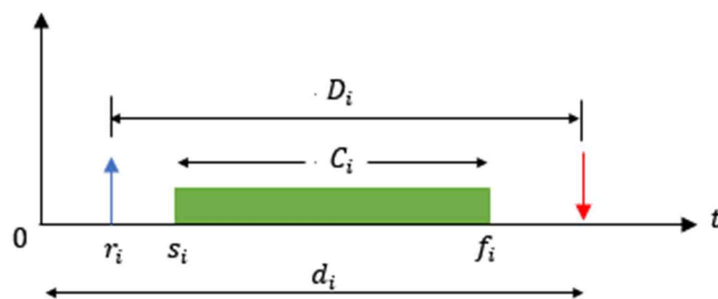

Fig. 3. Real-Time parameters

On Figure 3, we can see the parameters that differentiate a normal task from a Real-Time task.

The different parameters for Real-Time tasks are:

- Request time/Arrival time (r_i): it's the time when the job arrives and it can be scheduled.
- Start time (s_i): the time when the job is scheduled.
- Finishing time (f_i): the time when a job finishes its execution.
- Worst-case execution time (C_i): the execution time in the worst case.
- Relative deadline (D_i): the time from the request time to the deadline.
- Absolute deadline (d_i): the time from the axis origin to the deadline.

2.1.4. Deadline

As we can see on Figure 3, tasks on real-time applications have a parameter called deadline.
The deadline is a specific point in time where the tasks must be completed to meet the real-time timing requirements. Missing the deadlines may have serious consequences.
Common schemes for a task's deadline are:

- Implicit Deadlines ($D = T$): Where the deadline equals the period, making the task's deadline coincide with the next activation time.
- Constrained Deadlines ($D \leq T$): Where the deadline is less than or equal to the period, allowing tasks to complete before the start of the next period.
- Arbitrary Deadlines: Where the deadline can be independent of the task's period, providing maximum flexibility in scheduling tasks with varying time constraints.

2.2. Introduction to Real-Time Systems

A Real-Time System is a computational system that must execute within time constraints, it is composed of tasks and is characterized by its ability to produce the expected results within a defined deadline.

Characteristics of the real-time systems are:

- Timing constraints: specific timing requirements for the task completion.

- Predictability: All the response times must be consistent and predictable.

- Reliability: High availability and robustness are essential.

Indeed in those kinds of systems, failures to respect the pre-established times can have more or less serious consequences.
Thus, providing guarantees on the respect of the timing constraints for such systems is mandatory.

If we think about the systems that we found on cars, nuclear plants, planes, … they are all Real-Time Systems, typically Embedded Systems that control a much bigger system.
They need to be reliable not only on logic correctness, the absence of functional bugs, but also on timing correctness, to have an on-time response.

Giving an input or an event to the system, the output must be less than the response time, to have a predictable and stable response. A delayed response can be useless and possibly dangerous.

2.2.1. Types of Real-Time Systems

There are three types of Real-Time Systems, all with a different field of application and with different operational needs:

- Hard Real-Time Systems: all those systems that are safety critical, everything must be really predictable since the miss of a deadline can have catastrophic consequences like death or serious damages to structures.

  Examples are: nuclear centrals, avionic systems, medical systems,... .

- Firm Real-Time Systems: they are systems that are less rigid to deadline miss. In those systems the miss of a deadline does not result in damages but the result most of the time is discarded.

  Examples are: bank transactions, high frequency trading, automated production lines, … .

- Soft Real-Time Systems: In those systems the miss of a deadline results in a degradation of the service. Here we found the systems that have more performance but can be less predictable.

  Examples are: streaming, traffic management, online games, … .

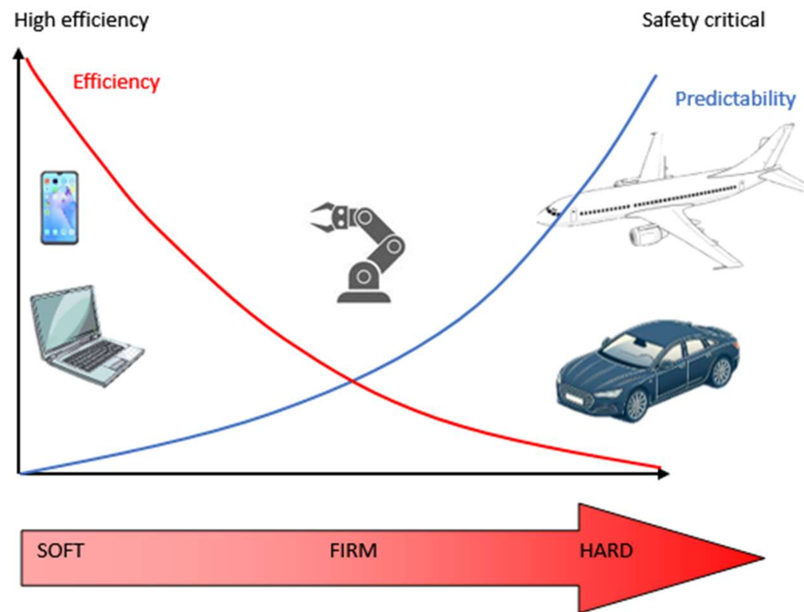

Fig. 4. Real-Time Systems distribution

In Figure 4 we can see a representation of what is much more required in the different kinds of Real-Time systems.

As we can see a Soft-Critical System needs to have performance but less predictable, if we miss a frame in a video stream it is not so critical.

On the other hand Hard-Critical Systems needs to be much more predictable and can be less efficient. We are dealing with human lives and we need to know exactly what the system is going to do.

In the middle we have the Firm-Critical Systems, that need a combination of performance and predictability, they need to be safe but also have performance.

2.2.2. Certifications and standards

The most crucial sectors that need to guarantee low risk of safety, due to working with human lives, have developed some regulations and standards that need the WCET to provide certifications.

The most relevant sectors and certifications are, for example:

- Avionic sector: where we found the DO-178C certification, that describes the requirements that the software must guarantee and the guarantee on timing executions to meet the operational safety[21].

- Automotive sector: ISO 26262 an international standard on electric and electronic systems of a road vehicle. This standard guarantees that all the temporal functions are respected[20].

- Railway sector: EN 50128 a European standard that specifies the requirements on developing and maintaining the software used on control and protection of the railway systems. This standard requires that the execution times meet the safety operative times[22].

Those are just a few certifications that a high-risk sector must guarantee when dealing with critical systems.

These standards require that designers and engineers look carefully on execution timings and guarantee that the system is strong on timing constraints. This is of fundamental importance for all the products that have to do with human lives.

2.3. Worst-Case Execution Time

The Worst-Case Execution Time (WCET) [10] is the maximum time that a program or an algorithm takes, varying the inputs and executing it on a specific architecture, to finish all the jobs.
This time is computed considering the worst possible scenario and the worst possible inputs that should trigger the WCET.

2.3.1. Importance of the WCET

The importance of the WCET is referred to:

- Safety and reliability: on critical systems, like automotives, avionics and others it is of crucial importance to have an “on time response” to prevent failures and avoid human loss.

- In this sector certifications are required to show that the system has a strong reliability and a strong safety against critical faults.

- Scheduling and resource handling: on multitasking systems, the scheduling of the tasks strongly depends on the WCET. This information is used to guarantee the deadlines of all the tasks and help handling the system resources.

- Optimization on the performance: the WCET can be analyzed and optimized to have a much more efficient system.

2.3.2. WCET on real-time systems

Real-time systems are computational systems that consist of a number of tasks that need to execute some operations within temporal constraints called deadlines.
A task typically shows a certain variation of execution times depending on the input or different behavior of the environment.
On a real-time system we must guarantee timing responses to be bounded and predictable, and we need this kind of warranties for each task and for all the possible events.
When dealing with timing constraints, such as deadlines, the WCET estimation is a must to perform a schedulability analysis.

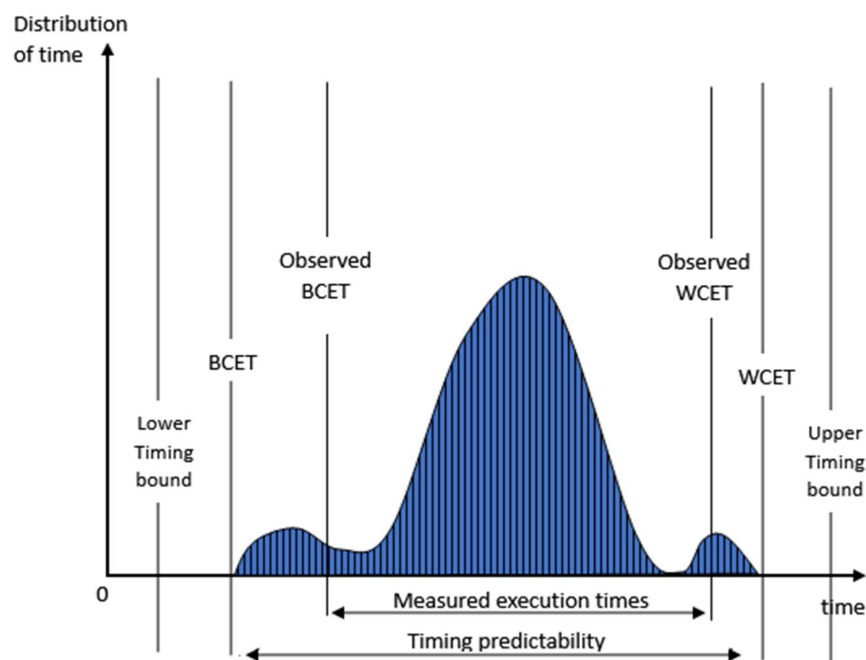

Fig. 5. Execution Time distribution

In Figure 5 we can see some relevant properties, the shortest execution time is called Best-Case Execution Time (BCET) and the longest time is called the Worst-Case Execution Time (WCET).
When we do a measurement based approach we measure the times to have a knowledge on the best and worst scenarios. This could lead to an observed execution time for the best and worst case that may not be the correct WCET and BCET.

We can also see some upper and lower timing bounds, these are the bounds that a task will for sure not be able to exceed even with the worst or best cases.

Also dealing with the innovations over the years introduced at the microarchitectural level to meet the ever-increasing demand for performance (cache, pipeline, etc..) can lead to a difficult estimation of the WCET and to some erroneous values.

On hard real-time systems all those problems must be resolved to have a safe and reliable system and need certifications to give guarantees.

2.3.3. WCET analysis

A WCET analysis is a method used to test the system to obtain an upper bound and a lower bound.
In general, we have three categories of WCET analysis:

- Static methods: these methods do not execute the code on a real hardware platform or on a simulator but on some model of the hardware, and combine this with the task code, analyze the control-flow paths and obtain an upper bound for the execution time.

- Measurement-based methods (or dynamic methods): these methods execute the task on the given hardware or a simulator for a set of chosen inputs and then take the measured times and derive the maximal and minimal observed execution times.

- Hybrid methods: these methods use some elements from both the previous analyses, combining them to obtain a much more precise and accurate estimation of the WCET.

2.3.3.1. Static methods

Static methods [12] compute bounds on the execution time guaranteeing that the execution time will not exceed the computed bounds.

The advantage of the statical methods is that it can be done without running the program to be analyzed, which often needs complex

equipment to simulate the hardware and peripherals of the target system.

They use the control-flow analysis to cover all possible execution paths and use abstraction to cover all possible context dependencies in the processor behavior. The price to pay for this safety is the necessity for processor-specific models and possibly imprecise results, such as overestimated WCET bounds.

The abstraction of the architecture of the system needs to be concrete in respect to the timing behavior but, in general, such information is hard to achieve, because not all computer vendors give enough information about the microarchitecture. Without such information any static WCET tool cannot be trusted.

Advantages:

- Full coverage: analyze all the possible executions, or flows, of the code.
- Safety: estimating the upper bound produces a safe WCET estimate, guaranteed to be higher than the actual unknown WCET.

Disadvantage:

- Conservative: trying to consider all the possible cases can overestimate the WCET.
- Complex hardware: if the system is too complex or we don’t have enough information about the system it is not possible to model accurately all the hardware components, this leads to imprecise and unsafe results.

2.3.3.2. Dynamic methods

The measurement-based methods [13] for estimating the best and worst case execution time use the real hardware to execute and run the program while measuring the times it takes to finish. The time is measured with always the same dataset of input varying from the worst input to the best input cases.

They use control-flow analysis to include all possible execution paths and produce WCET and BCET estimates that are more precise than the bounds from static methods, especially for complex processors and complex applications but can be difficult to cover all the possible control flows of the program.

Still, since the exact WCET or BCET are usually not known, there is really no way to check how precise an estimate or bound is.

Advantages:

- Easy collect: measure the effective execution time on the specific hardware and in specific scenarios

Disadvantage:

- Incomplete coverage: can be difficult to test all the possible execution paths of the code

- Test dependent: the quality of the estimation strongly depends on the quality and completeness of the tests

The common method to estimate execution time bounds is to record the execution times of the task for a subset of the possible executions/test cases, determining the minimal and maximal observed execution times, the bounds are obtained adding a safety margin (from 10-20%).
These will in general underestimate the BCET and overestimate the WCET and so are not safe for hard Real-Time systems.

2.3.3.3. Hybrid methods

The hybrid method for the estimation of the worst-case execution time combines the dynamic and the static analyses to obtain a much more accurate and balanced result. This approach tries to overcome the limits derived from the other methods.

Flow of the hybrid analysis:

- Initial static analysis: execute the static analysis to learn the control-flow and all the possible execution paths of the code.

- Execution of dynamic tests: take the effective execution times by executing the code on the real hardware.
- Correlation of the results: combine the dynamic tests to validate and improve the static ones to have a much more precise estimation on the WCET.

Advantages:

- More precision: combining the full coverage of the static methods and the accuracy of the dynamic methods leads to precise and balanced results.
- Reduction of conservatism: the combination of dynamic results reduces the exceed of estimation of the static analysis.
- Complex hardware: measure the effective execution time on complex hardware and in specific scenarios that are difficult to model statically.

Disadvantages:

- Complexity: combining static analysis and measurement-based approach can be challenging, particularly in ensuring consistency and accuracy.
- Time consuming: Hybrid analysis takes longer due to the need to perform both the analysis.
- Methodological challenges: developing reliable methodologies to combine static and measurement-based methods is an ongoing area of research, and current methods may not be fully robust.

2.4. Scheduling

Scheduling is a method used to distribute computing resources such as processor time, bandwidth and memory, to various tasks.
It is a process that determines which task should execute next, based on the system's resources and task's priorities and deadlines.

The scheduling process is complex but necessary to ensure that all tasks complete within their specific timing constraints.

The scheduling activity is carried out by a process called scheduler.

2.4.1. Scheduler

The scheduler is a critical component of the operating system that manages the allocation of resources to tasks. A scheduler is of fundamental importance to guarantee that the system resources are used efficiently and all the services run without any significant interruption.

The scheduler manages the allocation through a scheduling algorithm that decides which process runs at a certain point in time, retrieves access to resources and plans an order of execution with respect to a scheduling policy.

Schedulers produce a schedule, a particular assignment of the tasks to the processor.

This schedule must be able to manage the task allocation in respect to the timing constraints, if all the tasks are able to complete within the timing constraints, then the schedule is said to be feasible and the set of tasks is schedulable.

