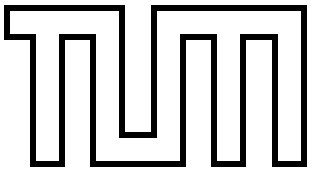


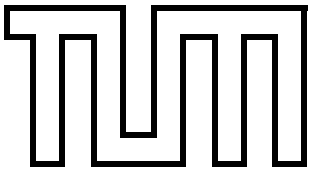
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Institute for Integrated Systems

Mixed-criticality scheduling of an autonomous driving car

Lu Cheng



DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Institute for Integrated Systems

Mixed-criticality scheduling of an autonomous driving car

Author:	Lu Cheng
Supervisor:	Prof. Dr.-Ing. Walter Stechele
Advisors:	Dr. sc. nat. Kai Huang M. sc. Biao Hu
Date:	September 12, 2016

I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, 12. September 2016 Lu Cheng

Acknowledgments

I would like to thank Mr. Biao Hu, Prof. Kai Huang and Prof. Walter Stechele who have given me such an interesting topic as my master thesis and were always ready and willing to answer my questions and give suggestions. I also want to thank the students who participated in the SS 2016 lab course "Hardware/Software Co-Design with a LEGO car". Especially Simon Rummert, Oleksii Moroz, Christoph Wüstner, Yann Fabel, Adnan Makhani and Fang Yuan who always took part and continuously improved their codes. Special thanks goes to Roger Rösch and Florian Hisch. Although Roger has already graduated he still keeps providing help for the project. And Florian Hisch has given me such a lot of suggestions and tips on my works. Without their help my work also the project would be in a far earlier state. Finally I would like to thank Prof. Dr. -Ing. Elmar Schrüfer, who is in charge of my double master program, he is such a nice professor, who cares not only our study but also our lives. With his help I adapted myself into the life in germany such quickly. I appreciate the accompany and encourage from all of my friends, together with them I had a great time in germany!

Abstract

An automotive system is a typical mixed-criticality system, in the sense that tasks with different criticality levels are running on a shared platform. In mixed-criticality system, the timing of safety-critical tasks must be strictly guaranteed. Especially in a multi-core platform, the exists of resource conflict, deadlock make the design and analysis of real-time system even more difficult. There are already several scheduling algorithms, which are proposed to solve multi-core scheduling problem, such as partitioned EDF-VD, Flexible Time Triggered Scheduler (FTTS). But none of the them is designed for precedence-constrained tasks system. And the framework used to implement the partitioned EDF-VD and FTTS is only designed for evaluating the overheads of different scheduling algorithms with simulated busy-waiting tasks. There is no portable framework, which could implement different scheduling algorithms with real-life task set in multi-core platform.

This thesis is based on the work of the mentioned framework called Hierarchical Scheduling Framework (HSF). This framework is easily extensible, such that some new scheduling algorithms can be implemented and evaluated. The research object is an autonomous model car, which uses a quad-core platform "Raspberry Pi 3" with a Linux operating system as a main processor. After studying the task set of the autonomous car, two scheduling algorithms "Time Triggered Scheduling with Mode Change(TTS-MC)" and "Event-scheduler in Multi-Core (Event scheduler-MC)" for precedence-constrained task set are presented and implemented. After all a general method is presented, with which a mixed-criticality precedence-constrained task set could be easily implemented. The results show that with the new scheduling algorithms there is no deadline miss of safety-critical tasks and the scheduling overhead is less than 0.1%.

Except the design of schedulers, the design of the hardware and software structure as well as the functionality development of path tracking of the autonomous car are also big parts of this thesis.

Contents

Acknowledgements **vii**

Abstract **ix**

List of Abbreviations **xiii**

1. Introduction **1**
1.1. Background . . . 1
1.2. Motivation . . . 1
1.3. Related Work . . . 2
1.4. Contributions . . . 4
1.5. Organization . . . 4

2. Hardware and software Design for the autonomous car **5**
2.1. Project Overview . . . 5
2.2. Hardware Design . . . 6
2.3. Software Design . . . 8

3. Task allocation **11**
3.1. Comparison of the global and partitioned scheduling . . . 11
3.2. Notation Declaration . . . 11
3.3. Worst Case Execution Time measurement . . . 13
3.4. Makespan Optimization . . . 13
3.4.1. Graham's List Scheduling . . . 14
3.4.2. Polynomial Time Approximation Scheme . . . 14
3.5. Torsche scheduling toolbox . . . 15

4. Mixed-Criticality scheduling of precedence-constrained task set **19**
4.1. Problem description . . . 19
4.2. Time Triggered Scheduler with Mode Change . . . 20
4.3. Event Scheduler in Multi-Core . . . 25
4.4. Scheduling examples . . . 27

5. Implementation **29**
5.1. Hierarchical Scheduling Framework . . . 29
5.1.1. The Thread Hierarchy . . . 30
5.1.2. Scheduling Mechanism . . . 32
5.2. Implementation of TTS-MC . . . 35
5.2.1. Parser Extension and XML sample . . . 40

5.2.2. Simulation Output and Visualization 42
5.3. Event scheduler-MC 42
5.3.1. Parser Extension and XML sample 44
5.3.2. Simulation Output and Visualization 45
5.4. Real task set support 46
5.5. Time Measurement 46

6. Integration of software modules **49**
6.1. Path tracking 49
6.2. integration of tasks 54

7. Experiments **59**
7.1. Evaluation of TTS-MC 59
7.2. Evaluation of Event scheduler-MC 61

8. Conclusion and problems **65**
8.1. Conclusion 65
8.2. Problems 65

Appendix **69**

A. Installation of Hierarchical Scheduling Framework **69**

B. XML file for experiments **73**
B.1. XML file for TTS-MC experiment 73
B.2. XML file for Event scheduler-MC experiment 76

Bibliography **79**

List of Abbreviations

MCS	Mixed-Criticality System
EDF-VD	Earliest Deadline first with Virtual Deadline
WCET	Worst Case Execution Time
FTTS	Flexible Time Triggered Scheduler
HSF	Hierarchical Scheduling Framework
TTS-MC	Time Triggered Scheduler in Multi-Core
ASIL	Automotive Safety Integrity Level
AUTOSAR	AUTomotive Open System ARchitecture
GPS	Global Position System
LITMUS^{RT}	Linux Testbed for Multiprocessor Scheduling in Real-Time Systems
DLS	Dynamic List Scheduling
IMU	Inertial Measurement Unit
PWM	Pulse Width Modulation
I^2C	Inter-Integrated Circuit
ILP	Integer Linear Programming
POSIX	Portable Operating System Interface
TSC	Time Stamp Counter
RGB	Red Green Blue
OpenCV	Open source Computer Vision
CPU	Central Processing Unit
FIFO	First-In First-Out Queue
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
OS	Operating System
RAM	Random Access Memory

1. Introduction

1.1. Background

To integrate components with different critical levels onto a shared hardware platform is one of the trends in the design of real-time and embedded systems. Such kind of system is called mixed-criticality system (MCS)[9]. Unmanned aerial vehicles(UAVs)[35] and automotives are typical mixed-criticality systems. Take UAV as an example, there are usually 3 different critical levels of tasks: 1) the most critical tasks are "safety-critical" operations, which adjust the flight surfaces to maintain stability. These tasks are not allowed to miss their deadlines (hard real-time). 2) the second critical tasks are "mission-critical" operations, which are in charge of the jobs such as communication, decision making. Several misses of deadlines are tolerant, but tasks should not be delayed for a long time (soft real-time). 3) Tasks in charge of other works are of lowest criticality, such as the optimal route searching algorithm, which may take a lot time (best effort). A mixed-criticality system could have two or more distinct levels according different standards. For example the Automotive Safety Integrity Levels (ASILs)[38] is a classification scheme defined by ISO 26262[29][32], which defines 5 criticality levels of automotive systems: ASIL A, ASIL B, ASIL C, ASIL D and a non-hazardous level QM. There are already some software standards, which are focus on mixed-criticality issues, such as the AUTOSAR[8] (AUTomotive Open System ARchitecture) in automotive industry and ARINC standards in the avionics industry.

Another trend in development of real-time and embedded system is the extension of using multi-core platform[15]. This is due to the demand on higher computational ability, smaller size, less weight, less power consumption as well as lower cost, so called "SWaP +C" concerns. Parallelism is an effective solution to improve computational ability. But problems such as resource conflict, cache contention, deadlock caused by parallelism has made the development of mixed-criticality system more difficult. The biggest challenge is to guarantee the safety assurance of safety-critical tasks. A lot of researches have been done to improve the timing assurance of safety-critical tasks in phases of modeling, verification, design as well as implementation on both of hardware and software design.

1.2. Motivation

This thesis is based on a project from the lab-course "HW/SW co-design with a LEGO car" , which targets to build a low-cost autonomous model car. The car has been equipped with several sensors such as cameras, ultrasonic sensors, GPS module to allow autonomous

driving. The car should be able to drive on a road with clear board towards a target location and follow instructions of road signs and traffic lights. In the designed autonomous car system a quad-cores hard-ware platform Raspberry Pi 3 is used as a main processor. And there will be 10 tasks with different criticality levels running inside, which makes the system a typical mixed-criticality system. The car weighs more than 20 kg, could reach the speed up to 80 km/h. Deadline misses such as the speed control or the steering function could lead to serious damages. Therefore it is essential to study the timing feature of this system and provide a solution to guarantee the real-time of safety-critical tasks.

1.3. Related Work

The first research on mixed-criticality scheduling on multiprocessor platform is presented in 2009 by Anderson et al.[2] and extended in 2010[27]. The main mechanism of their presented architecture (referred as "MC^2") is to execute less critical task in the left time of more critical task, which is called "slack shifting" in order to improve the utilization of computational bandwidth. This mechanism is achieved by assigning high critical tasks with multiple Worst Case Execution Time(WCET). The MC^2 consists of 5 criticality levels: A, B, C, D, E. Different scheduling methods are used in different layers, e.g. level A tasks are statistic assigned and cyclic released, level B with a Partitioned-EDF(Eariest deadline first) scheduler, level-C, D with G-EDF. When scheduling, each task is allocated a budget equal to its WCET value for its own criticality level. For implementation a $LITMUS^{RT}$[10] based scheduler was developed as a testbed for experimentally exploring various implementation-related tradeoffs. With this framework Bommert[7] proposed a method to segment parallelized algorithms in mixed criticality multi-processor system in order to obtain proper load distributions with reduced overhead.

Kritikakou et al.[21] provided a new architecture, in which tasks are distinguished as only two levels: HI-criticality and LO-criticality. Same architecture was proposed by S.Baruah et al.[4]. The algorithm called EDF-VD(Earliest Deadline First with Virtual Deadlines)[40] has also two criticality levels. The concept of virtual deadline is introduced, which is calculated according to a speed up factor and utilization bounds. The schedulability has been theoretically proved. Both of their works found HI-criticality task will be interfered by a LO-criticality task running on a different core due to the use of shared buses and memory controllers. LO-criticality task will be just aborted when some specific event happened, e.g. when a task overruns its virtual deadline. Implementation of Kritikakou et al.'s works were applied on a multi-core platform, the TMS320C6678 chip of Texas Instrument. The scheduler functions such as suspension and the resume of low criticality tasks is implemented using the event and interrupt mechanisms.

Task allocation in multi-core platform is also an important factor, which impacts the performance of a scheduler. The research on task allocation varies from different optimization's objectives. Lakshmanan et al.[20] presented a compress on overload packing scheme to maximize the tolerance of high-criticality tasks to overloads. Optimization routines are presented by D.Tamas-Selicean et al.[37] to obtain schedulability with minimum resource usage. Genetic Algorithms(GA) are used in Zhang et al.[41]'s research to under-

take task placement in security-sensitive MCS in order to minimize energy consumption. There are also works, which target to optimize task allocation for precedence constrained task graph[31]. The objective is to minimize the makespan. Two optimization approaches were presented, the simulated annealing and Dynamic List scheduling(DLS) heuristic. Li and Baruah[24] compare the use of partitioning scheduling and global scheduling for MCS using a task generation algorithm introduced by Guan et al.[18]. Li and Baruah's research shows that partitioning scheduling performs significantly better than algorithm global scheduling. Socci et al.[34] also studied the task allocation problem of precedence constrained task graph. In whose work the multiprocessor scheduling problem of a finite set of precedence related mixed criticality jobs were studied. An algorithm MCPI (Mixed Criticality Priority Improvement) was presented, which is used to improve the schedulability of a task set. But this algorithm was only for fixed-priority based scheduling and only for scheduling jobs, not tasks.

With a hierarchical scheduler[11][12][23] partitioning is easily achieved. Components of Hierarchical are usually managed as Threads. The Hierarchical Scheduling Framework used by this paper is based on the framework called SF3P[16] from Andres Gomez et al. in ETH zurich. This framework is used to explore and prototype hierarchical compositions of real-time Scheduler. Different from frameworks based on $LITMUS^{RT}$, this framework is a reconfigurable scheduling framework running in the user-space, which means the SF3P could be easily transplanted to other Linux platform since the kernel is not involved. In this framework common interfaces to all scheduling algorithms are defined so that scheduling algorithms are separated from its low-level implementation. Furthermore it could hierarchically compose different scheduling algorithms with its easily extensible structure. However in the SF3P all the scheduling algorithms are designed for uni-processor systems. The effectiveness of SF3P is demonstrated by implementing it on both Raspberry Pi and Intel Core i7 desktop processor.

Later in 2013, a multi-core extension[25] of the SF3P is presented by Lukas Sigrist from ETH zurich, in whose work a lot of new features are added in this new framework in order to support multi-core scheduling and evaluation of the overhead. Two multi-core scheduling algorithms are implemented in this Framework: the Flexible Time Triggered Scheduler (FTTS) and an extension of the single-core EDF-VD scheduler called Partitioned EDF-VD scheduler. Busy-wait tasks are used in simulation to evaluate the effectiveness and overhead of the schedulers. The results showed that the runtime overhead is below 0.15% but will increase when periods and execution times of the tasks decreased. Because only busy-wait tasks are used in simulation, Lukas's work did not provide a method about how to apply a real-life task graph with this framework.