Schedulers need to be efficient trying to minimize the wait times, the response time, and to maximize the CPU usage and the throughput.

To be able to decide if a taskset is schedulable a programmer can use a method called schedulability analysis.

2.4.2. Scheduling categories

The scheduling algorithm is used to distribute resources among tasks which simultaneously and asynchronously request them.

The scheduling algorithm has an important role ensuring tasks meet their deadlines. The two main types of scheduling algorithms are:

- Preemptive: The scheduler has the ability to pause a running task and switch to a higher priority task. This is of fundamental

importance for ensuring that critical tasks meet their deadlines, even if a lower priority task is executing..

- Non-Preemptive: The scheduler selects which task executes first but the task keeps the resource until it finishes or interrupt by itself (it can't be preempted). This is important for tasks that need to run to completion without being interrupted, but in the real-time world this can lead to deadline misses for higher priority tasks.

Other properties are based on:

- The way it takes decisions:
  - On-line: All the scheduling decisions are taken at run time on the set of active tasks.
  - Off-line: All the scheduling decisions are taken before task activation by producing a schedule.
- The priority assignment:
  - Static: Scheduling decisions are taken based on fixed parameters, statically assigned to tasks before activation. The order of execution is statically assigned before the execution begins and can't be changed.
  - Dynamic: Scheduling decisions are taken based on parameters that can change in time. The order of execution can change at runtime.
- The schedule it produces:
  - Optimal: Always find a feasible schedule, if there exists one.
  - Best-effort: Do their best to find a feasible schedule, if there exists one, but they do not guarantee that.

There is no distinction between Non-Real-Time and Real-Time schedulers but some of them present better efficiency to respect the timings constraints of real-time systems than others.

Schedulers may not have strict deadlines to respect but other metrics to optimize like, throughput, average waiting time or response time.

Common schedulers are priority based methods that can be distinguished in:

- Fixed priority schedulers
- Deadline-based schedulers

The choice between one scheduler and another depends on the specific need of the real-time systems.

For example a system with a mix of critical and non-critical tasks might use a priority based scheduler. Systems that have hard deadlines might use a deadline-based scheduler.

2.4.2.1. Fixed Priority schedulers

Fixed Priority schedulers (FP) assign a priority to each task, the task with the highest priority is selected for execution, tasks with the same priority are served with FIFO or Round Robin.

The priorities may be assigned arbitrarily by the developer or a priority assignment algorithm can be used instead as for example RM and DM.

Common fixed priority schedulers are:

- First In First Out (FIFO)
- Round Robin (RR)
- Rate Monotonic (RM)
- Deadline Monotonic (DM)

First In First Out (FIFO)

FIFO is an algorithm that assigns the resources to tasks based on their arrival times.

Properties are:

- Dynamic: Based on arrival time that can change during execution.
- On-Line
- Best effort

It's simple to implement but can have problems with waiting times, especially if processes arrive late.

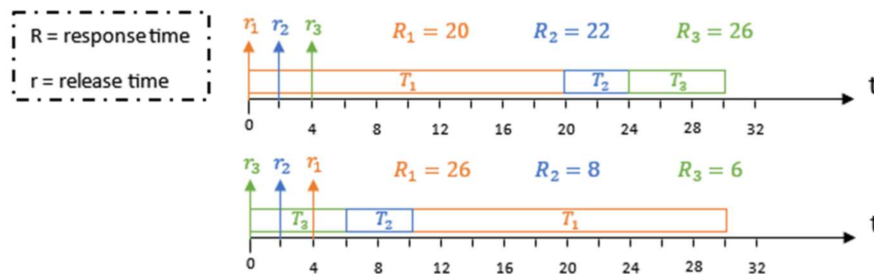

Fig. 6. Scheduling executed with FIFO

On Figure 6 we can see two schedules for the FIFO algorithm, here we can see the functioning of the FIFO scheduler, the first task to arrive is the one that is scheduled.

The first and second schedule on Figure 6, shows different release times for each task and consecutively a different schedule with FIFO. Also, we can see the huge difference in the response time caused by the difference in release time of the tasks.

Round Robin (RR)

The RR algorithm executes like FIFO, the first to arrive is also the one that is served, but the system as the concept of time quantum, each task cannot execute more than a quantum time unit. When the time quantum expires the task is preempted.

Properties are:

- Static or Dynamic: Based on the time quantum that is static but there is still a priority assignment that can be done dinamically.
- On-line

- Best-effort

It is a fair algorithm, but we need to pay attention in setting the time quantum, a short time quantum gives more overhead due to the context switch, a longer time quantum penalizes the response time.

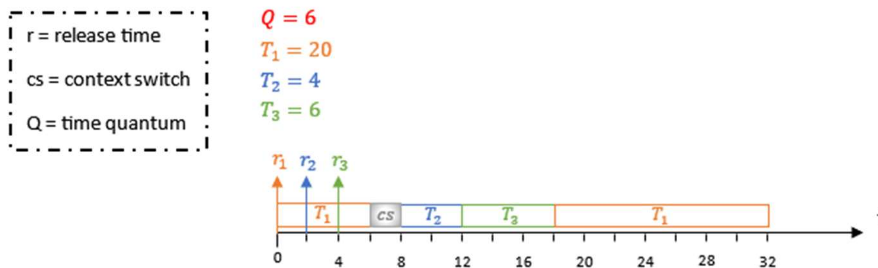

Fig. 7. Scheduling executed with RR

On Figure 7, we can see a schedule with the RR algorithm. RR is executing like FIFO but using a time quantum (Q) for each task.

When the task exceeds the time quantum, it is preempted with a little overhead, called the context switch (cs), and the execution passes at the next task.

Here on Figure 7, we can see the task in orange that arrives first and it starts to execute, but has an execution time (T) of 20, so it is executing for all the quantum of 6 and then is preempted.

The other task, the blue and the green one, arrives and they are the next to execute, both need to run for a time that is lower in respect of the time quantum, so they are executing completely.

At the end the orange task remains alone and it can execute completely.

This is just an example of how the RR scheduler works.

Rate Monotonic (RM)

The RM algorithm assigns to each task a fixed priority proportional to its rate, is specifically designed for Real-Time systems with periodic tasks.

Properties of RM are:

- Static: It is based on priorities assigned based on the periods of the tasks, these priorities do not change.
- On-line
- Optimal

Each task is assigned a priority based on their period, the shortest period gets the highest priority, this is because tasks with shorter periods need to be executed more frequently.

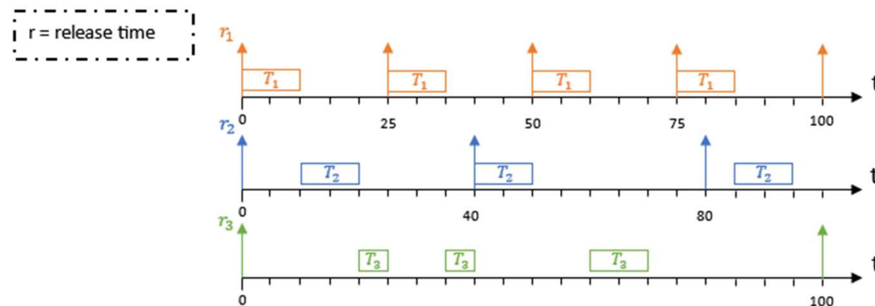

Fig. 8. Rate Monotonic scheduling

On Figure 8, we can see an example of execution of the RM algorithm, the priority is based on the rate, the less rate the higher priority the task has.

The 3 lines on Figure 8, shows 3 different tasks, in this example the tasks are not on the same graph just to show better the splitting of the execution time, it is not referred to as multi-core.

Here we can see that the orange task has a rate of 25, the blue one has a rate of 40 and the green of 100. In this case the first to execute with the RM scheduler is the orange one, that is the task with the higher priority, the second is the blue and the third the green one.

With this task configuration, in the third line, we can see that the execution of the higher priority tasks interferes with the green task and is so splitted on the timeline.

Deadline Monotonic (DM)

The DM algorithm is an extension of RM, which assigns to each task a fixed priority based on their deadlines, and is specifically designed for Real-Time systems with periodic tasks.

It's a static method that uses the relative deadline to schedule a taskset.

Properties of DM are:

- Static: It is based on priorities assigned based on the deadlines of the tasks, these priorities are fixed and do not change during the execution of the tasks.
- On-line
- Optimal

Each task is assigned a priority based on their deadlines, the task with the earliest deadline gets the highest priority, here we can see an example of a schedule with DM.

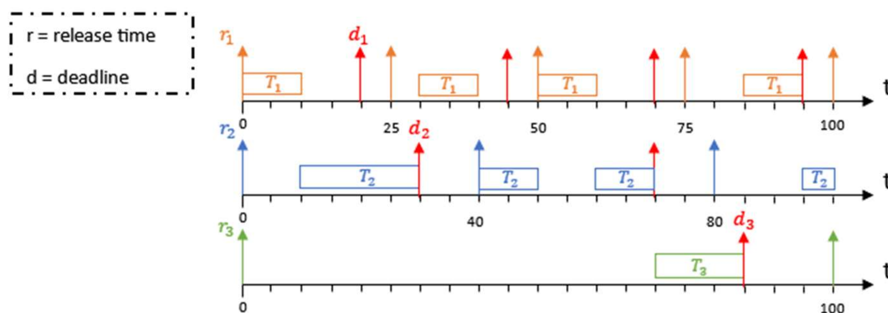

Fig. 9. Deadline Monotonic scheduling

On Figure 9 we can see an example of schedule for DM, the deadlines are different from the period and the schedule is done by the earliest deadlines, if two tasks have the same deadline then it can be scheduled with FIFO or by the earliest period.

Also here, we split the execution schedule on three timeline to better show the interference of all the tasks in the execution and doesn't mean we are on multi-core.

Here we can see that with DM the task with higher priority is the orange one, the one with the lower deadline, then the blue task has a lower priority and the green one has the lowest priority.

As we can see here the lower priority tasks are being preempted in favor of the higher priority tasks but all the tasks meet their deadlines.

2.4.2.2. Dynamic priority schedulers

Deadline Based schedulers assign each task a priority based on his deadline and the task with the higher priority is selected for the execution.

A common deadline-based scheduler is: Earliest Deadline First (EDF).

Earliest Deadline First (EDF)

The EDF algorithm assigns priorities based on absolute deadlines. The task with the earliest (closest) absolute deadline is given the highest priority and is scheduled to execute next.

Properties of EDF are:

- Dynamic: It is based on an absolute deadline that requires the arrival time that can change during execution.
- On-line
- Optimal

It can minimize the maximum lateness, and can be difficult to implement requiring a precise management of the execution times and deadlines.

It is an optimal algorithm for Hard Real-Time systems because it is based on the absolute deadline.

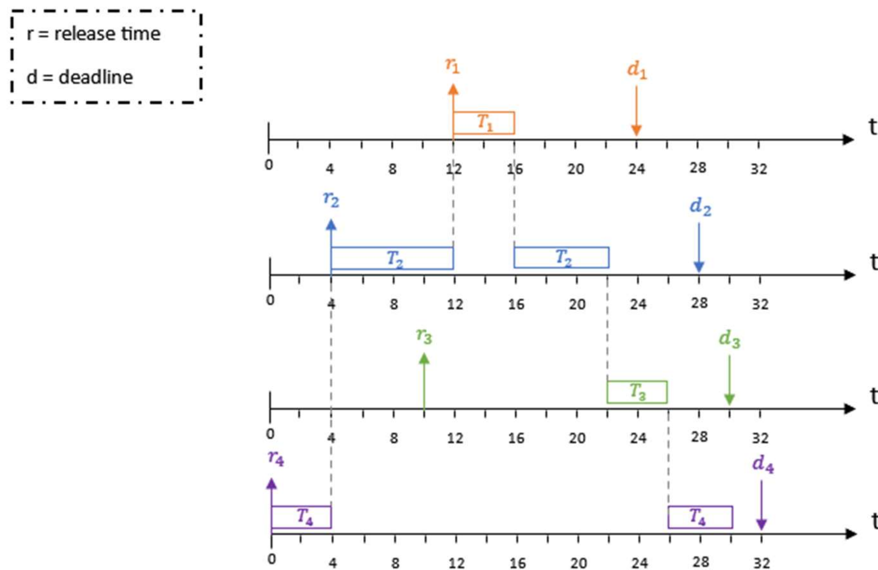

Fig. 10. Earliest Deadline First scheduling

On Figure 10 we can see an example of schedule for EDF, when a task arrives it's scheduled but if another task with higher priority, or better with earlier deadline arrives, the execution is preempted, also all the tasks with a higher deadline than the executing task are blocked.

To have a better understanding on the preemption the schedule is splitted into 4 timelines, one for each task, this doesn't mean that we are on multi-core.

The task's priority is defined by the earliest deadline, so the orange has the highest priority, the blue has a lower priority than orange, the green has a lower priority than the blue and the purple has the lowest priority.

Here on Figure 10 we see that the task in purple arrives first and is then executed until the blue one arrives and preempts the purple task. Also the green task arrives but it cannot execute because a higher priority task is executing.

At the end each task can continue the execution when the higher priority tasks end.

2.4.3. Schedulability analysis

Schedulability analysis is the process of determining whether a set of tasks (taskset) can be scheduled to meet their deadlines under a given scheduling algorithm. It is crucial in real-time systems where tasks must complete within specific time constraints, such as in embedded systems, automotive control systems, or avionics.

The goal is to ensure that every task in the system finishes before its deadline, even in the worst-case scenario.

Different scheduling algorithms, such as Rate Monotonic (RM), Deadline Monotonic (DM), and Earliest Deadline First (EDF), have their own methods for performing schedulability analysis. These analyses range from simple utilization tests to more complex methods like response-time analysis.

Now we introduce some schedulability analysis concepts for RM, DM and EDF, specifically in detail for DM.

2.4.3.1. Utilization bound

The simplest schedulability analysis for RM, DM and EDF is the utilization test. It ensures that the total processor utilization remains below a certain threshold.

Each task uses the processor for a fraction of time:

$$U_i = \frac{C_i}{T_i}$$

This is called the utilization (U_i), is the amount of time a task needs from the CPU to execute and is computed by dividing the computational time (C_i) by the period (T_i).

The total processor utilization (U_p) is the sum of all the tasks utilizations:

$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

This is also called the processor load, if $U_p > 1$ the processor is overloaded and the taskset cannot be scheduled.

There are two types of conditions that need to be guaranteed to have a schedulable taskset:

- Sufficient condition: is the condition that if satisfied, guarantees the schedulability.
- Necessary condition: is the condition that if not satisfied, do not permit the taskset to be schedulable.

Necessary condition for RM and DM is:

$$U_p < 1$$

This doesn't mean that the taskset is schedulable, but it can be.

For the sufficient condition, we need to introduce the Least Upper Bound:

$$U_{lub}^{RM} = n(2^{\frac{1}{n}} - 1)$$

If the utilization is under this condition, the taskset is schedulable:

$$U_p \leq U_{lub} \rightarrow schedulable$$

Another sufficient condition for RM and DM can be also the Hyperbolic bound:

$$\prod_{i=1}^{n}(U_i + 1) \leq 2$$

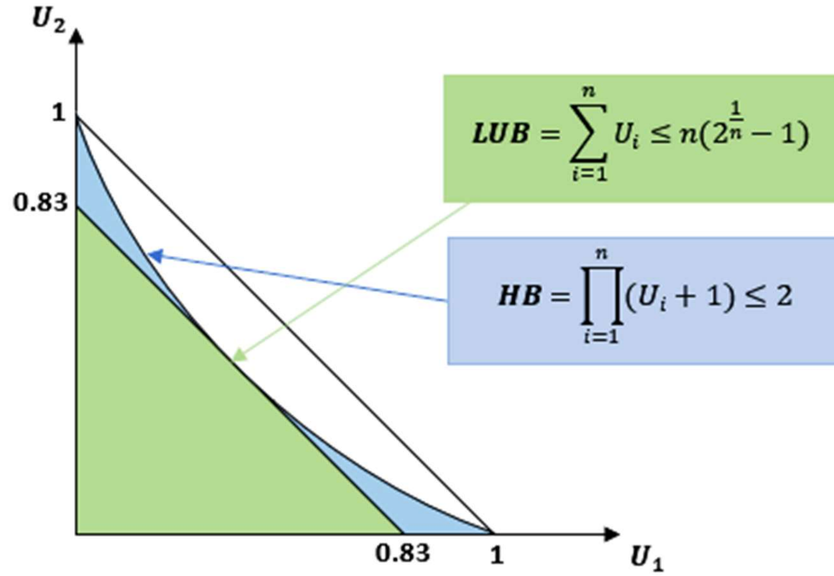

Fig. 11. Difference between LUB and HB

Figure 11, shows the solution space for a system of two tasks where the tests provide schedulability guaranties with the DM scheduler.

The difference between the Least Upper Bound (LUB) and the Hyperbolic Bound (HB) is shown on Figure 11, the HB has a relaxed bound, this means that it is more precise than the LUB.

The curve that the HB have, gives us more guarantees that a system is schedulable also reaching this zone.

This is true always for RM but not for DM, in fact for DM this is true only if the deadline is equal to the period.

When this condition lacks, the system may still be schedulable even if the utilization test fails.

In respect to DM and RM, the EDF scheduler shows a sufficient condition and a necessary condition that is:

$$U_p \leq 1$$

In other words, if the processor utilization is less than or equal to 100%, all tasks will meet their deadlines under EDF.

2.4.3.2. Response-Time Analysis

If the utilization test fails or if a more precise analysis is needed, for RM and DM, we can use the Response-Time Analysis (RTA)[14].

RTA calculates whether each task finishes before its deadline by considering interference from higher priority tasks. This involves:

- Calculating the Worst-Case Response Time: For each task, calculate the worst-case response time considering the interference from higher-priority tasks.

- Comparing with Deadlines: Ensure that the worst-case response time of each task is less than or equal to its deadline.

The response time R_i of a task i is calculated as:

$$R_i = C_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Where:

- C_i is the WCET of task i.
- $hp(i)$ is the set of higher-priority tasks.
- T_j is the period of a higher-priority task j.

Each task is schedulable if:

$$R_i \leq D_i$$

For EDF the RTA analysis is not useful but it can be used with the Demand Bound Function, we will not go any further with the schedulability analysis.

For other information and more technical data about schedulability analysis [6].

2.4.3.3. Schedule Abstraction Graph

The tool used, np-schedulability-analysis[5], is designed to analyze the schedulability of tasks in real-time systems, specifically in non-preemptive scheduling scenarios. The schedulability analysis assesses

whether a set of tasks can complete their execution within their deadlines under a given scheduling algorithm.

The tool focuses on non-preemptive scheduling, where once a task starts executing, it runs to completion without being interrupted by higher-priority tasks.

This tool builds a schedule abstraction graph [1] which is a graph that models all the possible states that a scheduler can encounter. As this state space can be huge, the authors of this method provide a technique [4] to merge some states and avoid exploring some branch of the graph that would inherently lead a state from another previously visited branch. In the end the last state explored to build the schedule abstraction graph will contain all the information to compute the worst-case response time of the taskset.

Chapter 3

YASMIN

YASMIN [7] is a middleware designed to schedule end-user applications with real-time requirements directly in user space, operating alongside the OS. It is versatile and can be deployed across a broad range of architectures, making it well-suited for managing heterogeneity on COTS (Commercial Off-The-Shelf) embedded platforms.

YASMIN offers a user-friendly programming interface while maintaining portability across various architectures. This flexibility allows designers to seamlessly work with different hardware setups without significant code changes.

Additionally, YASMIN supports multiple functionally equivalent task implementations, providing designers with the ability to quickly experiment with different scheduling policies and task-to-core mappings. This capability is essential for optimizing system performance, as it enables fine-tuning of the application to meet specific real-time requirements while fully leveraging the underlying hardware's capabilities. By offering these features, YASMIN simplifies the process of developing and deploying real-time applications in different and complex environments.

Yasmin matches most of the industrial needs, offering several key advantages:

- Customization: YASMIN is designed with a clear separation of concerns, including aspects like task mapping, scheduling,

and priority ordering. This modular approach makes it much easier and faster to use. By isolating different functionalities, developers can quickly customize the middleware to suit specific application requirements without altering the entire system.

- Adaptability: YASMIN's support for multiple versions of tasks allows users to dynamically adjust the behavior of applications at runtime, accommodating changing environmental conditions. YASMIN enables real-time adaptation, ensuring the application continues to perform optimally under varying conditions.

- Maintainability: YASMIN is not tied to any specific operating system or version. This independence simplifies maintenance and upgrades, as YASMIN can seamlessly integrate with new or updated OS environments without requiring significant modifications.

- Portability: Built to run on any POSIX-compliant OS, YASMIN is highly portable and can be deployed across a wide range of platforms. This flexibility ensures that developers can use YASMIN in various hardware environments without being constrained by platform-specific limitations.

- Compatibility: In scenarios where a specific platform lacks RTOS support, often due to vendor-specific drivers, YASMIN provides a valuable alternative. It offers more robust timing guarantees than a standard OS, such as soft real-time guarantees on a vanilla Linux system.

- Flexibility: YASMIN supports different workload packages, allowing for distinct configurations tailored to each package, including different task models and scheduling strategies. This flexibility enables developers to fine-tune the system according to the specific needs of different applications, maximizing performance and efficiency.

- Design Space Exploration: Selecting the optimal scheduling policy for a system is often a complex task. YASMIN facilitates this process by offering multiple scheduling options

that can be chosen at compile time. This feature allows both real-time experts and those with less experience to explore and experiment within the scheduling design space by selecting the policy that delivers the best performance for their specific use case.

YASMIN is a powerful tool for developing and managing real-time applications in a wide range of industrial environments. By addressing the specific challenges associated with heterogeneous and evolving systems, YASMIN provides a reliable and efficient solution that meets the high demands of modern embedded computing.

YASMIN provides several priority assignment strategies:

- Deadline Monotonic (DM)
- Rate Monotonic (RM)
- Earliest Deadline First (EDF)
- Fixed Priority (FP)

3.1. Design

YASMIN is designed to facilitate rapid prototyping and deployment of non-safety-critical real-time applications on heterogeneous parallel COTS platforms.

In the context of non-safety-critical real-time systems, YASMIN handles sporadic or periodic tasks, where each task represents a distinct, indivisible feature of the end-user application. The period between consecutive activations of a task must be defined, allowing YASMIN to manage the timing accurately.

However, YASMIN also accommodates non-periodic tasks, giving end-users the flexibility to manage tasks that do not link to a regular activation pattern. This is particularly useful in applications where task execution may be triggered by irregular external events or conditions.

YASMIN supports all three commonly used deadline schemes, providing further flexibility in real-time application design:

- Implicit Deadlines
- Constrained Deadlines
- Arbitrary Deadlines

To address the complexities of heterogeneous platforms, YASMIN allows tasks to be represented as a set of versions, all functionally equivalent but with distinct non-functional behaviors such as different Worst-Case Execution Times (WCET) and energy consumption profiles. This multi-version task approach is crucial in environments where it is not possible to determine in advance whether a task should run on the CPU or be offloaded to accelerators. By supporting different versions of a task, YASMIN enables dynamic decision-making at runtime, optimizing the use of available processing resources and allowing tasks to execute in parallel across different hardware units.

Moreover, YASMIN supports tasks with precedence constraints through the use of directed acyclic graphs (DAG). In this model, tasks are represented as nodes in a graph, with edges denoting dependencies between tasks. This ensures that tasks are executed in the correct order, respecting their interdependencies. For more complex graph-based task models, such as Synchronous DataFlow (SDF), YASMIN requires a transformation into a DAG-compatible format. This transformation ensures that the task model can be efficiently managed within the YASMIN framework, leveraging its scheduling capabilities while maintaining the integrity of task dependencies.

3.2. Implementation

YASMIN is developed as a modular library that can be compiled independently and then linked to the end-user application, providing a flexible and customizable real-time scheduling solution. Its design emphasizes modularity, allowing developers to easily tailor the library to their specific needs.

One of YASMIN's key features is its support for multiple scheduling policies. Developers can choose from a variety of scheduling algorithms and effortlessly switch between them at compile time. This

is made possible through a configuration header file, which serves as the central point for defining the system's scheduling behavior.

The configuration header file is highly customizable, containing settings such as preprocessor definitions, the chosen scheduling policy, task-to-core mapping strategies, and priority assignment methods. By adjusting these configurations, developers can fine-tune how YASMIN manages tasks, making it adaptable to a wide range of application requirements.

This modular and configurable approach not only simplifies the integration of YASMIN into existing projects but also ensures that the library can be adapted to a wide variety of hardware platforms and application scenarios.

3.2.1. Overview of YASMIN API

To use YASMIN, users must follow a structured initialization and configuration process. Here is a detailed overview of how to set up and manage tasks, channels, and hardware accelerators using YASMIN:

1. **Initialization**: Before any other operations, users must call the ‘init’ function. This function initializes all necessary internal structures and prepares YASMIN for task management and scheduling. Proper initialization is crucial for ensuring that the middleware operates correctly and efficiently.

2. **Task Declaration**: Users declare their tasks using the ‘task_decl’ macro. This macro allows for the definition of the various tasks that the application will execute. Each task can be associated with one or more versions using the ‘version_decl’ macro. This enables the specification of different versions of a task, each with unique performance characteristics such as execution time and energy consumption.

3. **FIFO Channels**: To facilitate communication between tasks, YASMIN provides a mechanism for declaring and managing FIFO channels using the ‘channel_decl’ macro. Users connect tasks using the ‘channel_connect’ function. Data can be pushed into a channel using the ‘channel_push’ function and

retrieved using the ‘channel_pop’ function. This enables effective data flow and synchronization between tasks.

4. **Hardware Accelerators**: YASMIN supports integration with hardware accelerators through the ‘hwaccel_decl’ macro. This macro allows users to declare hardware accelerators that can be used to offload certain tasks. Tasks can be linked to these accelerators using the ‘hwaccel_use’ function. YASMIN is also able to detect if a targeted accelerator is currently busy. If so, it can evaluate whether using an alternate version of the task that is suited for a different execution unit would be more efficient.

5. **Priority Inheritance Protocol**: YASMIN applies a Priority Inheritance Protocol (PIP) to ensure that if a higher-priority task requires access to hardware resources, the scheduler will reschedule tasks accordingly to avoid priority inversion.

6. **Preemption and Scheduling**: YASMIN supports preemption with on-line scheduling policies. The scheduler continuously monitors tasks and their priorities. When a higher-priority task becomes ready to execute, the scheduler sends a signal to all worker threads executing lower-priority tasks. These threads then check the task queue for higher-priority tasks. If a higher-priority task is found, a context switch is performed to ensure that the higher-priority task is executed in a timely manner.

7. **Starting and Stopping the Scheduler**: Once the application setup is complete, users call the ‘start’ function to initiate the scheduling and execution of tasks. The start function activates the scheduler, which then manages task execution according to the specified policies. To halt the execution, the ‘stop’ function is called, which stops the scheduler.

3.3. Functioning

The fundamental concept behind YASMIN is to dedicate specific cores exclusively to the execution of Real-Time (RT) tasks, minimizing interference from system tasks and ensuring predictable performance.

To achieve this, YASMIN spawns a dedicated thread on each reserved core, referred to as a worker thread or virtual CPU. These worker threads act as containers for executing user-defined real-time tasks, providing a controlled and isolated environment that prioritizes the timely execution of these tasks.

3.3.1. On-Line Scheduling and Task Management

YASMIN rely on an on-line scheduler responsible for several critical functions:

- **Task Activation**: The scheduler activates tasks based on their arrival times, which are typically defined by their periods in the case of periodic tasks.

- **Version Selection**: Given the potential for multiple versions of a task, each optimized for different performance characteristics, the scheduler must decide which version to execute at runtime. This decision can be influenced by various factors, including the current state of the system, the availability of resources, and predefined optimization goals (e.g., minimizing energy consumption).

- **Task Dispatching**: Once a task is activated and the appropriate version is selected, the scheduler dispatches the task to an available worker thread. This dispatching process is critical for maintaining the real-time properties of the system, ensuring that tasks are executed promptly on the designated cores.

3.3.2. Scheduling Modes: Global vs. Partitioned

YASMIN supports two distinct scheduling modes, which determine how tasks are assigned to and managed by the virtual CPUs:

- **Global Mode**: In this mode, all tasks are eligible to be executed on any of the available virtual CPUs. The worker threads in global mode share a common ready queue, where tasks are placed as they become ready for execution. This shared queue allows for greater flexibility in task allocation, as tasks can be dynamically assigned to any available virtual

CPU, potentially balancing the load across multiple cores and improving overall system efficiency.

- **Partitioned Mode**: In partitioned mode, tasks are pre-assigned to specific virtual CPUs, and each worker thread has its own dedicated ready queue. This approach offers predictability, as tasks are bound to particular cores, reducing the complexity of scheduling and ensuring that tasks are executed on their designated virtual CPUs without contention from other cores.

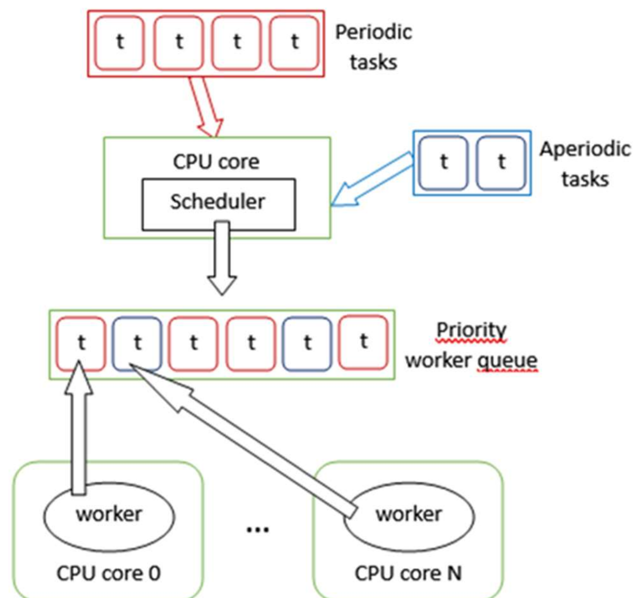

Fig. 12. Global online scheduling strategy

On Figure 12 we can see the global online scheduling where the ready task queue is shared among workers.