From the related works several conclusions could be made: 1) It is necessary to use multiple criticality levels in MCS, different levels should implement with different scheduling algorithms. 2) Partitioned scheduling has been proved better than global scheduling. In partitioned scheduling more attention should to be payed on task allocation problem according to different optimization objectives. 3) There are still no effective schedulers for scheduling precedence constrained task graph. 4) Implementation of scheduling algorithms are usually using kernel patch on linux, e.g. the LITMUS^{RT} [10], the SCHED_DEADLINE kernel patch[13] and the AQuoSA[28] framework. Using kernel

patch needs a lot efforts since it requires re-verification and re-testing of the kernel. Another way for implementation is adding new functionalities in the form of dynamically loaded kernel-space modules, e.g. the Hijack framework[14] and the ExSched framework[3]. But with dynamically loaded kernel-space module the scheduler are still running partly in the kernel-space and increase the instability of the system. The last way for implementation is to build the whole framework in user-space. The hierarchical scheduling framework is a typical user-space framework.

1.4. Contributions

The contributions of this thesis can be summarized as follows:

- Present the hardware/software co-design for the autonomous driving car by extending Roger Rösch's work[30].
- Design and implementation of two schedulers called Time Triggered Scheduler with mode change (TTS-MC) and Event scheduler in multi-core to support scheduling the precedence-constrained task graph of the autonomous driving car..
- Propose a method, with which real-life task graph could be easily implemented with the hierarchical scheduling framework and provide a method to quantify the scheduling overhead and performance of the schedulers .

1.5. Organization

The remainder of this thesis is organized as follows: In chapter 2 the hardware and software design of the autonomous car are introduced. Chapter 3 describes the modeling of the task graph and introduces the task allocation problem and its solutions. In chapter 4 the scheduling algorithm TTS-MC and Event Scheduler-MC are introduced and Chapter 5 mainly presents the implementation details of these two schedulers. Chapter 6 introduces the functionalities development of path tracking and the method about how to integrate real-life tasks with the HSF. Experiments which evaluate the scheduling overhead and prove the effectiveness of these two schedulers are presented in chapter 7. Conclusion and problems are finally presented in Chapter 8. The information about how to get this framework and installation can be found in Appendix A. Appendix B is the XML files used for experiments.

2. Hardware and software Design for the autonomous car

2.1. Project Overview

This project was firstly started by Roger Rösch and his team in the winter semester 2015 /2016. All of the mechanical structure has been designed and implemented. Solutions for functionalities developments were proposed and some of them have already been implemented but somehow still with problems. After Roger finished his master thesis, this project is handled over on me and the students on the lab-course ": Hardware/Software Co-Design with a LEGO car" in summer semester 2016. The objectives of this project are functionalities development of Path tracking(lane processing), Navigation, Compliance with traffic signs, lights , Collision avoidance and Mixed-criticality scheduling. Students were working on different subtasks (functionality development), while I am mainly focus on the system structure design(both software and hardware), the mixed-criticality scheduling problems, and also part of path tracking functionality development, after all also the integration of the whole system. Modifications were made both in hardwares and softwares.

Figure 2.1 and 2.2 show the overview of the model car. It has four wheel drives and four independent suspensions. Two servo motors are used for steering and a brushless motor with its EZRUN-150A-PRO brushless speed controller is used for speed control. With a 11.7V lithium battery the speed of this car could reach more than 80km/h.

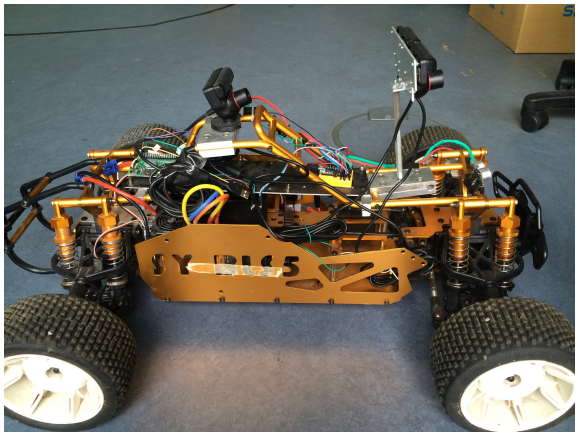


Figure 2.1.: autonomous car overview profile

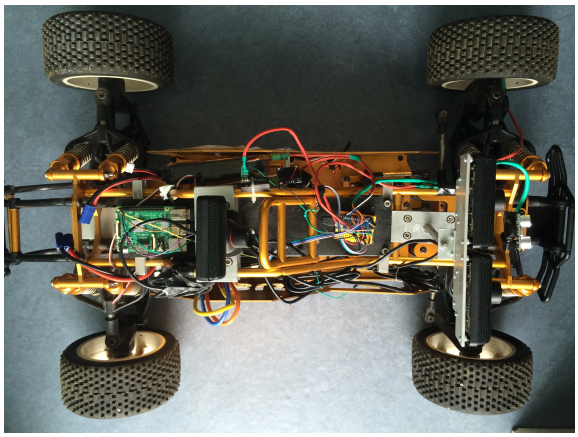


Figure 2.2.: autonomous car overview overlook

2.2. Hardware Design

Modifications have been made on the hardware design. Figure 2.3 and 2.4 show the old hardware design and new hardware design. Table 2.1 shows the details of every used components.

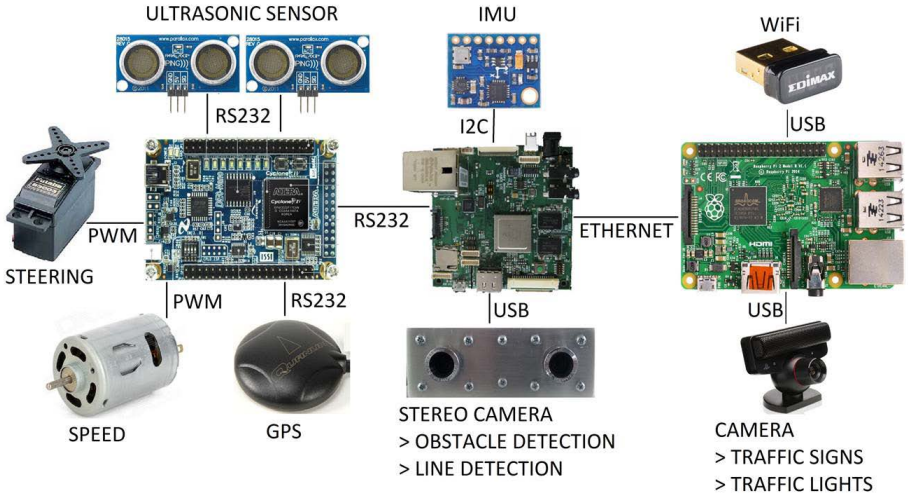


Figure 2.3.: old hardware design

In the old hardware design, Three development boards including Sabre Lite i.MX6, Raspberry Pi and DE0-NANO FPGA are working together, which makes this system a typical distributed system. This solution was replaced by a combination of a Raspberry Pi 3 model B and Atmega 328 micro-controller. There are several reasons for the replacements:

- A distributed system is usually regarded as much more stable as a multi-core platform, but which has also higher cost. Table 2.2 shows the comparison of the costs of the two hardware designs. It shows the cost of the old hardware design is more

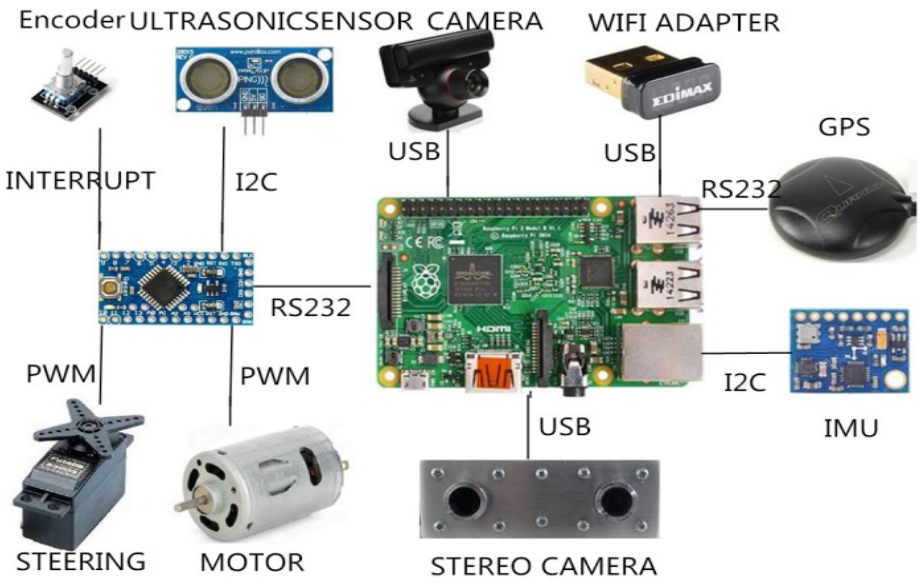


Figure 2.4.: new hardware design

Old Hardware Design	New Hardware Design
Raspberry Pi 2 Model B	Raspberry Pi 3 Model B
DE0-NANO FPGA Development Board	Atmega 328 micro-controller
Sabre Lite i.MX6 Development Board	
Two KS103 Ultrasonic Sensors	Two KS103 Ultrasonic Sensors
Play station eye Camera	Play station eye Camera
Stereo camera	Stereo camera
GPS module	GPS module
IMU	IMU
Wifi Adapter	Remote receiver

Table 2.1.: Comparison of old hardware design and new hardware design

than 10 times expensive as the new solution. This is exactly one of the meanings of develop systems on a multi-core platform.

- The new Raspberry Pi 3 model B delivered on February 2016 has the same price with raspberry pi 2 but is about 1.5 times powerful. As the table 2.3 shows, both of the CPU ,GPU and RAM frequency of raspberry Pi 3 is higher than pi 2. The image processing could save much more times with higher frequency[6]. Higher performance makes the solution of running the whole system on one hardware platform possible.
- The replacement from FPGA to a Atmega328 micro-controller is firstly because of the low cost as the table 2.2 shows. On the other hand in the old design an emulated Nios processor is running on the FPGA. Since it has no permanent memory, re-programming is always needed every time we restart the system. The advantage

Old hardware design	costs	New hardware design	costs
DE0-NANO FPGA	200 Euro	Atmega 328	3 Euro
Raspberry Pi 2	40 Euro	Raspberry Pi 3	40 Euro
Sabre Lite i.MX6	275 Euro		
Total:	515 Euro	**Total:**	43 Euro

Table 2.2.: Cost of processors in old and new hardware design

	Raspberry Pi 2	Raspberry Pi 3
CPU frequency	Quad-core 900MHz	Quad-core 1.2GHz
CPU architecture	Cortex-A7, 32-bit	Cortex-A53, 64-bit
GPU	VideoCore IV 250MHz	VideoCore IV 400MHz
RAM	450MHz DDR2	900MHz DDR2

Table 2.3.: Comparison of Raspberry Pi 2 and 3

of FPGA, parallel computing is totally not utilized. Hence we decide to just use a micro-controller instead of the FPGA. And Atmega328 has a PWM module, supports hardware interrupts, has serial port and I^2C port, which are enough to meet our requirements.

- The last reason is related to the specific task graph of this system. Using distributed structure has less impacts on decreasing the makespan due to the precedence-constrained task graph. More details about this problem will be presented in the section 2.3.

2.3. Software Design

According to the objectives the task graph in figure 2.5 could be designed, and every task's functionality is descried in the table 2.4.

The task graph shows four features, which are important for us to design scheduler:

no.	name	functionality
T1	Capture2	Get a frame from the top camera
T2	SignsProc	Detection and processing of traffic signs
T3	LightsProc	Detection and processing of traffic lights
T4	Capture0	Get a frame from the left of stereo camera
T5	Capture1	Get a frame from the right of stereo camera
T6	LanesProc	Detection of lanes and implementation of steering
T7	DepthMapProc	Generate Depth map and implement collision avoidance
T8	GPSProc	Get location from GPS and IMU
T9	SensorFusionSpeed	Sensor fusion of different sensors and set proper speed
T10	SensorFusionSteering	Sensor fusion of different sensors and set proper steering

Table 2.4.: Functions of every task

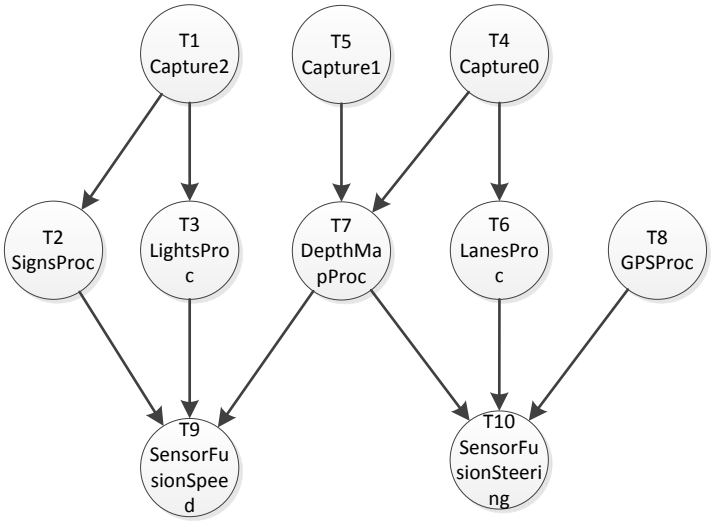


Figure 2.5.: task graph of autonomous car

- **Parallelism:** Inside the task graph there are mainly 5 branches which could be parallel executed, SignsProc, LightsProc, LanesProc, DepthMapProc and GPSProc. Proper using of parallelism would improve the system performance significantly.
- **Precedence-constrained:** Even though part of the application could be parallel executed, there is no totally independent task. Therefore the makespan of the task graph is the maximal execution time of all the branches. This is also the reason why using a distributed system could not decrease the makespan.
- **Mixed-criticality:** In this task graph not all tasks have the same critical level, e.g. DepthMapProc(collision avoidance) and LanesProc(path tracking) are more safety critical compared to other functionalities.
- **Synchronous-periodic:** Every task in the task graph needs to be executed only once in one cycle, which means the task graph has the same period with every individual task.