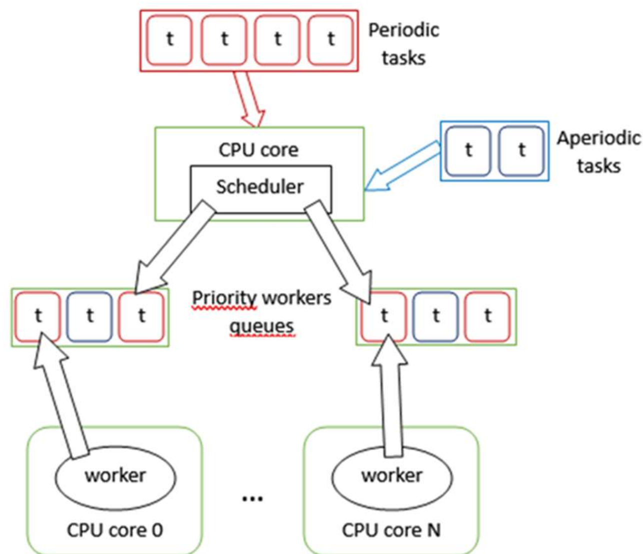

Fig. 13. partitioned online scheduling strategy.

On Figure 13 we can see the partitioned online scheduling where the ready task queue is splitted among workers.

The selection between global and partitioned scheduling modes, along with other configuration options, is made at compile time through the configuration header file. This file allows developers to customize the behavior of YASMIN to suit the specific needs of their application, including the choice of scheduling policy, task mapping strategies, and other operational parameters.

3.3.3. Ready Queue Management

Regardless of the chosen scheduling mode, the management of ready queues is a crucial aspect of YASMIN's operation. The ready queues, whether shared or individual, are populated by a separate scheduler thread. This thread operates in cycles, filling the ready queue(s) with tasks that are due for execution based on their scheduled activation times.

The scheduler thread waits between two actions for a period calculated as the greatest common divisor (GCD) of the task periods. This GCD-based period dictates how frequently the scheduler checks for and enqueues tasks, ensuring that the ready queues are populated

efficiently and that tasks are available for execution at the correct times.

3.4. Cecile

CECILE [8] is a compiler-like toolchain designed to facilitate the development of applications that prioritize non-functional properties such as time, energy, and security. It leverages a domain-specific functional coordination language called TeamPlay. Using TeamPlay, developers can describe the structure of an application, and CECILE then generates code that supports the deployment of parallel applications on actual platforms. Depending on the input parameters, the generated code can follow either a static or dynamic schedule. This flexibility allows users to utilize a range of analysis and scheduling techniques as if they were selecting tools from a toolbox.

The core idea behind CECILE is to streamline the design and deployment of applications with strict timing requirements. To achieve this, CECILE includes its own Domain-Specific Language (DSL), enabling concise, efficient, and versatile application structure descriptions. As a compiler infrastructure, CECILE starts with a high-level representation of an application and proceeds to perform various timing analyses, compute static schedules, or conduct schedulability analysis before generating the necessary glue code. This glue code handles tasks such as initialization, communication, synchronization, and more.

The primary goal of this toolchain is to generate C code that manages the runtime behavior of a set of components described by a high-level coordination language.

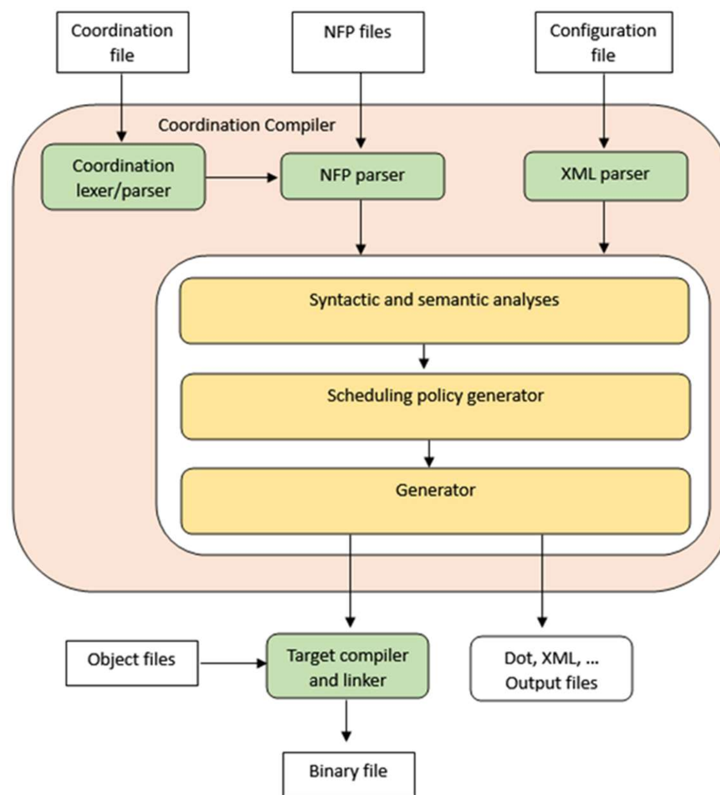

Fig. 14. Coordination Compiler workflow

On Figure 14 we can see the workflow of the coordination compiler. The workflow of CECILE relies on four main inputs:

1. **Coordination File**: This file describes the application's structure, detailing components and their dependencies, as well as specifying non-functional properties.

2. **Configuration File**: This file provides practical configurations, including target hardware descriptions, security level specifications for the mission, and compiler passes to be applied.

3. **(Optional) Non-Functional Properties File(s)**: These files contain timing and energy information provided by external analysis tools.

4. **Object Files**: These files contain the compiled C-code for all components and their respective versions.

Syntactic and Semantic Analyses

The first stage of the toolchain involves Syntactic and Semantic Analysis. During this stage, the tool parses the coordination and configuration files to generate an initial Intermediate Representation (IR) of the system model.

Scheduling Policy Generation

The second stage of the toolchain is Scheduling Policy Generation. Here, the tool uses Execution Time and Security (ETS) information from the Non-Functional Properties (NFP) file(s) to create a second IR, representing the mapping and scheduling decisions made at this step. The configuration file guides the scheduling policy generator in selecting the appropriate scheduling policy. This stage may either construct a schedule table for offline scheduling or perform a schedulability analysis to determine whether the component set can be scheduled on the given platform.

Generate and Export

The final stage of the toolchain is Code Generation and Export. In this stage, the tool generates the C-code for the specified hardware and Operating System (OS) as indicated in the configuration file. Various exporters are available to output reports on different elements, such as the application graph in Dot format or scheduling information in an XML file.

Chapter 4

Project

This project originates from a challenge proposed by the Euromicro Conference on Real-Time Systems (ECRTS)[3]. The challenge involves the study and development of an Augmented Reality Heads-Up Display (AR HUD) system for the automotive industry.

AR HUD systems extend the driver’s external view by overlaying virtual information (augmentations) onto the traffic conditions in front of the vehicle. These systems aim to enhance drivers' situational awareness by displaying graphics that interact with the driver’s field of view (FOV). The information displayed is generated in real-time from various sensors and typically includes advanced driver assistance system (ADAS) alerts, navigation clues, and other visual augmentations overlaid on real-world objects.

AR HUDs represent a high-performance, real-time application, where the interaction between software tasks and the allocation of shared hardware resources must be carefully coordinated to meet both functional and performance requirements.

A key requirement for AR HUD systems is the ability to project images with sufficient positional accuracy, creating the illusion that they are seamlessly "fused" with the real-world environment. Additionally, to adjust the image to the driver’s viewpoint, an eye-tracking or head-pose estimation function is typically employed. This feature determines the necessary rotation or distortion to apply to the image frame to ensure a proper alignment with the driver’s perspective.

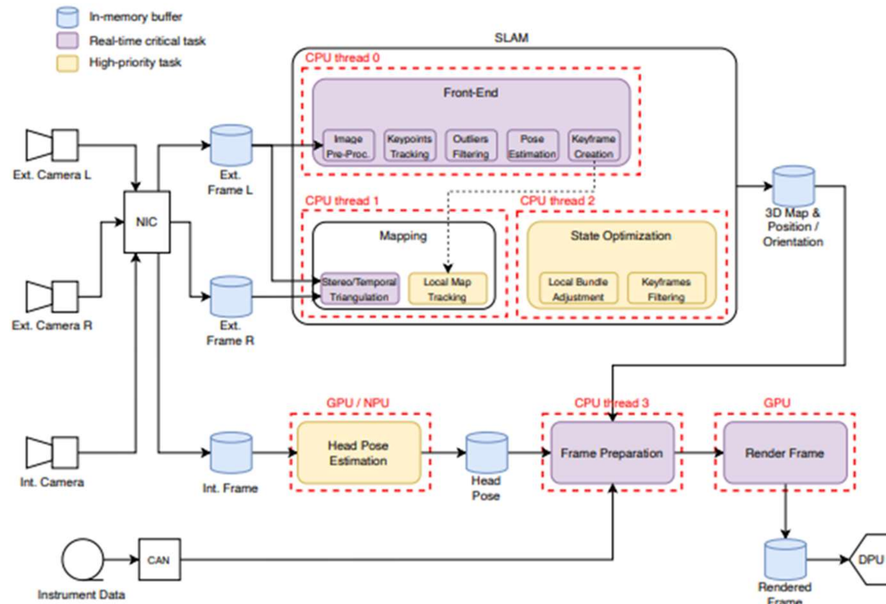

Fig. 15. AR HUD diagram

On Figure 15, the diagram illustrates a possible configuration of the system, showing the decomposition of functions into software tasks and their mapping onto the system's SoC compute resources.

The two primary algorithms depicted are ORB-SLAM (Simultaneous Localization and Mapping) and Head Pose Estimation, both of which are critical to the functioning of the AR HUD.

4.1. Hardware platform

The board selected for this experiment is the NVIDIA Jetson AGX Orin [15].

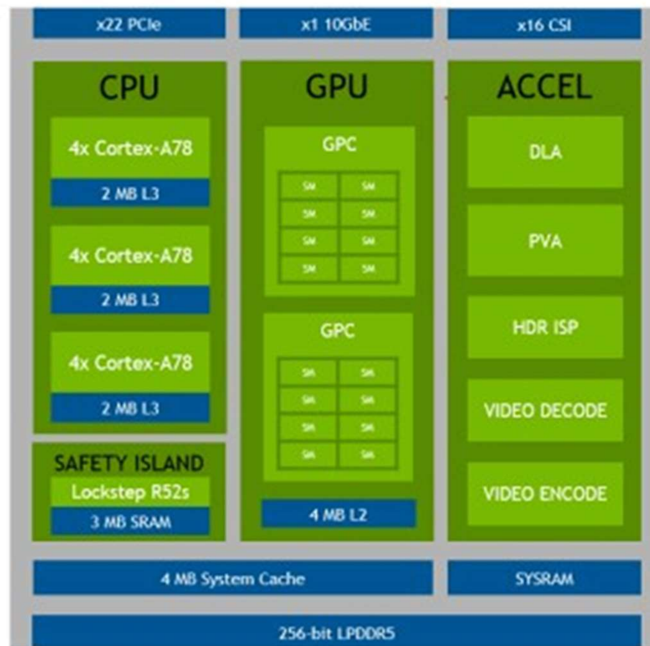

Fig. 16. Orin System-on-Chip (SoC) Block Diagram

On Figure 16, we can see a diagram that shows the Orin board, without too much details, the principal parts are as follows.

Jetson AGX Orin modules feature the NVIDIA Orin SoC with a NVIDIA Ampere architecture GPU, ARM Cortex-A78AE CPU, next-generation deep learning and vision accelerators, a video encoder and a video decoder. High speed IO, 204 GB/s of memory bandwidth, and 64GB of DRAM enable these modules to feed multiple concurrent AI application pipelines.

The SoC is composed of 12-cores of ARM Cortex-A78AE Core processors organized as multiple quad-core clusters.

The SoC cache is composed of 4 layers: an L1 and L2 cache that are private to cores, a L3 cache that is shared among CPUs from the same cluster and, a system cache that is shared between the cores of the SoC.

4.2. ORB-SLAM

A critical component of the AR HUD system is Visual Simultaneous Localization and Mapping (SLAM). SLAM is essential for determining the vehicle's orientation and trajectory, while also

building a map of the surrounding environment. This data is subsequently utilized to generate HUD graphics, ensuring that they are accurately positioned within the driver's field of view (FOV).

For this case study, the ORB-SLAM3 [9] implementation was selected. ORB-SLAM3 is a Simultaneous Localization And Mapping (SLAM) application which allows an autonomous system to navigate in an unknown environment using computer vision.

ORB-SLAM3 is a high-performance, feature-based SLAM algorithm, known for its capability to support both monocular and stereo camera setups, making it suitable for AR HUD applications.

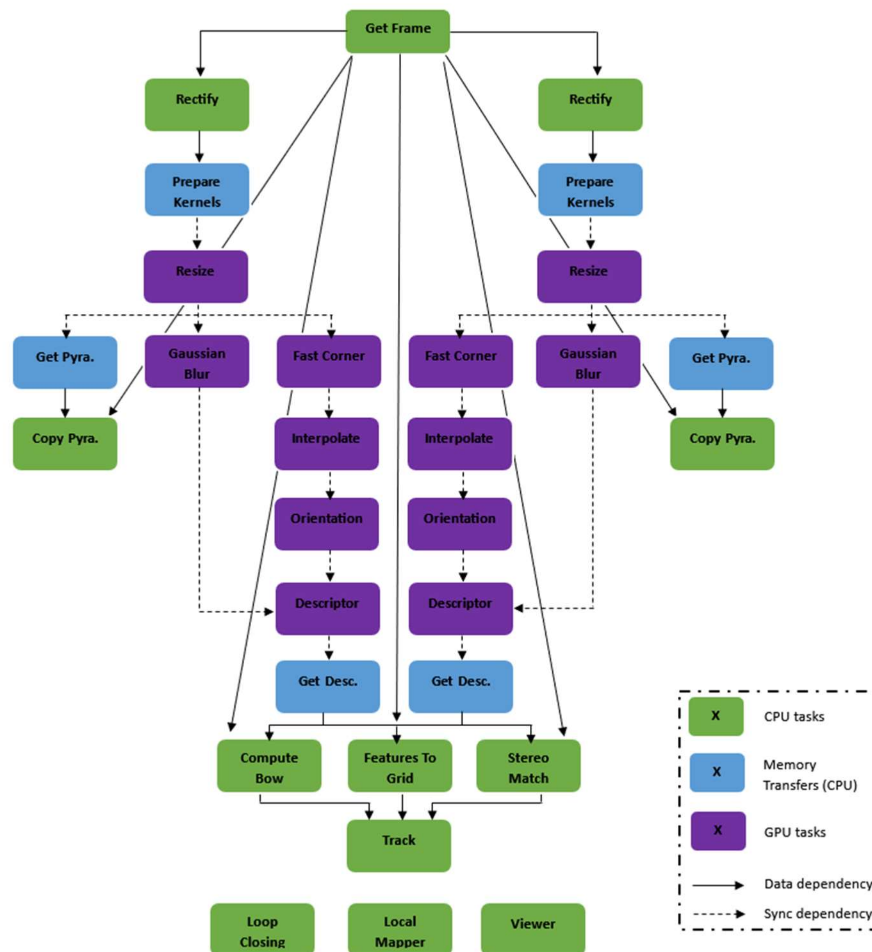

Fig. 17. Task graph of ORB-SLAM3.

As we can see in Figure 17, the process begins by fetching a frame that consists of two images, each image is passed to a separate, but

parallel, task graph, where the images take identical processing steps, mimicking a left and a right eye.

The images first go in the “rectify” function, where it is rectified to correct any distortions, ensuring alignment for stereo vision. Then is passed to the “prepare kernels” function, where the images are forwarded to the GPU for resizing.

The resize operation generates image pyramids, a multi-scale representation of the image that helps in detecting features at different levels of detail. Each level of the pyramid undergoes Gaussian blurring to reduce noise, followed by a Fast Corner Detection algorithm that identifies key corner points (corner of any object) in the images.

For each detected corner, the algorithm computes the orientation and generates a descriptor, a unique representation that allows matching of similar points between frames. Once the descriptors are computed, they are transferred back to the CPU, and key points are extracted and processed in order to track the localization of the system.

The tracking system evaluates whether the current frame should be classified as a key frame (frame that presents key corner points). If a key frame is detected, it is stored in an internal buffer for further processing by three independent tasks: Loop Closing, Local Mapper, and Viewer. These tasks operate periodically, checking whether a new keyframe is available in their buffers.

4.3. Head Pose Estimation

Before being rendered on the AR HUD, images usually require some form of corrections to accommodate for the real-time position of the driver’s viewpoint. Such corrections are usually applied by relying on the output of a head pose estimation function, determining the driver’s viewpoint and making adjustments to the displayed information accordingly.

For this case study, the head pose estimation model selected is Hopenet[11], a convolutional neural network (CNN)-based approach. It operates with an RGB monocular camera, often mounted inside the vehicle, directed at the driver’s face. This camera provides input

frames at a rate similar to external cameras, ensuring synchronization between the internal and external views.

In terms of real-time requirements, the head pose estimation is a high-priority task however it is not a hard real-time task, meaning that minor frame drops are acceptable without a significant impact on the AR HUD's functionality.

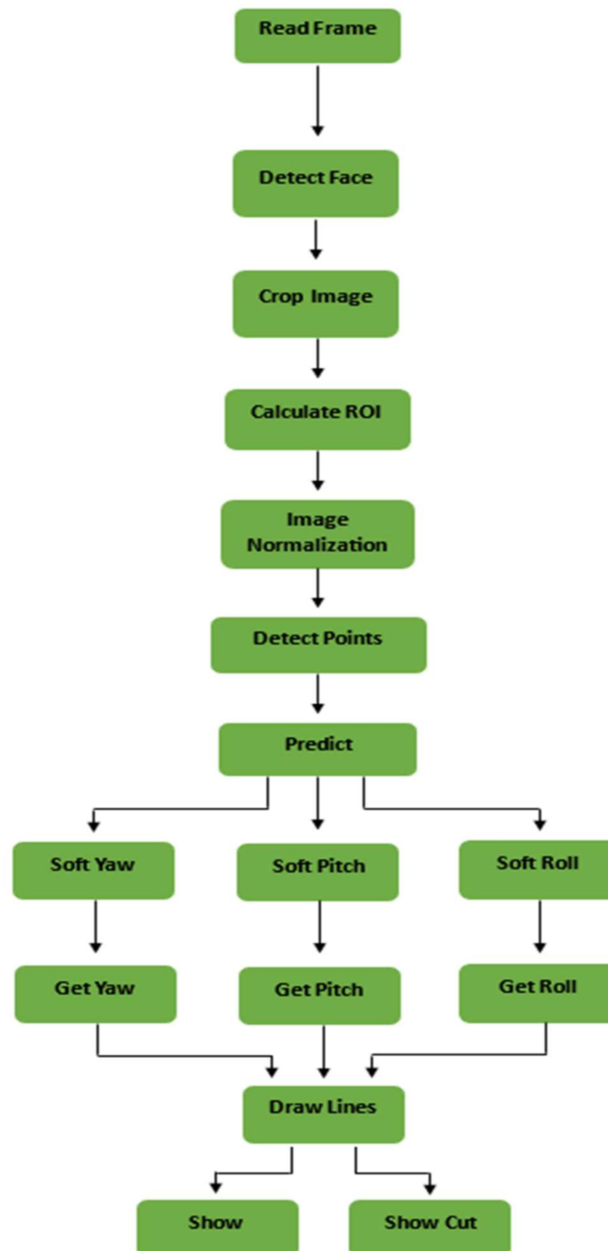

Fig. 18. Task graph of Hopenet.

As we can see in Figure 18, every task is in green, representing that Hopenet task's are CPU tasks.

Hopenet operates through several sequential steps to estimate the yaw, pitch, and roll of the driver's head, which represent horizontal, vertical, and rotational movements. These movements are then used to adjust the visualizations on the HUD.

The system captures an image from the inside camera, which becomes the input for head pose estimation. The next step involves detecting the face within the frame using a face detection algorithm.

Once the face is detected, the image is cropped to the region containing the face. The cropped image is further adjusted by calculating a well-defined Region Of Interest (ROI). This ensures that the face region is centered and potentially shaped into a square around the face.

Then the frame is passed to the Image Normalization function that prepares the image for the model inference.

The model attempts to identify facial landmarks or key points on the face (such as the eyes, nose, and mouth).

The detected points are passed through the Hopenet CNN model, which predicts the yaw, pitch, and roll of the driver's head. These predictions represent the angles that describe how the driver's head is oriented in 3D space.

After prediction, softmax functions are applied to the yaw, pitch, and roll outputs (called Soft Yaw, Soft Pitch, and Soft Roll). The softmax function converts the raw outputs into a probability distribution.

Then the values are converted in angles with the other three functions Get Yaw, Get Pitch and Get Roll.

The final task involves adding visual indicators (lines) to the image that represent where the driver is looking and outputs the full image and a more focused version, cropped on the face region.

4.4. Implementation and Integration

In this chapter we will explain all the steps taken to implement the two algorithms used, fuse together and integrate them into the Yasmin middleware.
Starting from the really first steps, understanding the algorithms, all the dependencies and also the middleware and all the criticality that could be related to these steps.
Getting hand on the code and providing a task graph to show all the tasks and the flow of all the tasks.
Evaluating these graphs and changing the code to provide a better understanding and a better flow of the algorithms.
We will go further to see also different kinds of schedulers and for each give an explanation through a graph that shows the stability reached.
All these tests have been done to have more understanding on the stability and to see which of the provided schedulers can achieve the best stability.
We will also see the schedulability analysis done for our taskset and see if it reaches a feasible schedule.

4.5. Initialization steps

The first task was to study and understand the two algorithms, Hopenet and ORB-SLAM3, that would be central to our project. This initial step relies heavily on analyzing the source code and available literature.

Once we had a solid grasp of both algorithms, the next step was to prepare the board environment to run them efficiently. This involved identifying and installing all necessary dependencies. However, due to the specific architecture of the board, we encountered several issues during the setup process, particularly when dealing with architecture-specific libraries. After overcoming these dependency challenges, the system was finally configured and ready to execute the algorithms.

To gain better control over the input data and ensure consistent testing across multiple schedulers, we decided to modify the algorithms to read frames from a folder instead of capturing them directly from a live camera feed. The variability inherent in real-time camera capture

can lead to inconsistent results, making it difficult to perform controlled tests across different schedulers.

This modification involved writing an algorithm to extract video frames from the camera and save them as a frame set. This ensured that our dataset was fixed, containing a balanced selection of frames, some with faces and others without. By doing so, we ensured that every scheduling approach we tested would have the exact same input, improving the reliability of our performance evaluations.

4.6. Task and period identification

The task graphs presented above were not given for free. This first step of the project was to actually identify the tasks from the code and create these task graphs.

By carefully reviewing the flow of both algorithms, we were able to restructure the code, resulting in better task flow.
This restructuring also facilitated the generation of the task graphs that we mentioned earlier, representing the individual steps and data dependencies for both Hopenet and ORB-SLAM3.
For each iteration of ORB-SLAM and Hopenet, only one iteration of each of their tasks is required to produce the results. Then, each Hopenet task share the same period and this is also true for ORB-SLAM tasks.

As mentioned in the challenge paper [3], Hopenet and ORB-SLAM should work with the frames generated by a camera recorded at 30 FPS (a frame is generated each 33ms).
As we want Hopenet and ORB-SLAM to handle each frame generated, we selected a period for both of their tasks, the time between two generations of frames. In the case of a 30FPS camera, this time is 33ms. As we want to finish working with one frame before starting another, we implicitly set the deadline to the period.

4.7. Yasmin and Cecile management

As a keychain to our project we started focusing on a middleware that could help us to check and measure the response times for a variety of schedulers.

The selected one is Yasmin, this middleware can provide us with a reliable solution having different schedulers and different configurations that can be chosen easily.

A key part of this project involved studying and understanding the Yasmin and Cecile scheduling algorithms. These two applications are related, and both play important roles in managing task execution. Our decision to utilize both was based on the understanding that Cecile generates code for the Yasmin runtime system by providing a first raw structure for Yasmin.

The first step was to develop code for both Hopenet and ORB-SLAM3 that would work with Cecile.

Cecile uses a Domain Specific Language called TeamPlay, so we needed to write the code in a format that Cecile could interpret. This involved creating two key components:

- Coordination File: This file describes the structure of the application and defines the dependencies between tasks. It also includes other important parameters such as deadlines, task periods, and other timing constraints.

- Configuration File: This file provides detailed information about the hardware setup, including the CPU and GPU computing units available, the data types used in the code, and the expected output format.

The coordination file is particularly important, as it requires a comprehensive understanding of the system's data flow and task dependencies. This allows the code to be structured efficiently in Cecile, ensuring that all tasks are executed in the correct order and that all input/output operations are properly defined.

After creating the coordination and configuration files, we were able to generate a rough initial structure of the Yasmin code.

Starting from the code generated, we added to the code the initialization of all the tasks, their synchronization mechanism and we refined the tasks itself.

Furthermore, we can tweak the Yasmin configuration file, to define the properties of the scheduler, such as whether it is preemptive or non-preemptive, the scheduling policy in use, the thread priorities for different tasks, and other parameters.

Once the generated Yasmin code and configuration files were refined, we moved on to the next phase, measuring response time.

4.8. Hopenet alone

To evaluate the performance of Hopenet, we conducted a measurement-based approach to capture a set of response times for each of its tasks. Our goal was to gather sufficient data to gain a clear understanding of the response time distribution for each FPS of Hopenet under different scheduling policies.

We ran the experiment using our prepared dataset of 3,000 frames, with a mix of frames that included faces and others that did not. For each frame rate (FPS) configuration, we repeated the experiment 10 times. This resulted in a dataset of 30,000 execution times for each FPS setting and each scheduler policy tested.

As a baseline performance metric, we measure the response time of Hopenet compiled with GCC and running on a vanilla Linux environment, without any influence of Yasmin. This allowed us to compare the impact of different scheduling strategies later.

We ran these tests across various FPS configurations to capture how the performance scaled at different frame rates. This provided us with an important amount of data for evaluating accurately the potential performance stability provided by each scheduling policy with Yasmin.

4.8.1. WCRT estimation – Baseline metric

After taking all the response times of the baseline metric we computed the WCRT estimation.

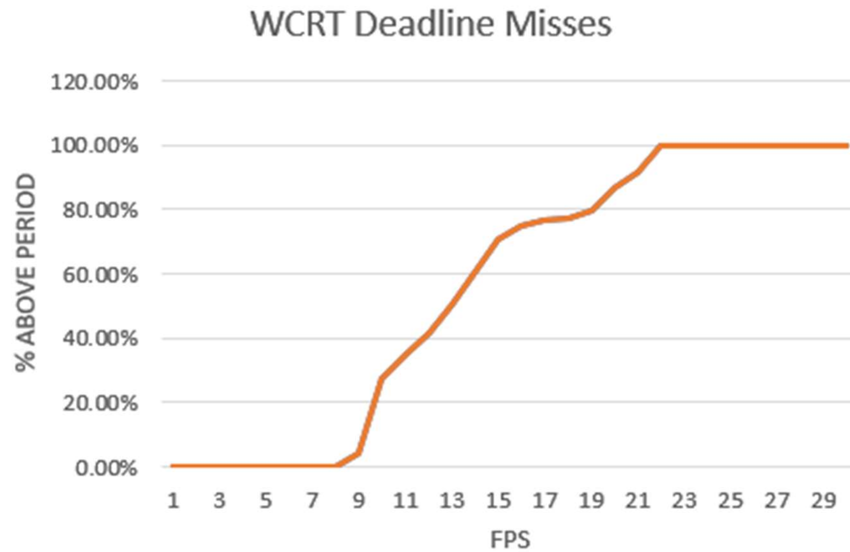

Fig. 19. WCRT Deadline misses for the baseline configuration.

In Figure 19 we can see a graph that shows the WCRT estimations in percentage that are above the related period for the baseline configuration.

As we can see, deadline misses appear starting at the 9 FPS configuration.
From 9 to 15 FPS, the increase of deadline miss rate is relatively constant, reaching a value of 75% at 15 FPS. From this point the speed of increase seems to reduce remaining relatively constant between 15 and 19 FPS then again starting to increase reaching 100% deadline misses on 21/22 FPS.

As we expected a smoother curve we decided to estimate the average response time in order to detect an unwanted behavior of the system. Indeed we expect a constant average of the response time whatever FPS value selected.

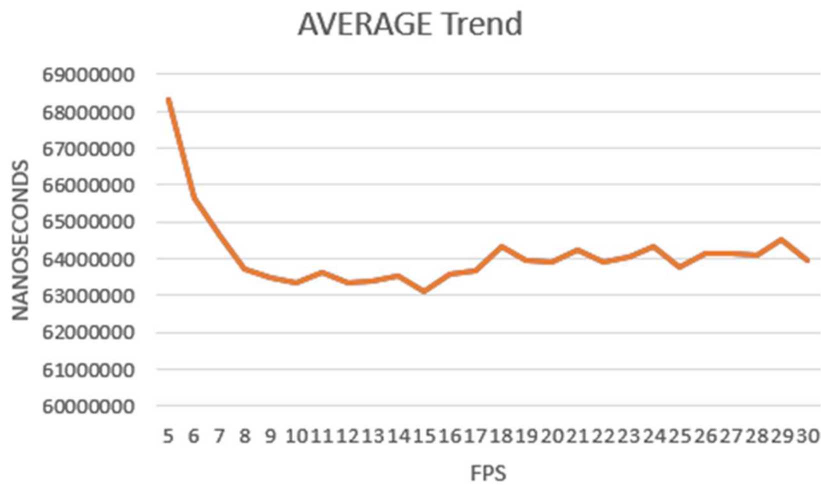

Fig. 20. Average execution times Trend.

Seeing the graph on Figure 20, the graph shows a larger value on the first values of the average and slowly gets smaller till remaining seemingly constant with some bumps.

The first thing we were thinking about was that the board, by settings, doesn't run at full performance, so we created a script that changes those settings and checks if this was executed.
After this change we retried the experiment taking all the measurements, we computed again the deadline misses and drew the curve.