3. Task allocation

3.1. Comparison of the global and partitioned scheduling

Works of Giovani Gracioli[17] and Alessandra Melani[26] shows a comparison (table 3.1) of global scheduling and partitioned scheduling. Even though the global scheduling has several advantages. This thesis is still focus on partitioned scheduling. Because the global scheduling will bring a number of preemptions and migrations, which produces more overheads. Partitioned scheduling could be divided into a job-shop problem and uniprocessor scheduling problem. The scheduling in uniprocessor is already well-known. And there are various heuristics for the job-shop problem. The project of this thesis is development of an autonomous car, which is better to be compatible with the automotive industry software standard AUTOSAR. Besides the global scheduling algorithms are hard to be implemented with the hierarchical scheduling framework because of the specific structure of the framework. Therefore partitioned scheduling is used in our case and task allocation should be handled properly.

Global Scheduling	Partitioned Scheduling
O automatic load balancing	O Supported by automotive industry (AUTOSAR)
O lower average response time	O No migrations
O more efficient reclaiming	O Isolation between cores
O optimal schedulers exist	O Mature scheduling framework
X higher migration cost	X Cannot exploit unused capacity
X need inter-core synchronization	X NP-hard allocation
X Loss of cache affinity	X System performance depends on allocation

Table 3.1.: Comparison of global and partitioned scheduling (O means advantages while X means disadvantages)

3.2. Notation Declaration

In partitioned scheduling, the behaviour of a task τ_i during scheduling could be defined by dynamic parameters: start time s_i, release time r_i, process time p_i and complete time c_i. Parameters such as WCET(Worst Case Execution Time) C_i, period T_i, static release time R_i, relative deadline D_i, processor index N_i, criticality level χ_i are static assigned parameters. m usually indicates the number of processors of a multi-core platform. The term job of a task τ_i is used to refer a single execution of the task.

- s_i: Start time is the time when a job of task start to execute.

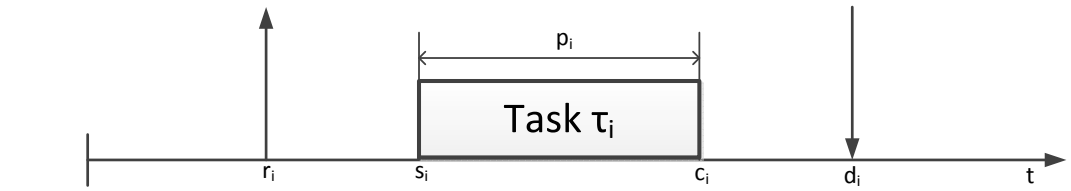


Figure 3.1.: Parameters of a task

- r_i: Release time is the time when the task is released. This parameter is used to describe the release time of a job of a task.
- p_i: Process time is the execution time that a job takes.
- c_i: Complete time is the time when a job finishes its execution.
- d_i: By the deadline a task has to finish its execution. Whether a task has missed its deadline could be determined by comparing the deadline d_i with the completion time c_i. It has usually the following relation: $d_i = r_i + D_i$, where D_i is the relative deadline. If a task could not finish before its deadline, we call it deadline miss.
- C_i: Worst case execution time is the maximum length of time a task could take to execute on a specific hardware platform. Since it is very hard to get the real WCET of a task, a measured maximal execution time is usually referred as WCET. Therefore it is possible that a task exceeds its WCET, we call it overrun of a task.
- T_i: T_i is the execution period of a task. In synchronous-periodic task set, T is referred as the common period of the task set.
- R_i: R_i is a predefined release of a task. This parameter is usually used in time triggered scheduler, where release time of a task is fixed.
- D_i: Relative deadline is a predefined limit time scale, in which a task should have finished.
- N_i: N_i is the processor index on which a task is assigned.
- χ_i: χ_i is the criticality level of a task. This parameter is important in mixed-criticality system.

Precedence constraints are also important features of a task set. Notation $\tau_i \succ \tau_j$ is used to imply that τ_j cannot be started before τ_i completes, it could also be represented as $\tau_i \rightarrow \tau_j$. In implementation a matrix is used to imply the precedence constraints, e.g. the matrix in equation 3.1 shows the precedence constraints of task graph 2.5, where $PREC\,(i,j) = 1$ means $\tau_i \succ \tau_j$.

$$PREC = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.1}$$

3.3. Worst Case Execution Time measurement

In order to allocate tasks with right release time. It is necessary to measure the worst case execution time of a task. Two solutions[39][19][5] are usually used by real-time system developer to measure the WCET of a task:

- **End-to-end measurements of code** is usually performed by setting an I/O pin on the device to high at the start of the task, and to low at the end of the task, using a logic analyzer to measure the longest pulse width, or by measuring within the software itself using the system clock or instruction count.
- **Manual static analysis techniques** is to count the assembler instructions for each function, loop etc. Once the system clock frequency is known the execution time of a task could be determined.

In this thesis, the WCETs were measured using the end-to-end measurements of code with the Linux operating system clock. Every Task is measured for 2 hours and the maximal execution time is regarded as its WCET. Therefore the WCET in this thesis is not exactly the notion of "worst case execution time". This is also the reason why a task is possible to overrun its WCET. It is also impossible to get a real WCET because the interference in a multi-core platform is unpredictable. The maximal measured time will be referred as WCET in following text of this thesis. After measuring the execution time of every task, WCETs were determined in table 3.2. The average execution time is also measured on purpose of analysis the behaviour of a task.

3.4. Makespan Optimization

Makespan indicates the total length of a schedule. It is used as the optimization objective in this thesis because in an autonomous driving system it is necessary to have a short response time. How to get a minimized makespan and its allocation is also called makespan scheduling problem or Job-shop problem.

Task	name	AverExecuT(ms)	WCET(ms)
T1	Capture2	3	9
T2	SignsProc	54	70
T3	LightsProc	43	76
T4	Capture0	5	9
T5	Capture1	6	9
T6	LanesProc	4	10
T7	DepthMapProc	50	72
T8	GPSProc	67	106
T9	SensorFusionSpeed	3	10
T10	SensorFusionSteering	4	10

Table 3.2.: WCET measurement

Makespan scheduling problem is an optimization problem in which n jobs are assigned to m machines. Jobs have varying processing times, which need to be scheduled on m machines with varying or identical processing power. The objective is to minimize the makespan, which in our case is to assign all tasks to four cores of Raspberry Pi to get a task allocation with minimized makespan. As the table 3.1 shows, the task allocation problem of multi-core scheduling is NP-hard, which means optimal solution could not be obtained in polynomial time and meanwhile the solution could not be verified in polynomial time as well. Heuristics such as list scheduling, Polynomial Time Approximation Scheme could be used to get the suboptimal solution of task allocation.

3.4.1. Graham's List Scheduling

The list scheduling is a heuristic which was designed to solve online scheduling problem by Graham[22] in 1960s. It could also be used to get task allocation offline. The list scheduling[1] algorithm works as follows: Determine any ordering of the job set J, stored in a list L. Starting with all machines empty, determine the machine i with the currently least load and schedule the respective next job j in L on i. It is proved that list scheduling is a 2-approximation algorithm. Sorted list scheduling is a modified list scheduling, on which the list L consist of the jobs in decreasing order of execution length. It is proved that sorted list scheduling has a $4/3-$approximation. List scheduling has a complexity $O(n)$, while sorted list scheduling has a complexity $O(n * log(n))$. With a relative low complexity are list scheduling and sorted list scheduling usually used in online global scheduling. The algorithm 1 shows the process of using sorted list scheduling to get task allocation offline.

3.4.2. Polynomial Time Approximation Scheme

Polynomial Time Approximation Scheme(PTAS)[1] has a $(1+\varepsilon)$-approximation, ε could be any value > 0. The PTAS has a complexity of $O\left(n^{2k} \cdot [log_2 1/\varepsilon]\right)$, where $k = [log_{1+\varepsilon} 1/\varepsilon]$ Which means, the more accurated solution we could get, the more time the algorithm will

Algorithm 1: Sorted List Scheduling

```
Sort list L in decreasing order according to execution length;
while List L is not empty do
    choose a processor with least task load;
    get a task T_i from List L;
    while Precedence constraints of T_i is not met do
        T_i = next Task in L;
    end
    assign task T_i on the processor with least task load;
    Delete T_i from L;
end
```

take. The process of PTAS is: (1) Firstly, assume that the optimal makespan T^* is given at the outset. Then we can try to construct a schedule with makespan at most $(1+\varepsilon)\cdot T^*$. Binary search is performed in an interval $[\alpha, \beta]$, where α is any lower bound on T^* and β any upper bound on T^*. This binary search will enable us to eventually find a number B, which is within $(1+\varepsilon)$ times T^* and where the number of binary search iterations depends on the error parameter ε. (2) Secondly, assume that the number of distinct values of job lengths is a constant k. Then we can determine all configurations of jobs that do not violate a load bound of t if scheduled on a single machine. This is the basis of a dynamic programming scheme to determine a schedule on m machines. Of course, this approach involves rounding the original job lengths to constantly many values, which introduces some error. The error can be controled by adjusting the constant k of distinct job lengths at the expense of running time and space requirement for the dynamic programming table.

3.5. Torsche scheduling toolbox

Torsche (Time Optimization of Resources, Scheduling) scheduling toolbox[36] is a Matlab based tool, which offers a number of solutions that allow the user to formalize various off-line and online scheduling problems. The task allocation problem in our case could be easily implemented with this toolbox. Since the Torsche scheduling toolbox already supports offline list scheduling, it is used to get a suboptimal task allocation solution. The following sentence shows how the list scheduling is applied in Torsche toolbox.

```
TS = listsch(T,problem,processors [,strategy])
```

Here T is the set of tasks. problem indicates the objective of optimization, in our case problem is 'P|prec|Cmax', which means to get an suboptimal makespan in precedence constraints allocation problem. processors indicates the total number of processors. Strategy means the order of list we are using. Since we are using sorted list scheduling, LPT(Longest Processing Time first) is used as the strategy. If the WCET is taken as every task's execution time, the makespan problem could be described in the following codes:

Listing 3.1: Sorted List Scheduling with Torsche scheduling toolbox

```
t1=task('T1',9);
t2=task('T2',73);
t3=task('T3',76);
t4=task('T4',9);
t5=task('T5',9);
t6=task('T6',10);
t7=task('T7',72);
t8=task('T8',106);
t9=task('T9',10);
t10=task('T10',10);

prec =
[0 1 1 0 0 0 0 0 0 0;%
 0 0 0 0 0 0 0 0 1 0;%
 0 0 0 0 0 0 0 0 1 0;%
 0 0 0 0 0 1 1 0 0 0;%
 0 0 0 0 0 0 1 0 0 0;%
 0 0 0 0 0 0 0 0 1 1;%
 0 0 0 0 0 0 0 0 1 1;%
 0 0 0 0 0 0 0 0 0 1;%
 0 0 0 0 0 0 0 0 0 0;%
 0 0 0 0 0 0 0 0 0 0];%

T = taskset([t1 t2 t3 t4 t5 t6 t7 t8 t9 t10],prec);
p = problem('P|prec|Cmax');
TS = listsch(T,p,4,'LPT');
plot(TS);
```

As a result we could get the task allocation in figure 3.2 and table 3.3.

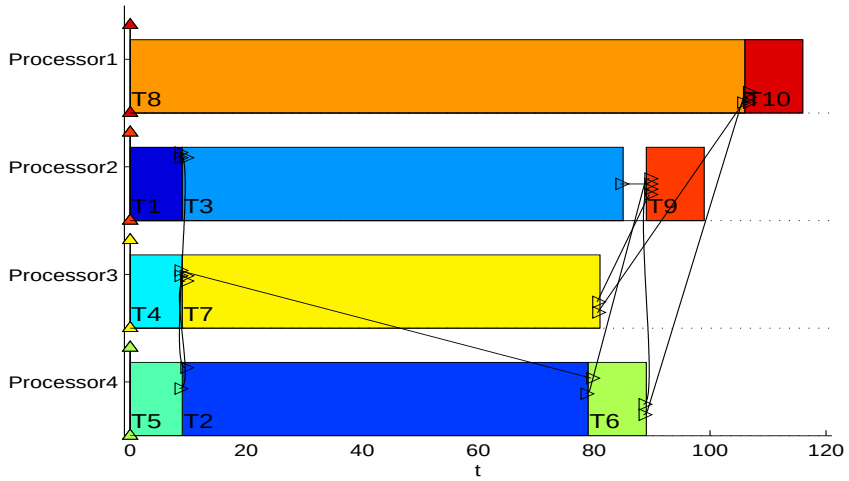


Figure 3.2.: Task allocation from Sorted List Scheduling

Task	name	Core Index	s_i(ms)
T1	Capture2	2	0
T2	SignsProc	3	9
T3	LightsProc	2	9
T4	Capture0	3	0
T5	Capture1	4	0
T6	LanesProc	4	81
T7	DepthMapProc	4	9
T8	GPSProc	1	0
T9	SensorFusionSpeed	2	91
T10	SensorFusionSteering	1	106

Table 3.3.: Task allocation

4. Mixed-Criticality scheduling of precedence-constrained task set

In this section the problem of mixed criticality scheduling is presented. A scheduler named Time Triggered Scheduler with Mode Change(TTS-MC) and Event Scheduler in Multi-Core(Event Scheduler-MC) are presented. The correctness of these two schedulers are also be proved.