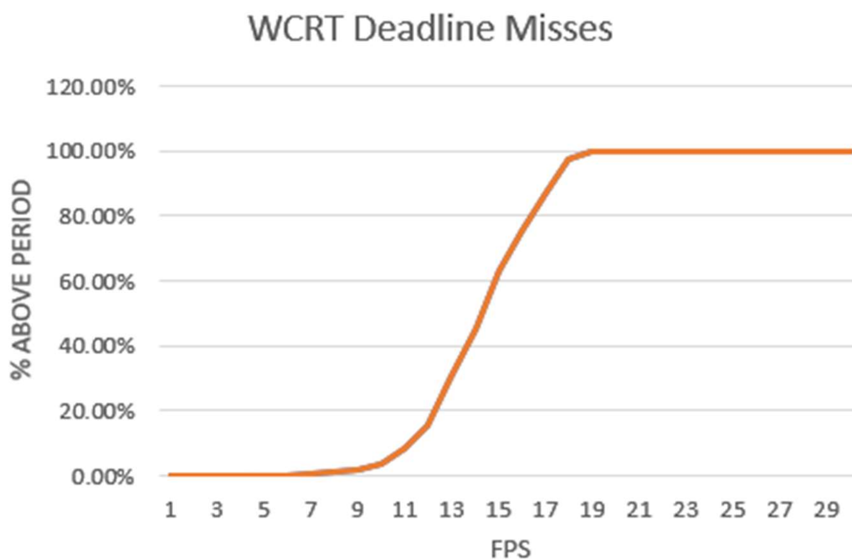

Fig. 21. WCRT Deadline misses graph with full performance.

As we can see in Figure 21, we found out that changing the performance at full, also changed the graph view.
Now as we can see, we found a nicer smooth curve that stays on 12 FPS with a amount of deadline misses below 20%.

Here the curve doesn't present any bump and goes straight to 100% deadline misses.
Also, with the board on maximum performance we checked the average execution times and drew the curve.

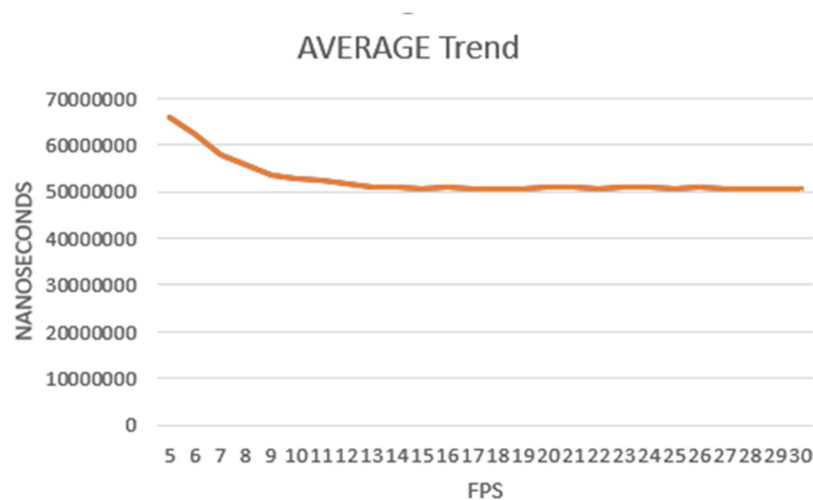

Fig. 22. Average execution times Trend on maximum performance.

As we can see in Figure 22, the average execution times trend took better performance as we expected but with this change, we were not able to solve the issue that caused the first average times to be higher than the rest of the values.

So, for now, also with this issue, trying to understand if the issue originated from the board or from something else, we continued the experiments to see if the scheduling would resolve the problem.

4.8.2. WCRT estimation – with schedulers

To continue the experiment, we began testing different scheduling policies to evaluate their impact on the stability of our algorithms. The schedulers we experimented with were:

- Rate Monotonic (RM)
- Deadline Monotonic (DM)
- First In First Out (FIFO)
- Earliest Deadline First (EDF)

We present here a comparison for the different schedulers.

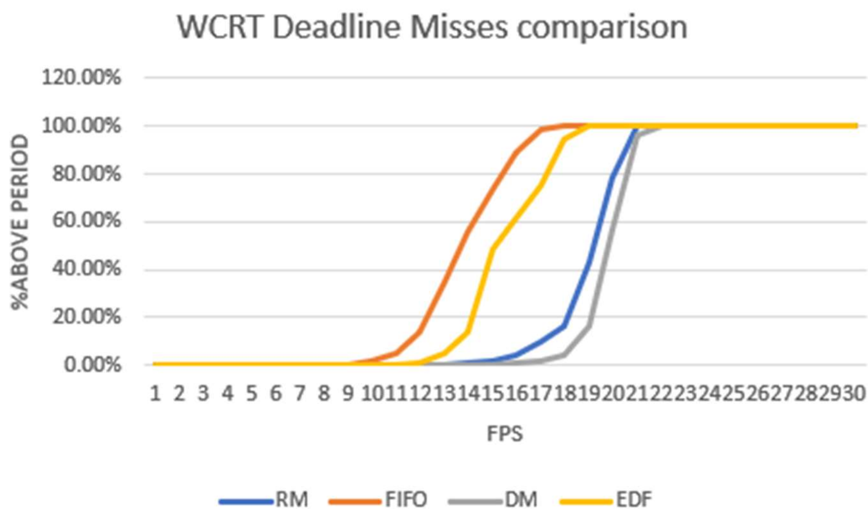

Fig. 23. WCRT Deadline misses graph comparison with schedulers.

In Figure 23, we see a comparison graph between all the schedulers.

Starting from the least stable scheduler, we see in orange the FIFO scheduler with a 11/12 FPS below 20% of deadline misses. In yellow the EDF scheduler with a 13/14 FPS below 20% deadline misses. In blue the RM scheduler with a 17 FPS below 20% deadline misses and at last the gray one is the DM scheduler with a stability of 19 FPS below 20%.

Each scheduler improves the stability taking us from 9 FPS to at least 12 FPS with the worst scheduler in the graph.

After conducting the experiments, we observed that DM provided the best stability among the schedulers. Consequently, we selected DM as the primary scheduling policy for further experiments.

As with the previous tests, we used the same dataset for consistency, which allowed for direct comparisons between results with and without schedulers. During these experiments, we ensured the system operated at the maximum performance settings of the board to minimize variability in the results.

Once the execution times were recorded, we computed the WCET estimation for each FPS setting.

Finally, we plotted the WCET curve to track the relationship between FPS and execution times under this optimized scheduling policy.

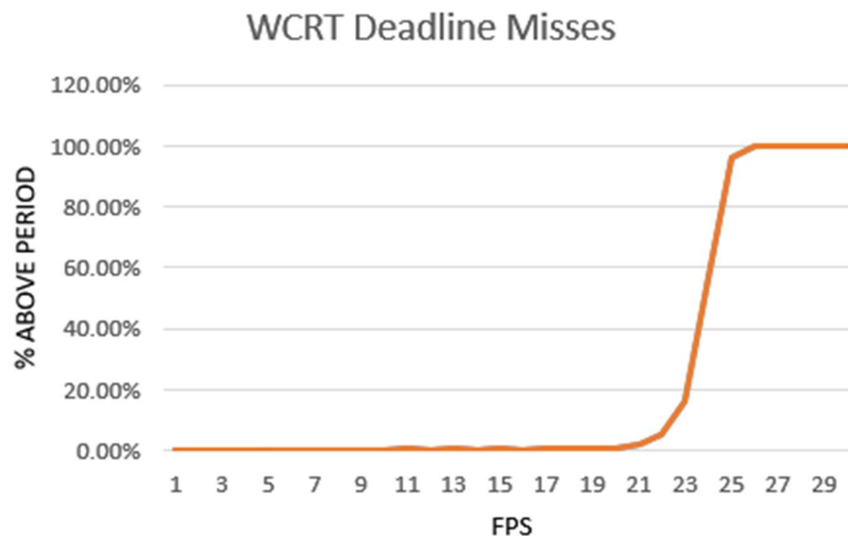

Fig. 24. WCRT Deadline misses graph with DM scheduler.

As we can see in Figure 24, using a DM scheduler improved the stability taking us from 12 FPS without schedulers and from 19 FPS with scheduler without maximum performance to 22/23 FPS with a little deadline miss times with full performance on the board.

We also computed here the average execution times to see if something has changed and draw the curve graph.

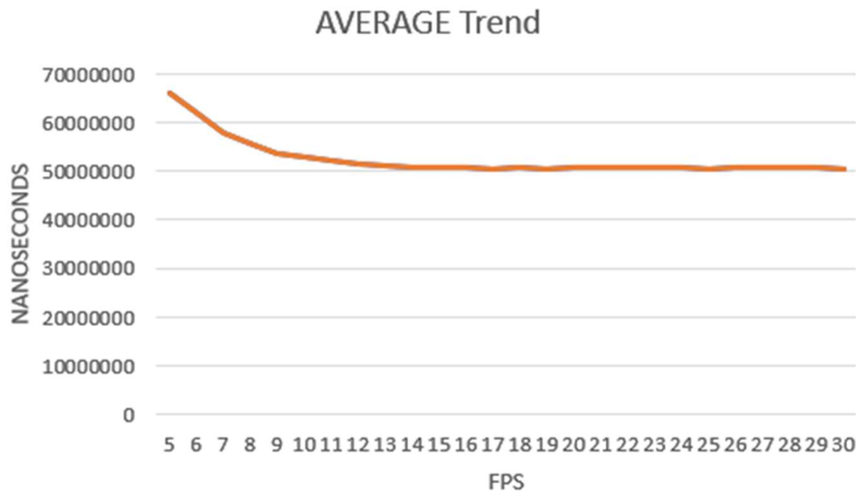

Fig. 25 Average execution times Trend with DM.

As we can see on Figure 25, drawing the curve we stated the same result, we still have a higher value on the initial FPS and then smoothly decrement and flatten around 50 milliseconds.

We started again thinking about the issue and what can be the cause of this problem.

4.8.3. Checking the caching system

We hypothesized that the observed overhead might be related to the caching system of the board. Our assumption was that after running one iteration of the experiment, the caches were filled with cache blocks of the tasks that may be used in future iteration. However, the longer the time interval between two task iterations, the more likely it is that these blocks have been evicted by other tasks on the system. So, with a shorter period, we could see an improvement in WCRT for the same task.

To test this hypothesis, we decided to implement a solution that could better control the cache behavior during the experiments. We began by creating a script designed to flush all the caches before each iteration of the experiment. This would allow us to isolate any potential overhead caused by cache flushing and determine whether it was responsible for the performance drop we were observing.

Next, we modified the Hopenet code to explicitly flush the caches before executing each iteration. The goal was to monitor whether this approach would eliminate or reduce the overhead and provide more stable performance across multiple runs. By systematically flushing the caches before each iteration, we aimed to ensure that the results were not affected by leftover data in the cache from previous iterations, thereby improving the accuracy of our measurements.

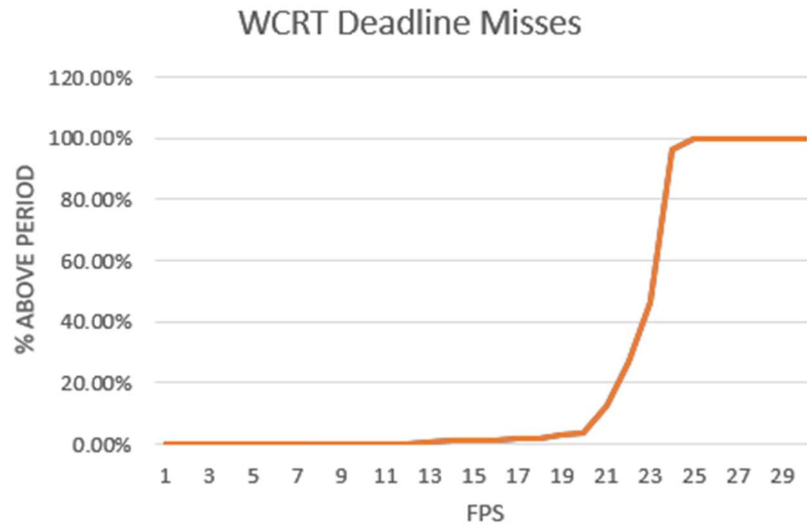

Fig. 26. WCRT Deadline misses graph with DM scheduler and flushing the caches.

As we can see on Figure 26, we now have a little less performance positioning on 20/21 FPS, due to the flushing of the caches.

We want to see now the average trend, to see if the issue is remaining.

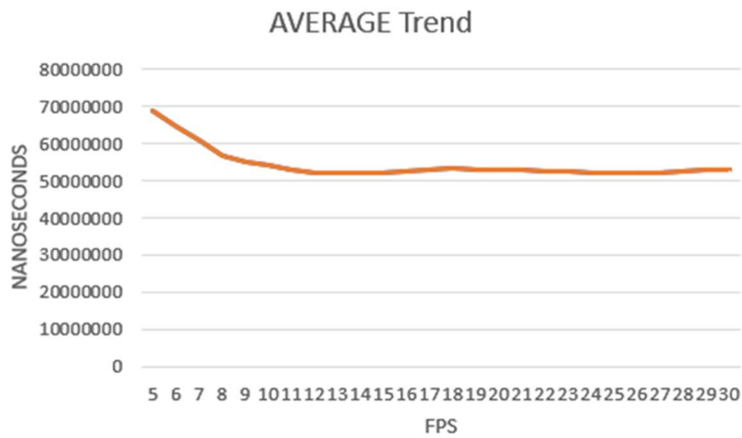

Fig. 27. Average execution times Trend with DM and flushing.

Again, as we can see on Figure 27, doing the average execution times and drawing the curve left us with the same question, we weren't able to understand why the initial FPS showed us an increment and then slowly decrement till flattening.

With this question we continued the project, the issue could be caused by some setting on the board or some other parts which goes beyond the scope of the project.

4.9. ORB-SLAM alone

As we did with Hopenet, we also took a measurement-based approach to gather global execution times for ORB-SLAM3.

This involved collecting execution times at various frames per second (FPS) settings, which allowed us to create a comprehensive dataset of execution times for each FPS level and for every scheduling policy we tested.

Also here, to have a baseline performance metric, we measured the response time of ORB-SLAM3 compiled with GCC and running on a vanilla Linux environment, meaning the algorithm was run without any scheduler or scheduling policy.

This provided us with metrics, which would later serve as a point of comparison between the algorithms and the unification.

After collecting this baseline data, we proceeded to apply various scheduling policies to observe the performance improvements they might bring.

4.9.1. WCRT estimation – Baseline metric

After taking all the execution times for the bare metal code of ORB-SLAM we computed the graph showing the deadline misses.

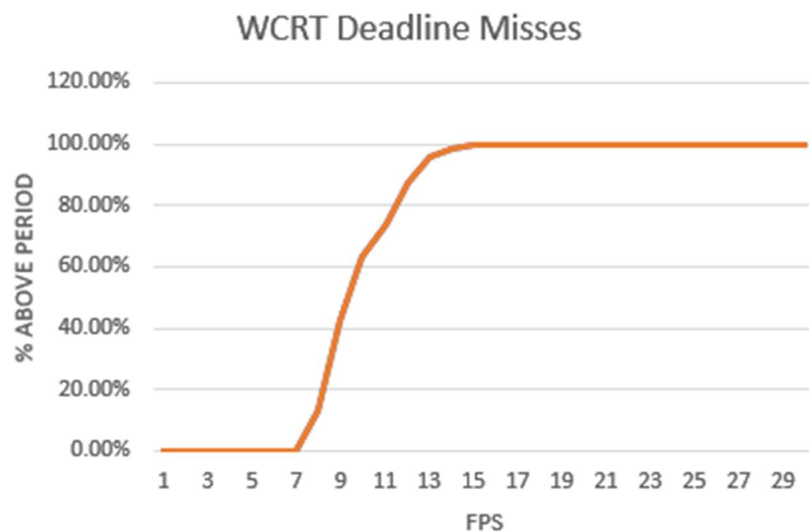

Fig. 28. WCRT Deadline misses graph.