4.1. Problem description

Mixed Critical Task Set According to section 2.3, the task set in our case is synchronous-periodic, therefore we only consider the scheduling problem of periodic task system. A periodic mixed critical task set could be characterized as $\tau = \{\tau_1, ...\tau_n\}$. T is the period of task set τ. Each task can have a criticality level from 1 (lowest) to L (highest).

A task could be characterized by $\tau_i = \left\{\chi_i, \overrightarrow{C_i}, \overrightarrow{R_i}, D_i, N_i\right\}$, where the parameters are characterized as follows:

- χ_i: $\chi_i \in \{1, ..., L\}$, is the criticality level of the task.
- $\overrightarrow{C_i}$: is a size-L vector of different worst case execution time(WCET). $C_i\,(l)$ indicates the worst case execution time on criticality level l.
- $\overrightarrow{R_i}$: is also a sized-L vector of the defined release times of a task. In our case a task has multiple release times.
- D_i: is the relative deadline of a task.
- N_i: is the assigned core index of task τ_i.

The reason why we need multiple WCETs is that the WCET is usually unnecessary pessimistic especially the WCET of high critical tasks. If we use only one WCET, then this WCET must be a large value to prevent deadline miss. As a result a lot of computational resource will be wasted. Therefore different WCET values could be assigned according to different pessimistic levels. As mentioned in our case we get the WCET by measuring the maximal execution time of a task in a certain time. This value will be referred as the $C_i(1)$ (in dual-critical task system $C_i(LO)$) of task τ_i. The multiple release times are only needed in scheduler TTS-MC. This will be declared in section 4.2.

Utilization The utilization[25] of a periodic task is defined as the ratio between the task's worst case execution time and its period. For a task set it is the sum of all task utilizations.

Because the WCET varies from different critical levels. There are also different utilizations in different critical level. The following formula defines the $level-l$ utilization for the task τ_i:

$$U_i(l) = \frac{C_i(l)}{T_i} \tag{4.1}$$

Mixed-criticality scheduling problem[25] The execution times of jobs of a task varies from period to period. The mixed-criticality scheduling problem is to assure the following two points. When the following two points are guaranteed during scheduling, we call the schedule is *correct*.

- If all jobs run with the lowest criticality execution profile (i.e. no task τ_i exceeds its WCET in level 1 ($C_i(1)$)), then all jobs could complete before their deadlines.
- If any job τ_i, which has criticality level greater than l ($\chi_i > l$), exceeds its $level - l$ WCET($C_i(l)$), then all jobs with criticality level l or higher could complete execution before their deadlines.

4.2. Time Triggered Scheduler with Mode Change

The time triggered scheduler with mode change(TTS-MC) is designed for precedence-constraints task set. Same with the EDF-VD scheduler[4], tasks will be divided into two criticality levels: HI and LO. And the scheduler has also two scheduling modes: HI critical mode and LO critical mode(also represented as HI-mode and LO-mode). Different from EDF-VD, tasks are released at static assigned trigger time. Task set partitioning and allocation are solved by the makespan minimization algorithm mentioned in section 3. High critical tasks will be assigned with two WCET values. Rather than virtual deadlines in EDF-VD, the $C_i(LO)$ is used as the mode change point. Because the task set is time triggered, overrun of either high critical tasks or low critical tasks will delay other task's execution, which might lead to miss deadline of high critical tasks. Therefore scheduler will change to HI-mode whenever a task has overrun its $C_i(LO)$. At this point the scheduler will cancel all the running LO critical tasks in all cores. In HI-mode only high critical tasks will be scheduled with modified trigger time, which gives the high critical tasks more budget time to execute but without changing the period. In a new cycle the critical mode of scheduler will change back to LO. A cycle has also criticality level. We call a cycle is a high critical cycle($X = HI$), if in this cyle the critical mode has changed to HI. A cycle is a low critical cycle ($X = LO$) if no overrun happens in this cycle. The utilization of a task set with TTS-MC is:

$$U_{total}(LO) = \frac{\sum_1^n C_i(LO)}{T} \leq m \tag{4.2}$$

$$U_{total}(HI) = \frac{\sum_{\chi_i=HI} C_i(HI)}{T} \leq m \tag{4.3}$$

Where m is the number of processors. We could say the scheduling with TTS-MC is correct under the following two conditions:

- **(1)** All tasks in LO critical cycle could finish before their deadlines. Equation 4.4 describes the bound the scheduling should meet.

$$\forall \tau_i \in \tau, X = LO, s_i + p_i \leq R_i(LO) + D_i \tag{4.4}$$

- **(2)** HI critical tasks in HI critical cycle could finish before their deadlines.

$$\forall \tau_i \in HI, X = HI, s_i + p_i \leq max(R_i(LO), R_i(HI)) + D_i \tag{4.5}$$

Where X indicates the critical level of the cycle, s_i is the start time of a job of the task, p_i is the execution time, $R_i(LO)$ is the low critical release time, $R_i(HI)$ is the high critical release time., D_i is the relative deadline. It needs to be mentioned that in TTS-MC the release time R_i is also not necessary the same with the start time s_i. Hence the deadline of a task is defined from the release time to the absolute deadline. We could say the scheduling with TTS-MC is correct if the equations 4.4 and 4.5 are met.

The figure 4.1 shows the flow-process of TTS-MC scheduler. Since mode change and task cancellation are the most important mechanisms of the TTS-MC, they will be discussed in detail.

Mode change The challenge of mode change is to change the release time of high critical tasks after mode change without changing the period of task set. It requires double assignments of release times. The two release times should be switched properly. A high critical task's $R_i(HI)$ could be either bigger or smaller than $R_i(LO)$. The mode change could happen at different times. Therefore the there are totally six possibilities as the Figure 4.2 shows. The implementation of mode change will be discussed in section 5.2. The analysis of the six possibilities shows that mode change could work well only at time point ①②④. But it will not affect the schedulability according to the proof.

- ①② Mode switch happens at the positions before $R_i(HI)$ and $R_i(LO)$. The task will get enough time to switch its release time.
- ⑤⑥ Mode switch happens at the positions after $R_i(HI)$ and $R_i(LO)$. The dispatcher is not possible to switch its release time because the task has released at $R_i(LO)$. In this case the task τ_i has the same release time in high critical mode as in low critical mode.
- ③ Mode switch happens after $R_i(HI)$ but before $R_i(LO)$. In this case the task still releases new job at $R_i(LO)$, but the high critical will not get more budget time any more.
- ④ Mode switch happens after $R_i(LO)$ but before $R_i(HI)$. Task will also release at $R_i(LO)$, and will get more budget time for execution.

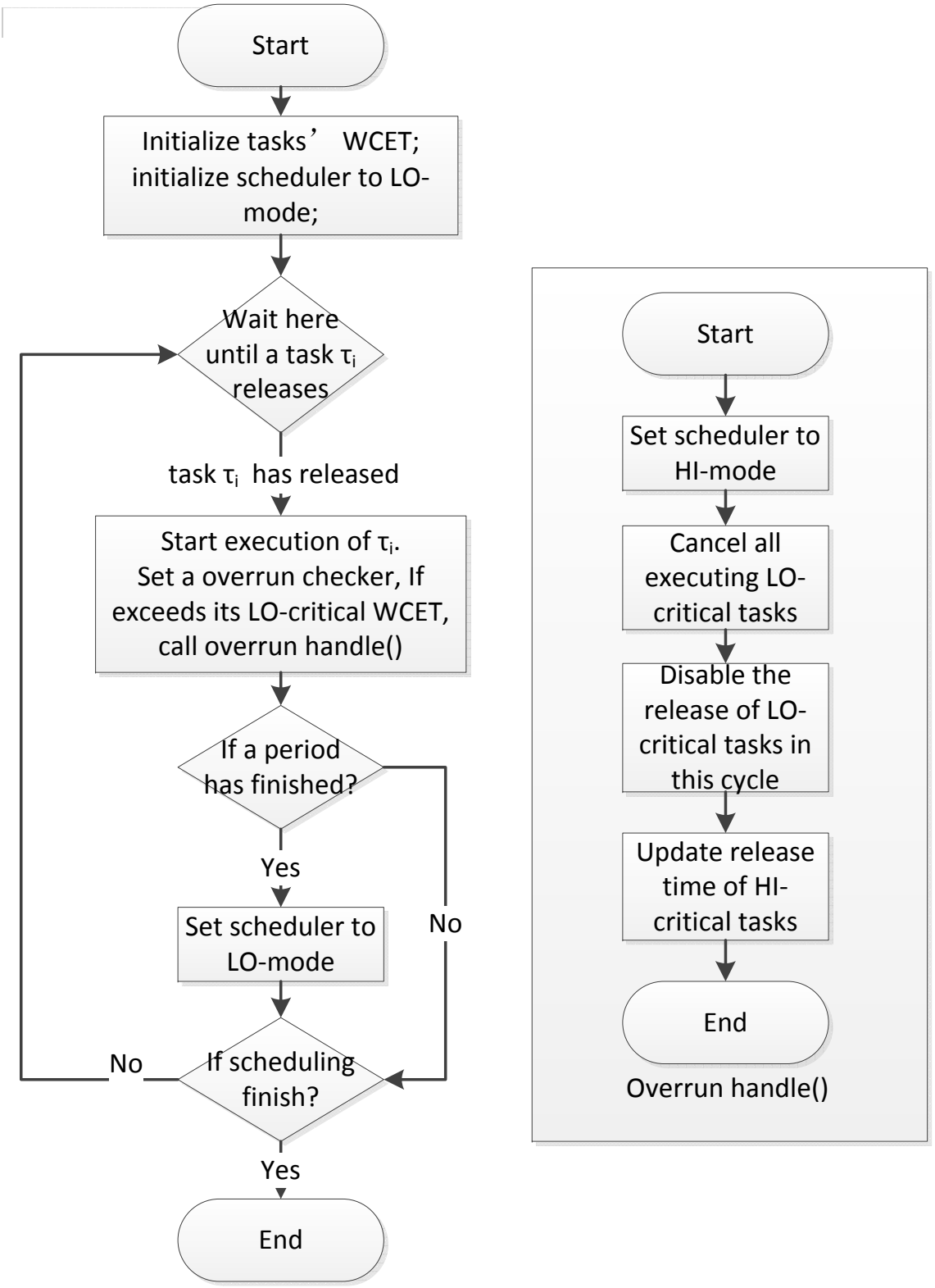


Figure 4.1.: Diagram of TTS-MC

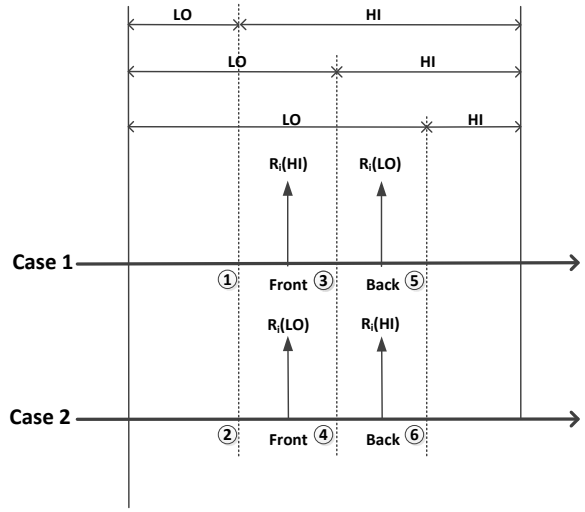


Figure 4.2.: Double release time and mode change

Task cancellation Another challenge of mode change is the cancellation of an executing task. In simulation, all tasks are busy-waiting, which could be easily canceled with the code structure (1) in Listing 4.1. But with real-life task in autonomous driving, few task could be restructured like this. One of the solution is to use a do-while structure as the structure (2) shows. But apparently this structure is also not suitable for all cases. Because some of the tasks could not be divided into several operations with small execution time. Therefore the WCET assignment for low critical tasks, which are not able to be canceled, should be very careful. If a task is low criticality, and it could not be restructured neither the structure (1) nor (2). Then it should be assigned with a pessimistic WCET in the design phase to prevent it from overrun since it cannot be canceled. There is no theory or good proved method to provide a suitable WCET. Hence in our case twice of the maximal execution time is regarded as a pessimistic WCET.

Listing 4.1: Code structure of task cancellation

```
Structure (1):

cancel_ = false;
for(i:n){
...;
if(cancel_)
break;
}

Structure (2):

cancel_ = false;
do{
operation 1;
if(cancel_ = false)
break;
operation 2;
if(cancel_ = false)
break;
operation 3;
if(cancel_ = false)
break;

...;

}while(false)
```

Prove of correctiveness In order to prove the correctiveness, some assumption should be made. (1) We assume high critical tasks will never overrun its $C_i(HI)$. (2) LO critical tasks, which cannot be canceled, will be assigned with an pessimistic value as its $C_i(LO)$ so that overrun is not possible.