In Figure 28, we can see a graph showing the execution times that are above the period in percentage.
As we can see, in the original code without any scheduling activity, the FPS reached before having too many deadline misses is 7/8 FPS, with a percentage below 20%.
Also here, we computed the average execution time to see the variation on this graph of the execution times taken.

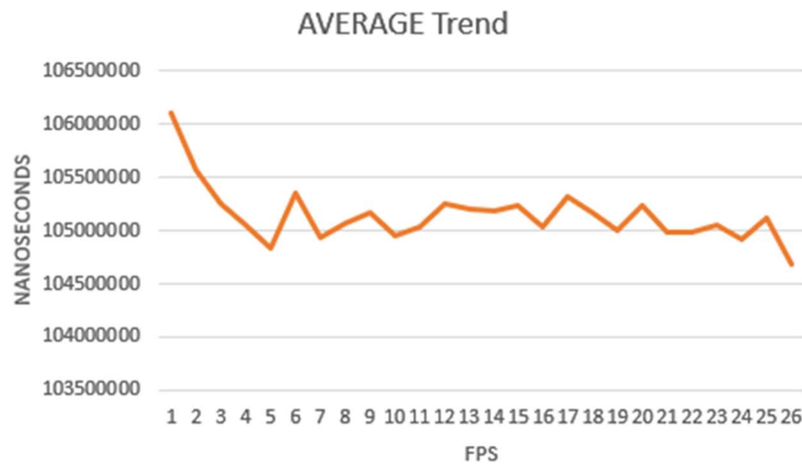

Fig. 29. Average execution times Trend.

As we can see in Figure 29, the variation on the average has some initial high peak and some other little peaks while increasing the FPS, but if we see it from a greater point of view this seems to be quite linear.

In fact, the variation here is from 105 to 106 milliseconds, so it is a little variation.

4.9.2. WCRT estimation – with schedulers

Also here, we started taking the execution times for ORB-SLAM with the schedulers, RM, EDF, FIFO and DM. Here we can see the comparison graph.

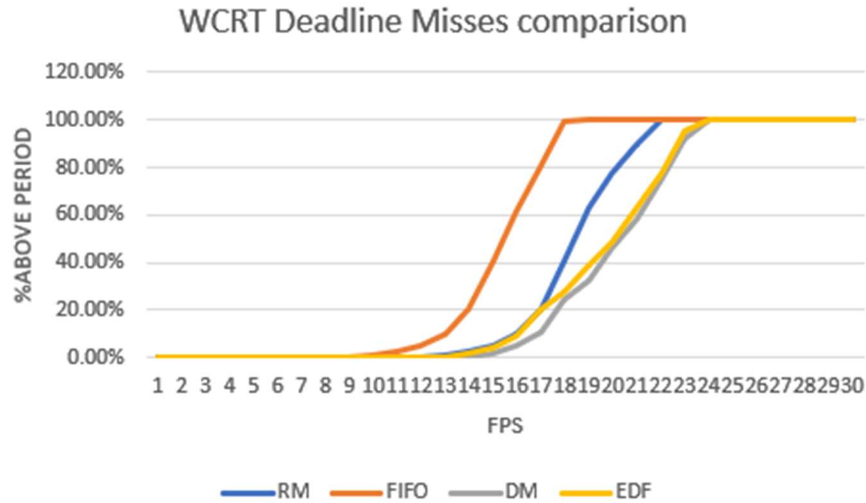

Fig. 30. WCRT Deadline misses graph comparison with schedulers.

In Figure 30, we see a comparison graph between all the schedulers for the ORB-SLAM algorithm.

Starting from the least stable scheduler, we see in orange the FIFO scheduler with a 13 FPS below 20% of deadline misses. In blue the RM and in yellow the EDF schedulers both with a 16/17 FPS below 20% deadline misses. At last the gray one that is the DM scheduler with a stability of 17/18 FPS below 20%.

As we see from the graph the EDF and the DM have a pretty similar stability but the best seems to be the DM one.
Each scheduler improves the stability taking us from 7/8 FPS to at least 13 FPS with FIFO, the worst scheduler in the graph.

After conducting the experiments, we observed that DM provided the best stability among the schedulers. Consequently, we selected DM as the primary scheduling policy for further experiments.
To have a complete overview we give the graph of the selected scheduler, DM and again we go further also with the average trend graph.

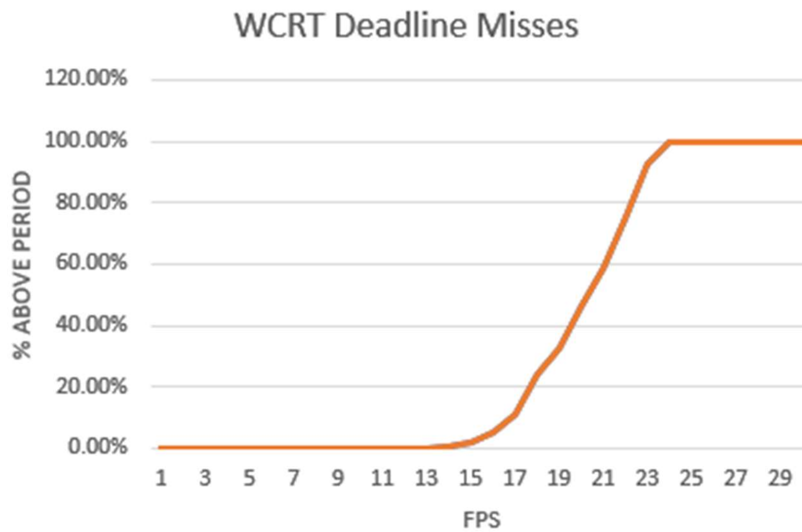

Fig. 31. WCRT Deadline misses graph with DM scheduler.

As we can see on Figure 31, using the DM scheduler improved the stability a lot, taking us from the 7/8 FPS of the computation without scheduler to a 17/18 FPS with the DM scheduler. We improved 10 FPS with the right scheduler.

We also computed the average trend to have a complete overview also for ORB-SLAM.

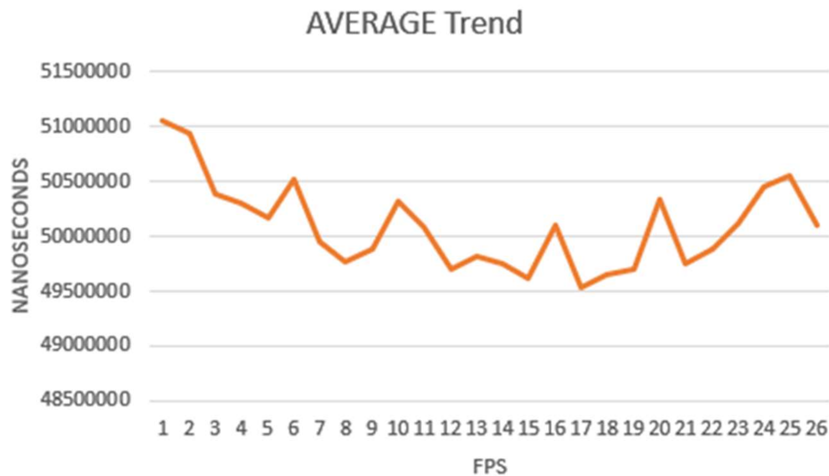

Fig. 32. Average execution times Trend with DM.

As we can see on Figure 32, the average execution time improved a lot, taking us from 105 to 106 milliseconds to 49,5 to 51 milliseconds for the average execution times.

This graph shows a little more peak here and there, but always a limited variation on the execution time.

For completeness of the tests, we also computed the experiment with the flushing algorithm.

4.9.3. WCRT estimation – with scheduler and flushing

Computing the same tests with the flushing algorithm stated us with this graph.

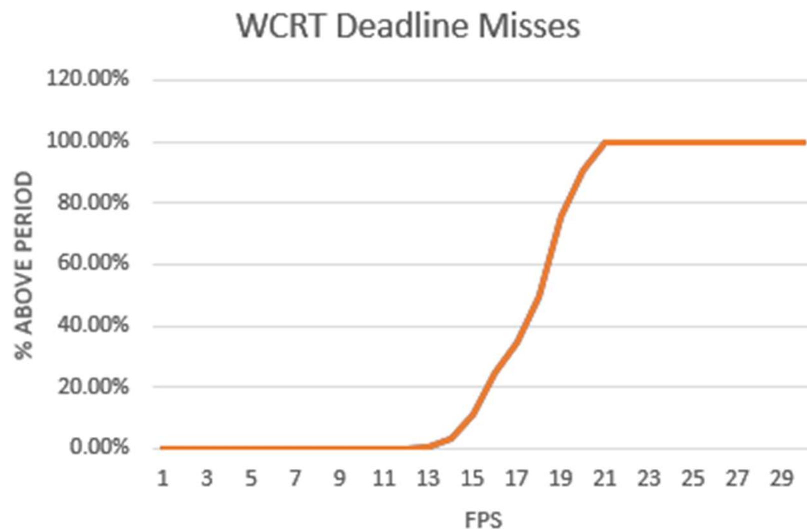

Fig. 33. WCRT Deadline misses graph with DM and flushing.

As we can see on Figure 33, with the flushing algorithm, clearing all the caches, we found a little less stability, as we could imagine.

This probably because we don’t know more information on the caches that needs to be repopulated and so giving us this little overhead.

This is about passing from the 17/18 FPS seen on Figure 31 to the 15 FPS of Figure 33, so a little overhead.

For completion we see also the average trend.

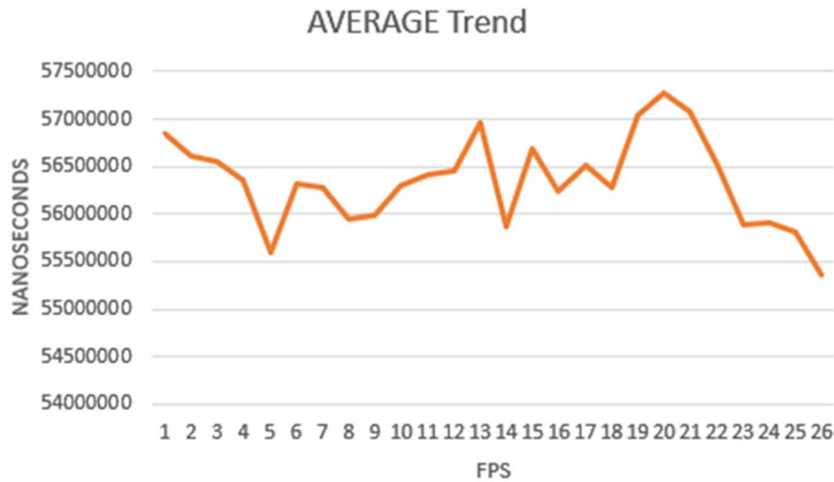

Fig. 34. Average execution times Trend with DM and flushing.

Also in this case, on Figure 34, we see peaks here and there but always with a little variation on the execution times, passing from the lowest 55 to the highest 57 milliseconds for the average times.

Another thing we see from Figure 33 and Figure 34 is the overhead that has been influencing also the average times, passing from the 49,5 to 51 milliseconds of Figure 32 to the 55 to 57 milliseconds of Figure 34.

4.10. Unified application

The next step of the project is to provide a unified version of the code, so a version composed of both Hopenet and ORB-SLAM that work together.

Using YASMIN we again tried all the different schedulers, DM, RM, EDF and FIFO schedulers and computed the graph comparison.

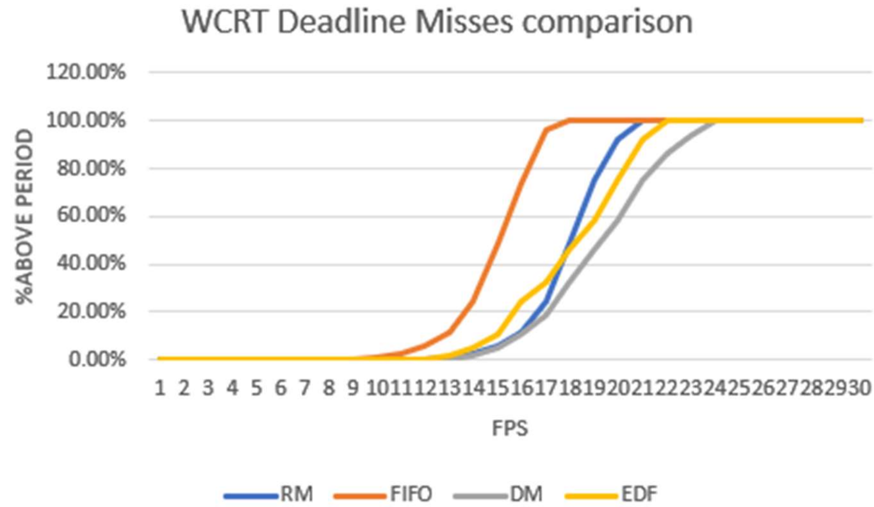

Fig. 35. WCRT Deadline misses graph comparison with schedulers.

In Figure 35, we see a comparison graph between all the schedulers for the ORB-SLAM algorithm.

Starting from the least stable scheduler, we see in orange the FIFO scheduler with a 13 FPS below 20% of deadline misses. In yellow the EDF scheduler with a 15 FPS below 20% deadline misses. At last we see the blue and the gray curve, that are respectively the RM and the DM schedulers with a stability of 17 FPS below 20%.

As we see from the graph the DM and the RM have a pretty similar stability but the best seems to be the DM one especially at higher FPS having a smoother curve.
Each scheduler performs a little worse than as we have seen on the other graphs, this could be related to the parallelization of the two algorithms.
To have a complete overview we give the graph of the selected scheduler, DM and again we go further also with the average trend graph.

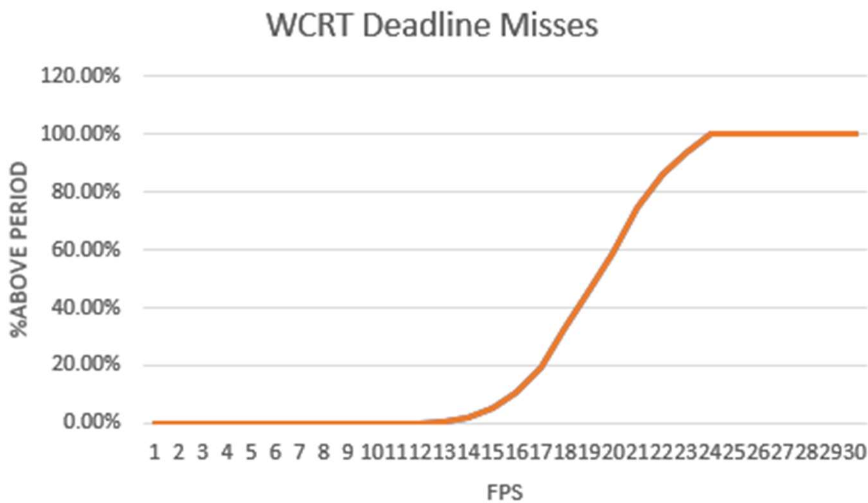

Fig. 36. WCRT Deadline misses graph with DM.