- In LO criticality cycle, because no overrun of either high critical tasks or low critical tasks happened, therefore the start time of a task is exactly its release time $s_i = R_i(LO)$. The execution time will not exceed its WCET $p_i \leq C_i(LO)$, we could get the following equation 4.6. If we set the relative deadline $D_i \geq C_i(LO)$, then the condition of equation 4.4 is met.

$$\forall \tau_i \in \tau, X = LO, s_i + p_i \leq R_i(LO) + C_i(LO) \tag{4.6}$$

- In HI critical cycle, the start time s_i of the high critical tasks met the following equation 4.7. And according to our assumption the execution time of high critical tasks will not exceed their WCET $p_i \leq C_i(HI)$.

$$s_i \leq max(R_i(HI), R_i(LO)) \tag{4.7}$$

  Hence we could get the following equation 4.8. If we set the relative deadline $D_i \geq C_i(HI)$ for high critical tasks, then the condition of equation 4.5 is met.

$$s_i + p_i \leq max(R_i(LO), R_i(HI)) + C_i(HI) \tag{4.8}$$

Therefore under the assumption (1)(2) we could get the condition of correctiveness is to set a right relative deadline as in 4.9 shows.

$$\begin{cases} \forall \tau_i \in (\chi_i = LO), D_i \geq C_i(LO) \\ \forall \tau_i \in (\chi_i = HI), D_i \geq C_i(HI) \end{cases} \tag{4.9}$$

4.3. Event Scheduler in Multi-Core

Even though the TTS-MC's correctiveness could be proved. With WCET set as the assumed execution time of tasks are still too pessimistic. A lot of computational resource will be wasted. A direct way to solve this problem is to release a task as soon as its precedence tasks have finished. With this idea the event scheduler is presented. In event scheduler task τ_i in $\tau_i \rightarrow \tau_j$ has finished or not is regarded as an event for task τ_j. The task τ_j should wait for the event of τ_i finished happen then continue its execution. All of tasks are triggered at the start point of a period with a predefined order. Like the task allocation of TTS-MC, the WCET of tasks could be used in makespan optimization to get a period T of the task set. In practice we observed that the period T is usually not necessary. Maximal makespan measured by running the system for a certain time could also be used as the period. Same with the TTS-MC, overrun of LO critical tasks will be canceled. Figure 4.3 shows the diagram of the flow-process of event scheduler-MC.

Prove of correctiveness If we define the correctiveness of event scheduler the same with TTS-MC. When the period of task set in event scheduler is set the same value with TTS-MC, then the correctiveness of event scheduler could be easily proved since $s_i \leq max(R_i(HI), R_i(LO))$. Equation 4.5 and 4.4 are met.

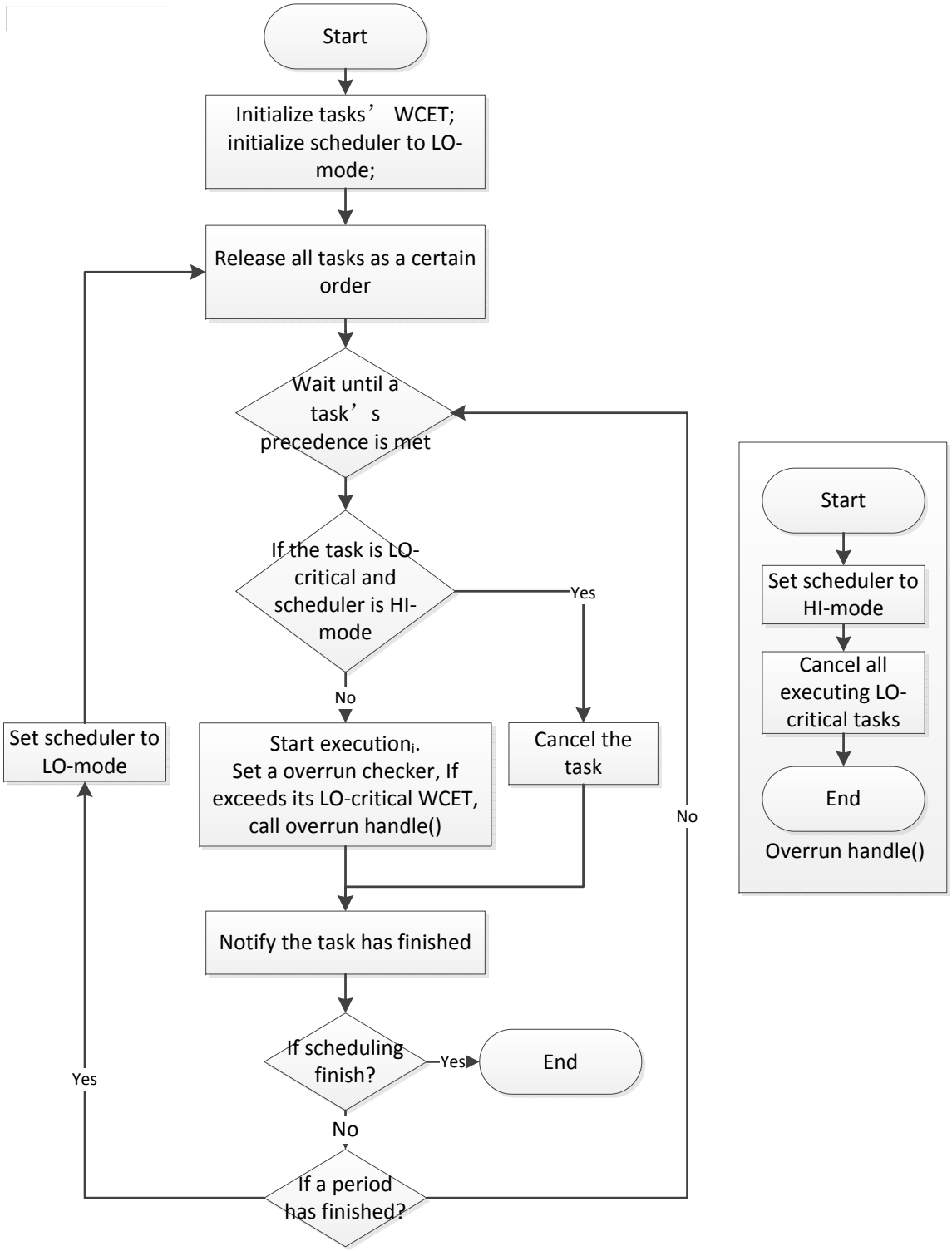


Figure 4.3.: Diagram of Event scheduler-MC

4.4. Scheduling examples

In order to illustrate how the introduced TTS-MC and event scheduler works, an example task set with four tasks is introduced in figure 4.4.

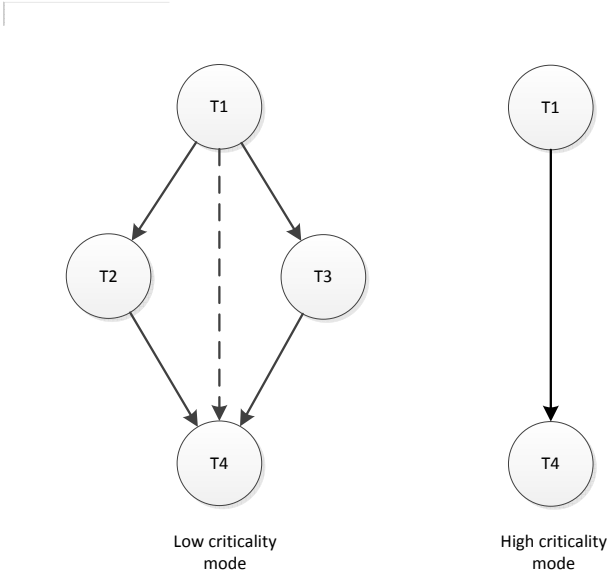


Figure 4.4.: An example task set with four tasks

The task set could be characterized as $\tau = \{T1, T2, T3, T4\}$. The individual task can be characterized as follows, the time unit is millisecond:

$$\begin{cases} T1 = \left\{80, HI, \overrightarrow{C_1}(25, 40)), \overrightarrow{R_1}(0, 0), 80, 1\right\} \\ T2 = \{80, LO, 25, 30, 80, 1\} \\ T3 = \{80, LO, 25, 30, 80, 2\} \\ T4 = \left\{80, HI, \overrightarrow{C_4}(15, 30)), \overrightarrow{R_4}(65, 50), 80, 1\right\} \end{cases} \tag{4.10}$$

As the Figure 4.5 shows, the TTS-MC scheduler is running for three cycles with period 80ms. First cycle without any overrun, second cycle with an overrun of high critical task $T1$. Therefore the low critical tasks $T2$ and $T3$ are aborted. At the same time the release times as well as the WCET of tasks $T1$ and $T4$ are changed and the scheduler switches to high critical mode. In third cycle there is an overrun of low critical task $T2$, but actually the low critical task will never overrun because it will be canceled exactly at the time point when it overruns. When a new cycle comes at time point 160ms the critical mode of scheduler switches back to low. It is necessary to mention that between the allocation of two tasks, there should be an offset time, which gives the scheduler a little bit time (1-2ms) to switch the critical mode when an overrun occurs.

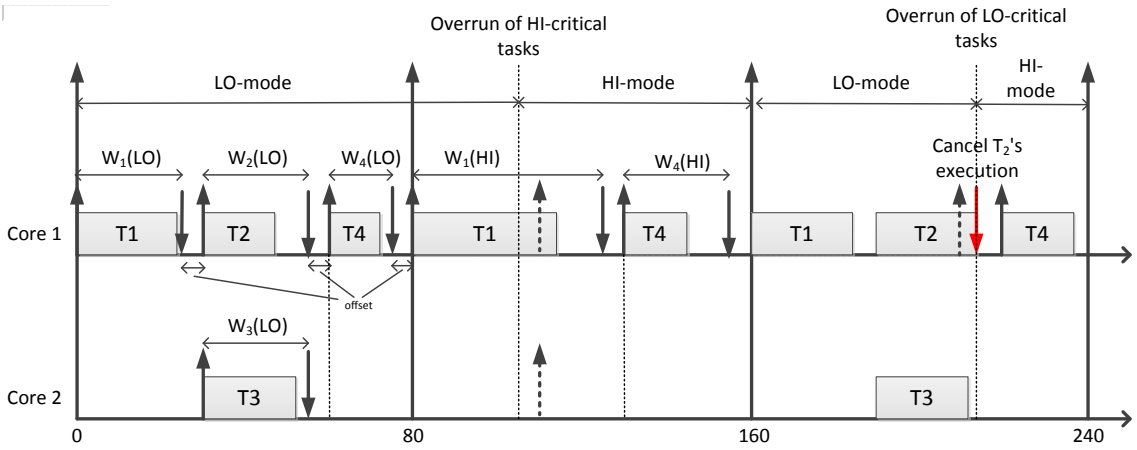


Figure 4.5.: TTS-MC example schedule

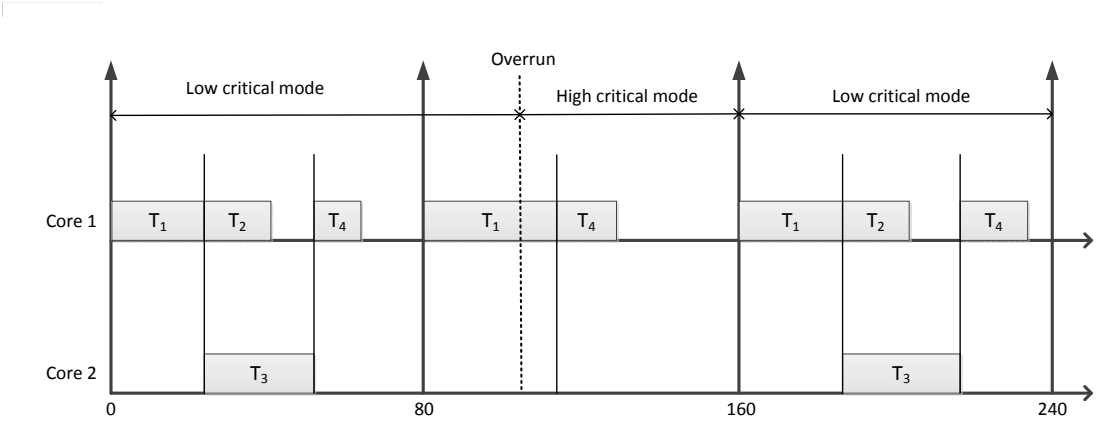


Figure 4.6.: Event scheduler example schedule

The Figure 4.6 shows an example scheduling with event scheduler-MC. In the second cycle there is also an overrun of task $T1$. Same with the TTS-MC low critical tasks $T2$ and $T3$ are aborted. As a result task $T4$ could start immediately after $T1$ finish.

5. Implementation

5.1. Hierarchical Scheduling Framework

The framework called hierarchical scheduling framework(HSF) is based on a framework to explore and prototype hierarchical compositions of real-time schedulers(SF3P)[16] .The SF3P enables the comparison and verification of different scheduling algorithm on uniprocessor platform. It separates the configuration of scheduler from the implementation of the algorithm itself, configuration is written in a XML file. HSF[25] has extended the SF3P to support multi-core mixed-criticality scheduling. Both of the SF3P and HSF operates directly on top of the Linux concurrency manager in user space and adds an additional layer between the tasks and the scheduling mechanisms of the operating system. The figure 5.1 shows the overview SF3P and HSF. Since the framework is located in user space without kernel involved, it allows the framework to be portable and compatible with almost all hardware platforms with Linux operation system.