In Figure 36, we see the graph related to the execution times that are above the period compared to the FPS.

As we can see, the curve shows a less than 20% deadline misses on 16/17 FPS then is incrementing, so we can assume we are in this range of activity.

This graph compared to the others is a little less stable, this could be related to the fact that here we are performing the two algorithms together and this gives more weight to the system.

We also computed the average trend so that we can see also here how the values are varying.

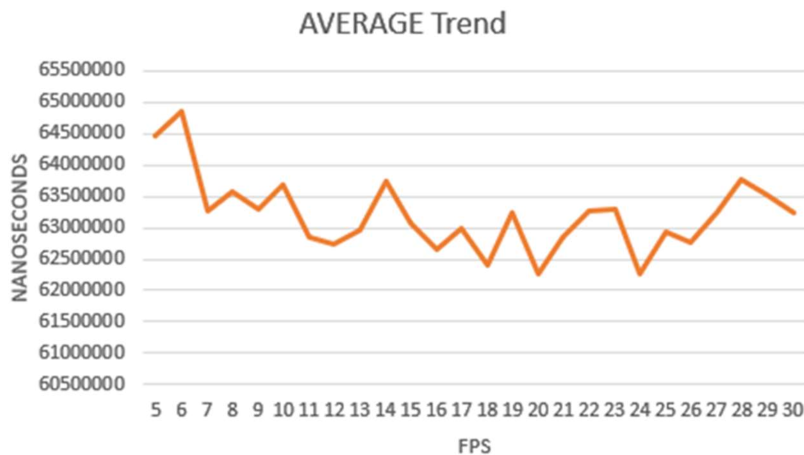

Fig. 37. Average execution times Trend with DM.

As we see on Figure 37, we have some peaks here and there, but the execution is limited in a range from 62,5 to 65 milliseconds.

As we can see there is a major peak at the beginning of 5 and 6 FPS, this can be related to the problem found before.

4.11. Schedulability analysis

Schedulability analysis in a real-time application ensures that all the tasks will meet their deadlines under the scheduling policies.

In this paragraph we will see a tool that we used to perform the schedulability analysis and check if the system will meet the timing constraints.
To have a better understanding on the schedulability analysis we computed this considering firstly 10 FPS as the framerate.

Then we started doing the analysis for every FPS until the taskset wasn't schedulable anymore.
For the schedulability analysis we used the np-schedulability-analysis tool [5], this tool helped us to have an overview of the schedulability for every FPS.

4.11.1. Tool overview

The np-schedulability-analysis tool is a software library designed for schedulability analysis of real-time systems. Specifically, it deals with non-preemptive scheduling meaning that once a task starts executing, it runs to completion without interruption.

The primary purpose of this tool is to analyze whether a set of tasks can meet their deadlines when scheduled under a non-preemptive policy. Non-preemptive scheduling is common in systems where task switching is costly or where task execution must not be interrupted due to the nature of the hardware or system constraints.

The tool tests whether tasks meet their deadlines by considering:

- Task periods

- Task execution times
- Task deadlines
- Blocking times

By analyzing these parameters, the tool checks if each task's response time is less than or equal to its deadline.

And also, as input, the task's precedence constraints, a file specifying the predecessor and successor of each task, if there is one.

The output of the tool indicates whether the task set is schedulable, meaning whether all tasks can complete within their deadlines without being interrupted or delayed by other tasks.

4.11.2. Task model

As a first phase we gather the parameters for all the tasks:

- The execution time: for every task we redone the testing and took the single task execution time in the best and worst cases.
- Period and Deadline: for the period and deadline as we stated before, in this situation the tasks share the same value. In this case we started from a period of 10 FPS and went ahead changing it to reach 30 FPS.
- Release time: each task is released at the same time.
- Priority: the priority of each task depends on the scheduling algorithm, as we stated before we computed the analysis for DM.

For the precedence constraints we can have a look at the task graphs, where we have also reported the predecessors and successors.

4.11.3. File construction

To run the tool, we encoded the information gathered in two separated files:

- The Task file

- The Precedence file

The task file is a CSV file describing each task, each row specifies a task and all the parameters saw before.

The precedence file is a CSV file defining the predecessors and successors, each row specifies this for every task.

4.11.4. Tool run and Results

After finishing the files, we ran the tool to evaluate the taskset. In this section we will see the schedulability ratio graphs of Hopenet, ORB-SLAM and the unified application.

We begin with a period and deadline of 10 FPS and continue until the taskset isn't schedulable anymore.

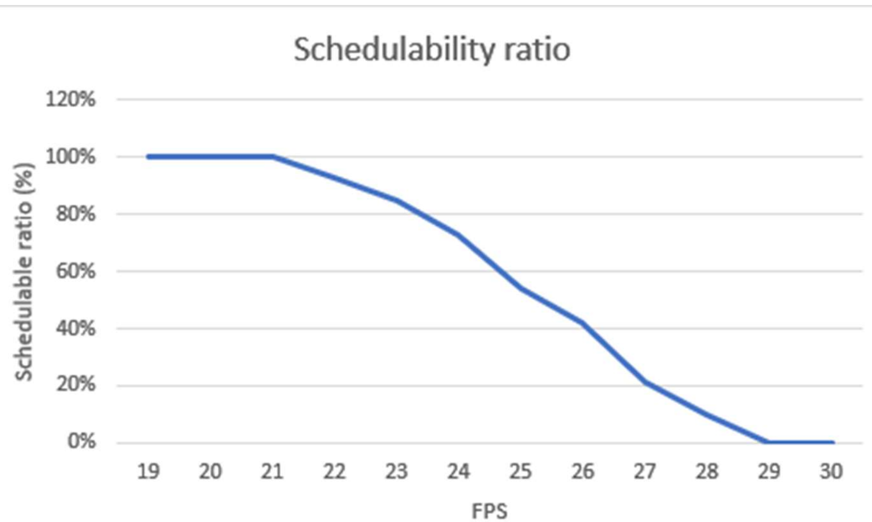

Fig. 38. Hopenet Schedulability Ratio graph.

In Figure 38 we can see the schedulability ratio for Hopenet alone, here trying from 10 FPS to 30 FPS the tool results in a schedulable taskset until 21 FPS then begins to drop reaching the non schedulability on 29 FPS.

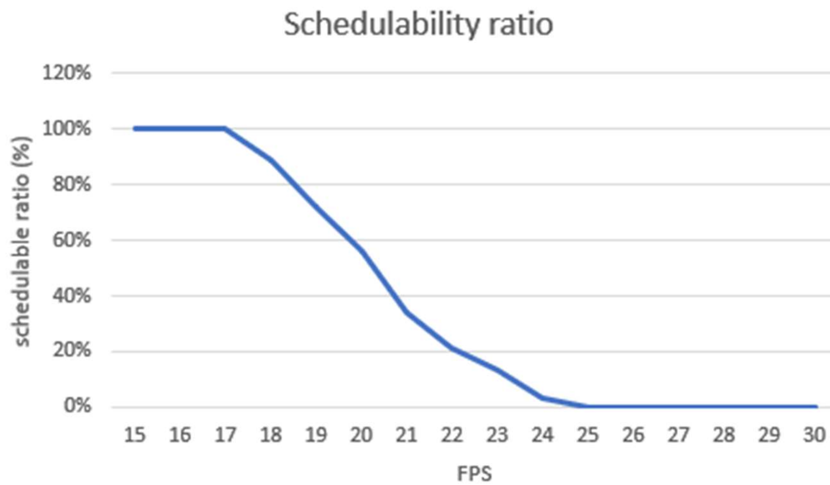

Fig. 39. ORB-SLAM Schedulability Ratio graph.

In Figure 39 we can see the schedulability ratio for ORB-SLAM alone, also in this case trying from 10 FPS results in a schedulable taskset until 17 FPS to a non schedulable taskset at 25 FPS.

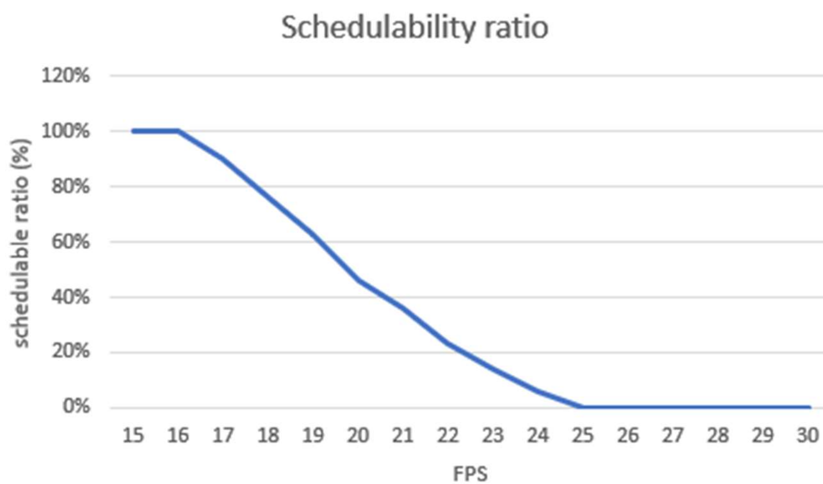

Fig. 40. Unification Schedulability Ratio graph.

The schedulability ratio we can see on Figure 40 shows a schedulable taskset reaching an FPS of 16 and then dropping and becoming non schedulable at 25 FPS.

After the experiment, the analysis confirms that all tasks can meet their deadlines and so the system is schedulable under the period and deadline of 16 FPS.

After this point the tool indicates that the deadline misses start to appear much more frequently reaching the point where the taskset is not schedulable.

Chapter 5

Conclusion

The work done so far points a line on the unification, the analysis of the stability and the schedulability analysis of different algorithms. From the beginning on choosing the right algorithms, have them working separately and test them to have a working separate system.

From the unification of the different algorithms to have a full fused system to work with and that it can be used and analyzed together.
To the goal of this challenge that aims to check the stability and the schedulability of the different algorithms together.

The goal of the challenge has been reached, with as we have seen the stability and the schedulability analysis that shows a schedulable system.
However, the work done shows that the stability reached is of 16/17 FPS with a percentage of deadline misses under 20% that is not so good dealing with a camera and a schedulability analysis that shows a schedulable system under a limit of 16 FPS.

5.1. Future works

This section is intended to provide some ideas and possible improvements to the project for a final and complete larger project.

Improvement on the stability

The stability of the two algorithms can be improved to improve also the reactivity and the performance of the system.

This implies using a more performant board or changing the algorithms and choosing those who perform better separately and also together.

Collision detection

The AR-HUD displays are intended to be used also to have some mechanism of attention signal that highlights an area of interest to be warned of.

A possible improvement of the project can be the introduction of this safety method on the ORB-SLAM algorithm to highlight the points of interest in case of possible accident.

In approaching an object or a person the AR-HUD highlights the area the give more driver attention on this part of the windshield

Tiredness and wake-up mechanism

Also, the Hopenet algorithm can be improved for example by doing some mechanism of tiredness or sleeping awareness and a wake-up safety mechanism.

When dealing with car displays that use a mechanism of angle adjusting in respect to the driver’s view port, a really cool and safer feature could be the tiredness or sleeping checking.

A feature that tries to identify a sleeping driver and then another mechanism of wake up actions that tries to awake the driver.

Bibliography

[1] M. Nasri and B. Brandenburg, “An Exact and Sustainable Analysis of Non-Preemptive Scheduling”, Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS 2017), pp. 12–23, December 2017.

[2] M. Nasri, G. Nelissen, and B. Brandenburg, “Response-Time Analysis of Limited-Preemptive Parallel DAG Tasks under Global Scheduling”, Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), pp. 21:1–21:23, July 2019.

[3] Matteo Andreozzi, Giacomo Gabrielli, Balaji Venu, and Giacomo Travaglini. Industrial Challenge 2022: A High-Performance Real-Time Case Study on Arm. In 34th Euromicro Conference on Real-Time Systems (ECRTS 2022). Leibniz International Proceedings in Informatics (LIPIcs), Volume 231, pp. 1:1-1:15, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2022).

[4] Ranjha, S., Nelissen, G. and Nasri, M., 2022, May. Partial-order reduction for schedule-abstraction-based response-time analyses of non-preemptive tasks. In 2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS) (pp. 121-132). IEEE.

[5] https://github.com/gnelissen/np-schedulability-analysis

[6] Davis, Robert I., and Alan Burns. "A survey of hard real-time scheduling for multiprocessor systems." ACM computing surveys (CSUR) 43.4 (2011): 1-44.

[7] Rouxel, Benjamin, Sebastian Altmeyer, and Clemens Grelck. "YASMIN: a real-time middleware for COTS heterogeneous platforms." Proceedings of the 22nd International Middleware Conference. 2021.

[8] Roeder, Julius, et al. "Towards energy-, time-and security-aware multi-core coordination." International Conference on Coordination Languages and Models. Cham: Springer International Publishing, 2020.

[9] Campos, Carlos, et al. "Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam." IEEE Transactions on Robotics 37.6 (2021): 1874-1890.

[10] Wilhelm, Reinhard, et al. "The worst-case execution-time problem—overview of methods and survey of tools." ACM Transactions on Embedded Computing Systems (TECS) 7.3 (2008): 1-53.

[11] Ruiz, Nataniel and Chong, Eunji and Rehg, James M., Fine-Grained Head Pose Estimation Without Keypoints, "The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops", June, 2018.

[12] Hirvisalo, V. (2014). On static timing analysis of GPU kernels. In 14th International Workshop on Worst-Case Execution Time Analysis (2014). Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[13] Dreyer, B., Hochberger, C., Wegener, S., & Weiss, A. (2015). Precise continuous non-intrusive measurement-based execution time estimation. In 15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015). Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[14] Baruah, S. K., Burns, A., & Davis, R. I. (2011, November). Response-time analysis for mixed criticality systems. In 2011 IEEE 32nd Real-Time Systems Symposium (pp. 34-43). IEEE.

[15] Karumbunathan, L. S. (2022). NVIDIA Jetson AGX Orin Series.

[16] Burns, A., & Wellings, A. J. (2001). Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX. Pearson Education.

[17] Vestal, S. (2007, December). Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In 28th IEEE international real-time systems symposium (RTSS 2007) (pp. 239-243). IEEE.

[18] Liu, C. L., & Layland, J. W. (1973). "Scheduling algorithms for multiprogramming in a hard-real-time environment." Journal of the ACM (JACM), 20(1), 46-61.

[19] Wilhelm, R., Engblom, J., Ermedahl, A., et al. (2008). "The worst-case execution-time problem—overview of methods and survey of tools." ACM Transactions on Embedded Computing Systems (TECS), 7(3), 36.

[20] ISO 26262 (2018). Road vehicles – Functional safety. International Organization for Standardization (ISO).

[21] RTCA DO-178C (2011). Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics.

[22] EN 50128 (2011). Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems. European Committee for Electrotechnical Standardization (CENELEC).