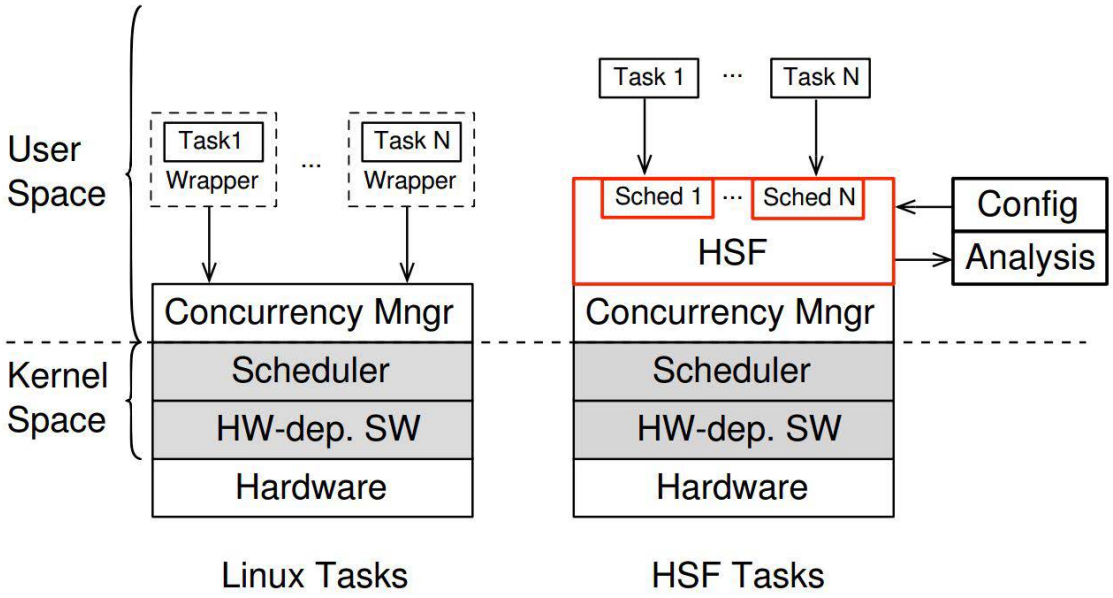


Figure 5.1.: Added HSF layer

The Figure 5.2 shows the overview of the structure of Hierarchical Scheduling Framework. As the Figure shows, the framework consists of INPUT, RUNTIME and OUTPUT. INPUT is a XML File while the OUPPUT are CSV files. Different from our case the origi-

nal HSF has a part named SIMULATION rather than RUNTIME without tasks. The first part of RUNTIME is the parser, which is in charge of reading configurations from the XML file. Configurations consist of properties of dispatchers, schedulers, workers and tasks. The new added component task is because the original HSF is only used for simulating and evaluate different scheduling algorithms with busy-waiting tasks, but in our case real tasks are used for scheduling, therefore a parser for different tasks are needed. Dispatcher is used to release a new job of a task. A task could be released by time, event or in specific condition. Scheduler is used to manage the whole scheduling with a given scheduling algorithm. Worker is used to manage the execution of a task. Workers, schedulers and dispatchers are running as threads. The last part is the statistics and output module, which collects traces during runtime. Scheduling overhead will be calculated according the traces data and finally all the data will be saved in different CSV files.

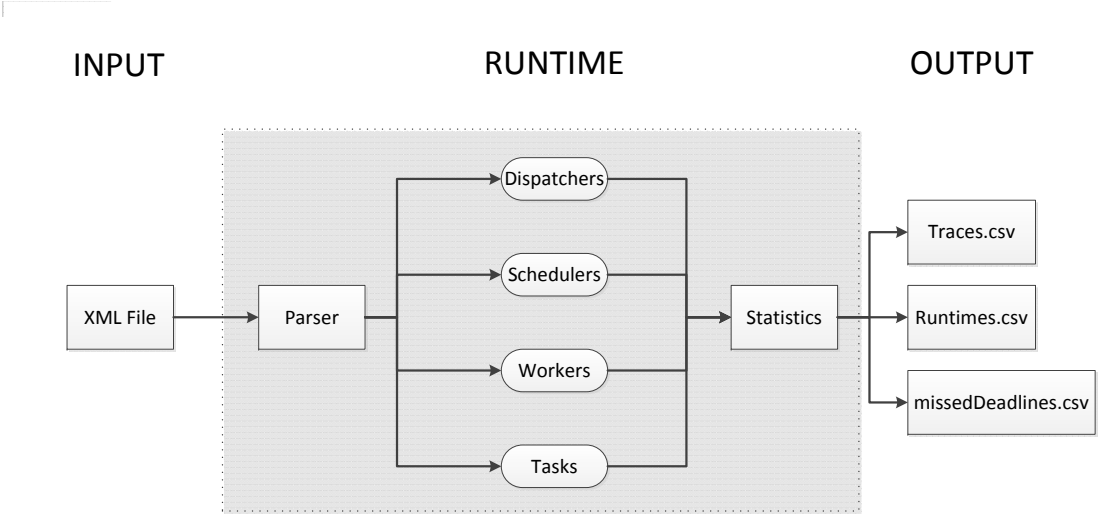


Figure 5.2.: Overview of Hierarchical Scheduling Framework

5.1.1. The Thread Hierarchy

In order to understand how the framework works in code level, it is necessary to understand the thread hierarchy. The thread hierarchy represents the derive relation of different components used in scheduling. In order to support the TTS-MC and Event scheduler-MC that presented in chapter 4, several modifications have been done and several new components are added to the original thread hierarchy. We describe here briefly the functionalities of used components and then present the changes that were made. The structure of the thread hierarchy is showed in figure 5.3. New or added components are marked as red. Modified components are marked as green.

From the figure 5.3 we could see that everything is derived from a Thread class. Different hierarchy level in this structure has different functionalities. Introduction of important used components:

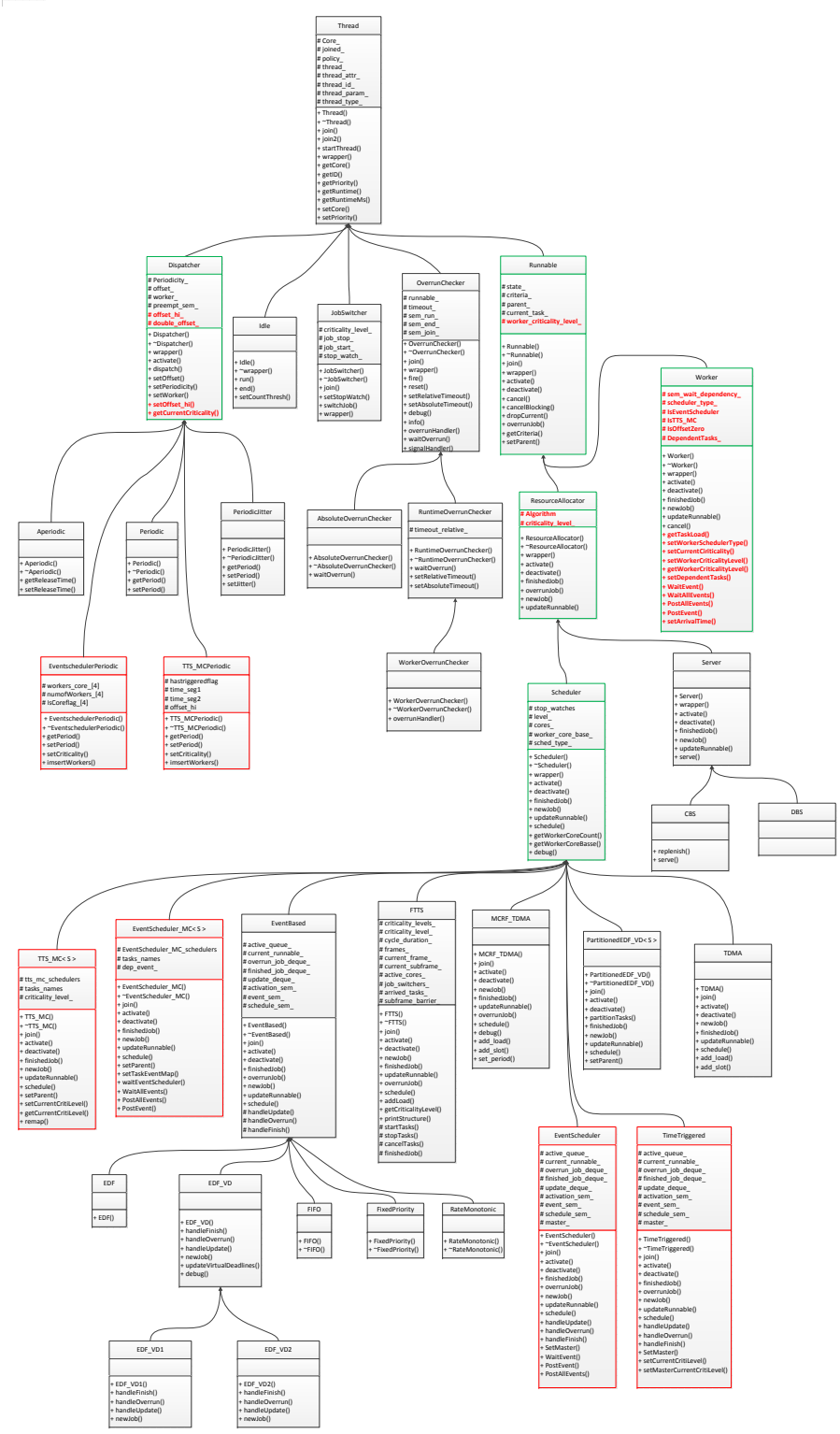


Figure 5.3.: Thread hierarchy

- **Thread** is the parent class of all of other components. This class is an object oriented wrapper around a POSIX thread (pthread) with some additional management functions. It creates a pthread object in a specific core by using $pthread\_create()$ function.
- **Idle** is refered as the idle state of a core. But in implementation it is running with busy waiting with the lowest priority. Therefore it could be preempted easily by the scheduler, worker or dispatcher.
- **Dispatcher** is in charge of triggering a new job of a task. In the original version of HSF there are three kinds of dispatchers: Periodic is used to dispatch a task periodically. Aperiodic is used to trigger a task only once, while the PeriodicJitter is used to trigger a task randomly.
- **OverrunChecker** is used to monitor a task execution. It has two different types. The AbsoluteOverrunChecker checks if a task has executed longer than a global timestamp, while the RuntimeOverrunChecker checks if a task has executed beyond a defined duration.
- **Runnable** is the parent class of worker and scheduler, which means the sorker and scheduler have some part of common interfaces. The reason why common interface is needed is that in the hierarchical framework a scheduler could also be scheduled by another scheduler[16].
- **Worker** is used to manage the execution of tasks. It received task release notification from dispatcher and interact with scheduler to execute the task properly.
- **Scheduler** implements different scheduling algorithms and interact with workers to manage proper execution of tasks.

Red marked components "EventschedulerPeriodic" and "TTS_MCPeriodic" are dispatchers designed only for TTS-MC and Event scheduler-MC. "TimeTriggeredScheduler" is a time triggered scheduler for uni-processor platform, while "Eventscheduler" is a event scheduler for uni-processor platform. TTS_MC is a scheduler consists of several TimeTriggeredSchedulers to achieve multi-core functionalities. Same with TTS_MC the Eventscheduler_MC consists of several Eventschedulers. The component worker has been added with a lot of new member functions and variables to achieve new functionalities.

5.1.2. Scheduling Mechanism

In order to implement scheduling from the user space, the framework uses priority-based scheduler with different priorities the operating system provides. Threads with higher priorities could preempt threads with lower priority. In practice the activation of a component is achieved by dynamically improving its priority. The Figure 5.4 and table 5.1 shows the priorities assignments of different components. An idle thread is used to represent the idle state of a core as mentioned in section 5.1. It simply does busy waiting and could be preempted by any thread with higher priority. The worker in active state has one level higher than the idle thread. The dispatcher has a higher level of priority than an active worker. Active scheduler has the highest priority in thread hierarchy. The priority

of the framework which organizes the whole system has the highest priority than all the mentioned components, which is not showed in the figure 5.4.

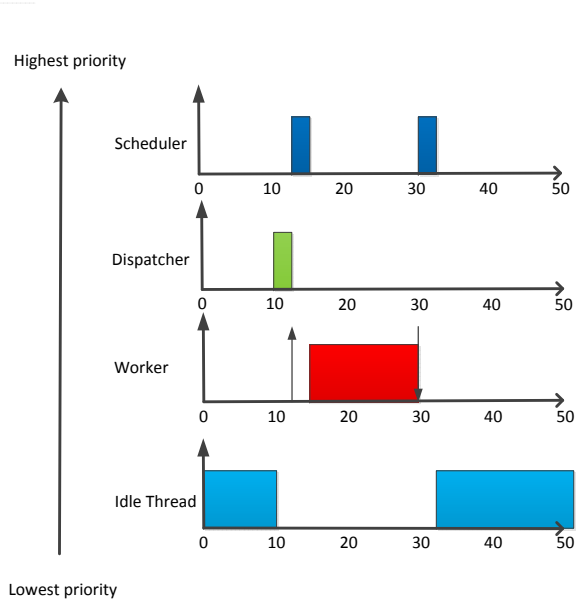


Figure 5.4.: Priority assignment of different components[25]

Priority	Threads
Highest	Framework, to manage the whole system
Highest in hierarchy level	Scheduler
5	Dispatcher, overrun checkers
4	Scheduling servers
3	Worker (active state)
2	Idle
1	Worker and Scheduler in inactive state

Table 5.1.: Priority assignment of different components[25]

During scheduling, the dispatcher is running independently with a predefined trigger time, while the worker and scheduler both has several states and interact with each other. The figure 5.5 shows the state flow of scheduler and worker. Figure 5.6 and 5.7 shows an example of how the framework works. The worker starts with the state idle. When the dispatcher releases a new job on this worker, the state of worker turns to ready. Then the worker notifies the scheduler the arrival of new job and keep ready state until the scheduler notifies the worker that it could start execution. Then the state turns to active and worker start execution of a job. When the executed job is preempted, the state turns back to ready. After the job is finished, the worker notifies the scheduler a finish event and then turns back to idle.

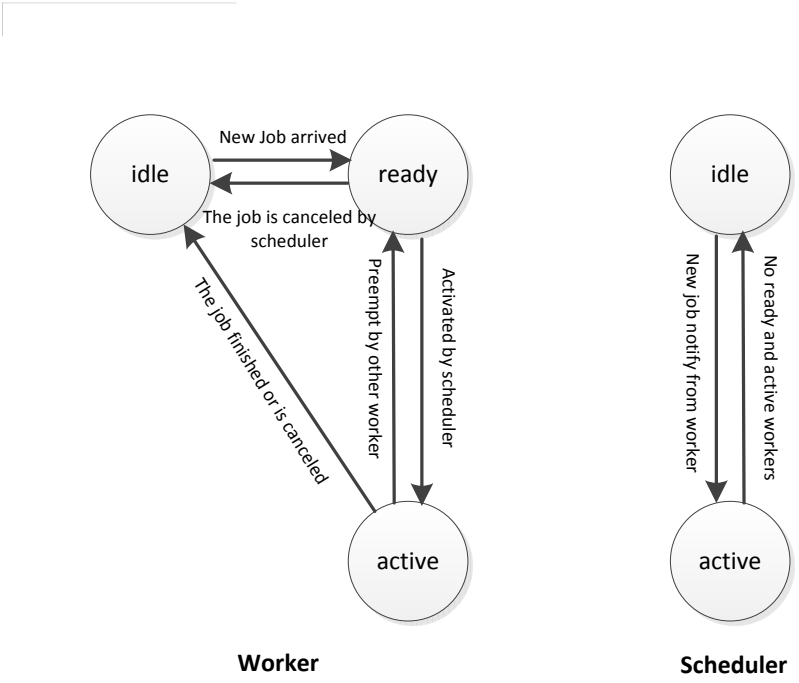


Figure 5.5.: State flow of Worker and Scheduler

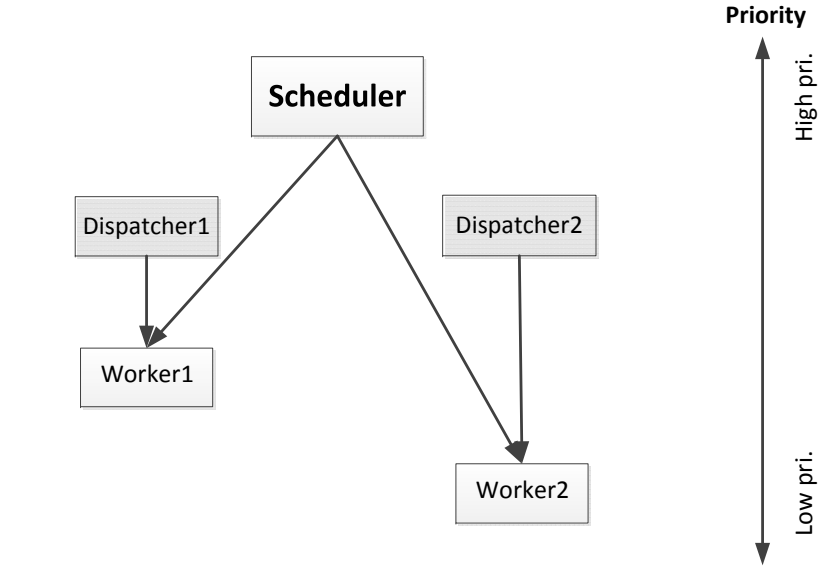


Figure 5.6.: An example of scheduler

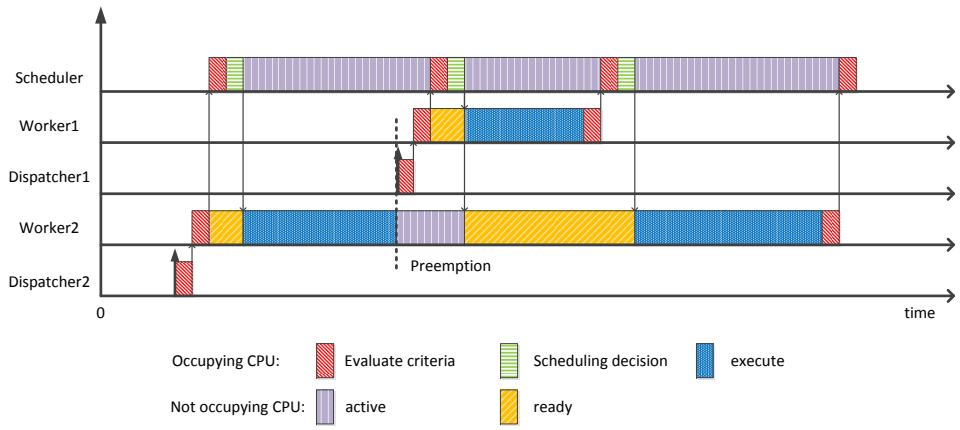


Figure 5.7.: Sample execution of a scheduler in Figure 5.6

The scheduler also starts with state idle, turns to active once get a job arrival notify. It turns back to idle when there is no more ready and active workers. It is necessary to mention that in [16] the scheduler also has three states exactly as worker. Because in SF3P a scheduler could also be scheduled by another scheduler. Since in our case we did not consider the situation where a scheduler is scheduled by another one. Therefore it is not showed in the figure 5.5.

5.2. Implementation of TTS-MC

When we talk about design of a scheduler, it means we need both design the dispatcher, worker and scheduler components in HSF. The structure of the TTS-MC shows in Figure 5.8. The TTS-MC consists of four normal time triggered schedulers(TTS). Each core has its own TTS, which is only in charge of scheduling tasks assigned on this core. Every TTS has a pointer $*master$, which points to the TTS-MC, so that scheduler in different cores could communicate with each other.

The TTS is changed from the "Eventbased" scheduler in [25]. Therefore in the TTS scheduling is controlled by a semaphore event $event\_sem\_$, it represents different events during scheduling, e.g. finish of a job, overrun of a job, new arrival of a job. Every TTS has a worker queue $active\_queue\_$ which is used to store all the workers with new job arrival. The variable $current\_criticality\_level$ indicates the criticality mode of the scheduler. When an overrun occurs, the TTS calls setMasterCurrentCritiLevel() of TTS-MC to set $current\_criticality\_level$ in all TTS to HI. The algorithm of a single TTS is shows as algorithm 2. The worker of TTS-MC shows in algorithm 3.

Since high critical tasks have two release times. The release time switch should handled properly. According to figure 5.9 the algorithm of dispatcher for high critical task could be represented as algorithm 4. Where $offset$ is the time before the first release of a task.

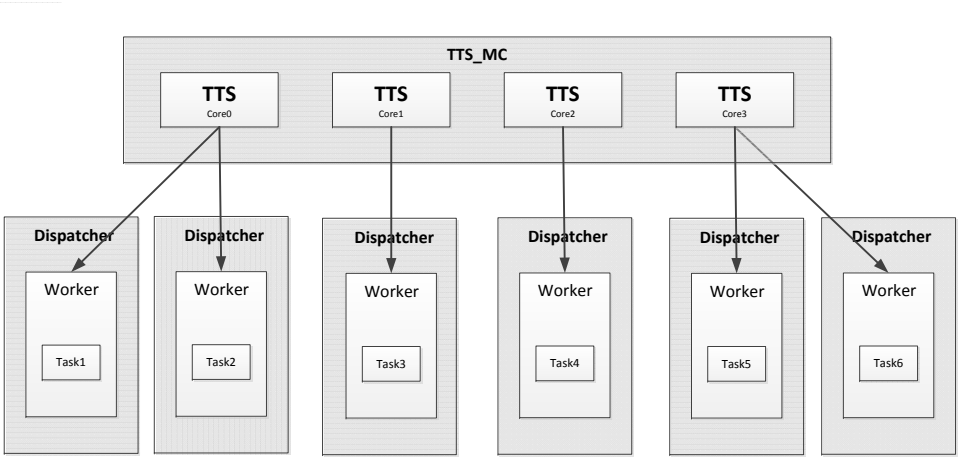


Figure 5.8.: Structure of Time Triggered Scheduler with Mode Change

Algorithm 2: Time Triggered Scheduler

Initialize $event\_sem\_$, $active\_queue\_$;
while *scheduling not finish* **do**
  $sem\_wait(event\_sem\_)$, wait here until the scheduler is activated;
  if $event\_sem\_$ *is overrun of a Worker i* **then**
    if *The task in Worker i is low critical* **then**
      cancel execution of task in Worker i;
      notify $master$ to cancel all executing low critical tasks in other cores;
    end
    get $current\_criticality\_level$ from TTS-MC;
    if $current\_criticality\_level$ *is LO* **then**
      Notify the $master$ to set the $current\_criticality\_level$ to HI;
    end
  end
  if $event\_sem\_$ *is finish of a job in Worker i* **then**
    delete the Worker i from $active\_queue\_$;
  end
  if $event\_sem\_$ *is arrival of a new job* **then**
    insert the Worker i into $active\_queue\_$;
    if *new Cycle begins* && $current\_criticality\_level$ *is HI* **then**
      notify the master to set the $current\_criticality\_level$ to LO;
    end
  end
  if $active\_queue\_$ *is not empty* **then**
    pop a $Worker$ from $active\_queue\_$;
    activate the $Worker$;
  end
end

Algorithm 3: Worker of TTS-MC

```
while scheduling not finish do
    wait here until a job release from dispatcher;
    if a job has released then
        Notify the scheduler with new job arrival;
        active the scheduler;
        wait here until the scheduler activate this worker;
        if if the worker is activated by scheduler then
            start execution of task;
            Set a overrun checker, if overruns, active scheduler and notify an
              overrun event;
        end
    end
end
```

pos indicates the position of dispatcher time in a cycle, it could be Front or Back. $worker$ is the worker related to this dispatcher. The meaning of other variables could be found in figure 5.9.

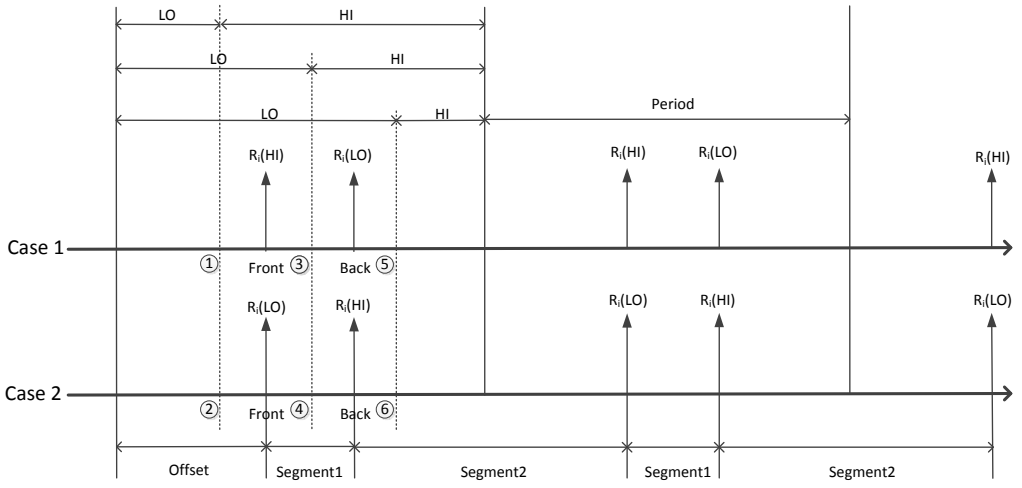


Figure 5.9.: Mode change illustration

Algorithm 4: Dispatcher of high critical task in TTS-MC

if $R_i(LO) > R_i(HI)$ **then**
  $offset \leftarrow R_i(HI)$;
  $Segment1 \leftarrow R_i(LO) - R_i(HI)$;
  $Segment2 \leftarrow Period - Segment1$;
else
  $offset \leftarrow R_i(LO)$;
  $Segment1 \leftarrow R_i(HI) - R_i(LO)$;
  $Segment2 \leftarrow Period - Segment1$;
end
$release\_time \leftarrow offset$;
$clock\_nanosleep(HSF\_CLOCK, TIMER\_ABSTIME, \&release\_time, nullptr)$;
$pos \leftarrow Front$;
$flag \leftarrow false$ indicates if the task has been triggered in this period;
while *scheduler active* **do**
  if $R_i(LO) > R_i(HI)$ **then**
    DispatchCase1();
  end
  if $R_i(LO) < R_i(HI)$ **then**
    DispatchCase2();
  end
  if $R_i(LO) == R_i(HI)$ **then**
    $worker \rightarrow newJob()$;
    $release\_time+ = Period$;
    $clock\_nanosleep(HSF\_CLOCK, TIMER\_ABSTIME, \&release\_time, nullptr)$;
  end
end

Algorithm 5: DispatchCase1()

```
if pos == Front then
    if current_criticality_level == HI then
        worker → newJob();
    else
        flag ← true;
    end
    release_time+ = segment1;
    clock_nanosleep(HSF_CLOCK, TIMER_ABSTIME, &release_time, nullptr);

    pos ← Back;
end
if pos == Back then
    if current_criticality_level == LO then
        worker → newJob();
    else
        if flag is true then
            worker → newJob();
        end
    end
    release_time+ = segment2;
    pos ← Front;
    flag ← false;
    clock_nanosleep(HSF_CLOCK, TIMER_ABSTIME, &release_time, nullptr);

end
```

Algorithm 6: DispatchCase2()

```
if pos == Front then
    if current_criticality_level == LO then
        worker → newJob();
    else
        flag ← true;
    end
    release_time+ = segment1;
    clock_nanosleep(HSF_CLOCK, TIMER_ABSTIME, &release_time, nullptr);

    pos ← Back;
end
if pos == Back then
    if current_criticality_level == HI then
        if flag is false then
            worker → newJob();
        end
    end
    release_time+ = segment2;
    pos ← Front;
    flag ← false;
    clock_nanosleep(HSF_CLOCK, TIMER_ABSTIME, &release_time, nullptr);

end
```

5.2.1. Parser Extension and XML sample

The parser extension of TTS-MC is a function $TTS\_MC < T > *Parser :: parseTTS\_MC($ $xml\_node\ TTS\_MC\_node,\ unsignedint*\ id, int\ level)$. A Sample XML file for a TTS-MC scheduler according to figure 4.5 shows in Listing 5.1.

Listing 5.1: XMLexamplettsmc

```
<simulation name="TTS_MC example">
<duration value="1" units="sec" />
<runnable type="scheduler" algorithm="TTS_MC" cores="2"
 criticality_levels="2">

<runnable type="worker" periodicity="periodic"
task="busy_wait" monitoring="true">
<offset value="0" units="ms" />
<offset_hi value="0" units="ms" />
<name value="T1" />
<core value="1" />
<period value="80" units="ms" />
<criticality_level value="2" />
```

```
<wcet criticality_level="1" value="25" units="ms" />
<wcet criticality_level="2" value="40" units="ms" />
<distribution value="uniform"/>
<relative_deadline value="80" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic"
task="busy_wait" monitoring="true">
<offset value="30" units="ms" />
<name value="T2" />
<core value="1" />
<period value="80" units="ms" />
<criticality_level value="1" />
<wcet criticality_level="1" value="25" units="ms" />
<distribution value="uniform"/>
<relative_deadline value="80" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic"
task="busy_wait" monitoring="true">
<offset value="30" units="ms" />
<name value="T3" />
<core value="2" />
<period value="80" units="ms" />
<criticality_level value="1" />
<wcet criticality_level="1" value="25" units="ms" />
<distribution value="uniform"/>
<relative_deadline value="80" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic"
task="busy_wait" monitoring="true">
<offset value="65" units="ms" />
<offset_hi value="50" units="ms" />
<name value="T4" />
<core value="1" />
<period value="80" units="ms" />
<criticality_level value="2" />
<wcet criticality_level="1" value="15" units="ms" />
<wcet criticality_level="2" value="30" units="ms" />
<distribution value="uniform"/>
<relative_deadline value="80" units="ms" />
</runnable>

</runnable>
</simulation>
```

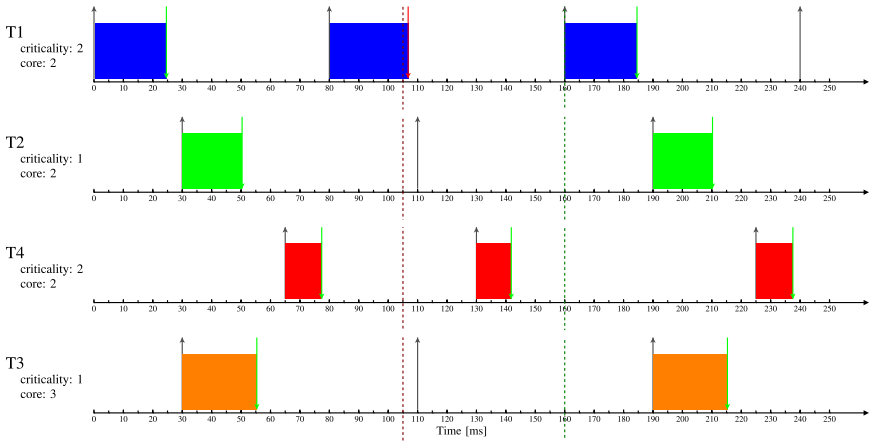


Figure 5.10.: Output of a TTS-MC scheduling example

5.2.2. Simulation Output and Visualization

The figure 5.10 shows the output and visualization after scheduling of the XML in list 5.1. An overrun of task T1 happened in the second cycle, where the red dotted line indicates the overrun of a task, and green dotted line indicates the time point when criticality mode turns back to LO.

5.3. Event scheduler-MC

The Figure 5.11 shows the structure of the event scheduler-MC. Different from the TTS-MC, it has only one dispatcher for all workers and tasks. All tasks' release are managed by this dispatcher called $EventschedulerPeriodic$. Every core has its own event scheduler which is only in charge of scheduling the tasks assigned on this core. Every event scheduler is actually a FIFO scheduler. But unlike the FIFO-like scheduler, in worker a task needs to wait the event of its precedence tasks finished. Every task has a related task event, which is used to represent the state of this task itself(finished or not). In the Event scheduler-MC there is a semaphore map $map < string, sem\_t > dep\_event$ that is used to represent all the events, e.g. $dep\_event\_[\text{“}LanesPro\text{”}]$ implies that the event whether the task “LanesProc” has finished. It needs to be mentioned that the $sem\_wait(\&event)$ of every task could not be implemented in the scheduler, because the wait operation will block the scheduler thread until the event is posted. Therefore the wait operation should be implemented in the worker thread. Since every worker has its own respon-

sible task $task\_$, the worker could get $task\_$'s dependencies $dep\_task\_names[n]$, where n indicates the total number of the precedence dependencies of $task\_$. An semaphore event $sem\_arrived\_jobs\_$ is used to represent if there is a new job comes to this worker. The main process of worker thread could be represented in algorithm 7. The same with TTS-MC, $current\_criticality\_level$ is the current critical mode(HI or LO) of the event scheduler-MC. Algorithm 8 shows the main process of the dispatcher EventschedulerPeriodic. The EventschedulerPeriodic has all workers' pointers $*workers_{[}N]$, where N indicates the total numbers of tasks also workers. $Period$ is the period of whole task set, $release\_time$ is the release time that calculated by the period we set.

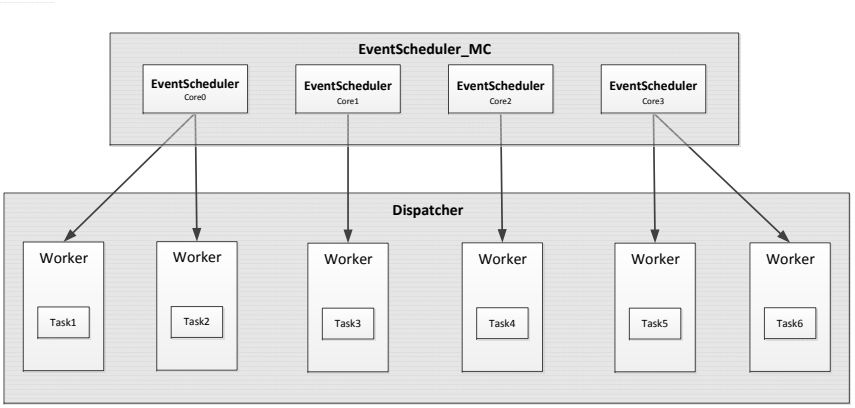


Figure 5.11.: Structure of Event scheduler in Multi-Core

Algorithm 7: worker of Event scheduler-MC

Initialize $dep\_event\_$, dep_task_names, $task\_$;
while *scheduling not finish* **do**
  $sem_wait(\&sem_arrived_jobs)$wait here until a job release from dispatcher, notify the scheduler with job arrival;
  for *i:n* **do**
    $sem\_wait(\&dep\_event\_[dep\_task\_names[i]])$;
    $sem\_post(\&dep\_event\_[dep\_task\_names[i]])$, wait until this task's depedence tasks finish;
  end
  get $current\_ciricality\_level$ from scheduler;
  if $task\_$ *is LO critical* && $current\_ciricality\_level = HI$ **then**
    cancel the execution of $task\_$;
  else
    start execution of $task\_$;
    set timing monitor of $task\_$, if task overruns, notify the scheduler with overrun event;
  end
end

Algorithm 8: Dispatcher of Event scheduler-MC: EventschedulerPeriodic

Initialized $releas\_time$;
while *scheduling not finish* **do**
  WaitAllEvents(), wait until all tasks finished in the last cycle;
  $workers[1:N]\_ \rightarrow post(\&sem\_arrived\_jobs)$, activate new Jobs on all workers as a predefined order;
  $releas\_time+ = Period$;
  $clock\_nanosleep(HSF\_CLOCK, TIMER\_ABSTIME, \&release\_time, nullptr)$, wait $Period$(ms) until next release;
end

5.3.1. Parser Extension and XML sample

The Parser Extension of Event scheduler in Multi-Core is a function $EventScheduler\_MC < T > *Parser :: parseEventScheduler\_MC(xml\_nodeEventScheduler\_MC\_node, unsignedint* id, intlevel)$. A Sample XML file for the task set according to figure 4.6 shows in Listing 5.3.

Listing 5.2: XML example of Event scheduler in Multi-Core

```
<simulation name="EventScheduler_MC">
<duration value="1" units="sec" />
<runnable type="scheduler" algorithm="EventScheduler_MC"
cores="2" >

<runnable type="worker" periodicity="eventschedulerperiodic"
 task="busy_wait" monitoring="true">
<offset value="0" units="ms" />
<order value="0"/>
<name value="T1" />
<core value="0" />
<period value="80" units="ms" />
<relative_deadline value="80" units="ms" />
</runnable>

<runnable type="worker" periodicity="eventschedulerperiodic"
task="busy_wait" monitoring="true">
<dependency value="T1" />
<order value="1"/>
<name value="T2" />
<core value="0" />
<period value="80" units="ms" />
<relative_deadline value="80" units="ms" />
</runnable>

<runnable type="worker" periodicity="eventschedulerperiodic"
task="busy_wait" monitoring="true">
<dependency value="T1" />
<order value="0"/>
```

```
<name value="T3" />
<core value="1" />
<period value="80" units="ms" />
<relative_deadline value="80" units="ms" />
</runnable>

<runnable type="worker" periodicity="eventschedulerperiodic"
task="busy_wait" monitoring="true">
<dependency value="T2" />
<dependency value="T3" />
<order value="0"/>
<name value="T3" />
<core value="1" />
<period value="80" units="ms" />
<relative_deadline value="80" units="ms" />
</runnable>

</runnable>
</simulation>
```

5.3.2. Simulation Output and Visualization

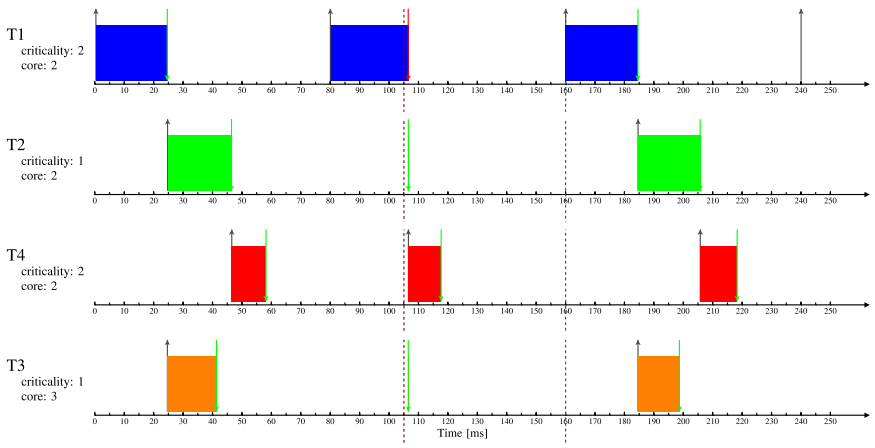


Figure 5.12.: Output of a event scheduler scheduling example

The figure 5.12 shows the output and visualization of event scheduler-MC, where red dotted line indicates the overrun of a task, and green dotted line indicates the time point when criticality mode turns back to *LO*.

5.4. Real task set support

In order to support a real task set, several modifications have been made on the Task class. All the tasks related to path tracking(lane processing) functions have been integrated into HSF. The derive relation shows in figure 5.13. Task should be written as a class derived from the class Task. The most important methods of real task class is the $initialize()$ and $fire()$ function. $initialize()$ will be called after the class is created. $fire()$ is the real execution codes of the task. In order to achieve cancellation, the code in $fire()$ should be restructured as mentioned in section 5.2.1 if it could.

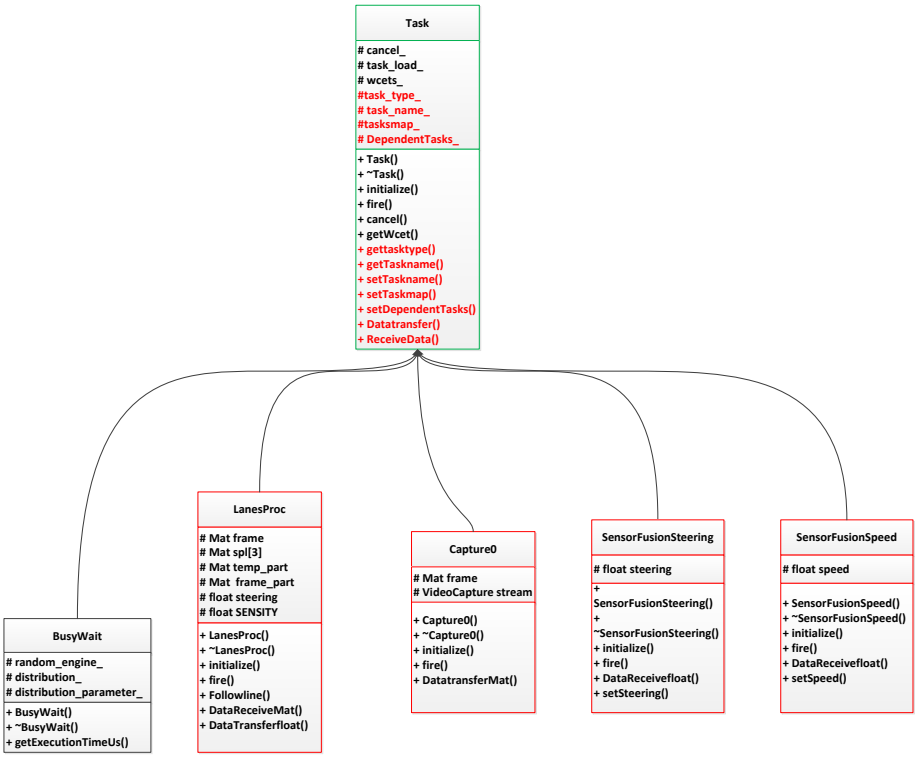


Figure 5.13.: Real task classes

Methods $Datatransfer()$ and $DataReceive()$ are used to achieve the function of data transfer between tasks. Each task has a whole task map $tasksmap\_$, which stores all other tasks' pointers. When $Datatransfer(target, data)$ is called, it actually calls $tasksmap\_[target] \rightarrow DateReceive(data)$.

5.5. Time Measurement

In the original HSF time is measured by using the x86-architecture-specific time stamp counter (TSC). Another way to measure time is using the system call interface ($clock\_gettime()$ function). The advantage of using TSC is that it accesses the timer with only one single CPU instruction and has an accuracy of one CPU cyle. Unfortunately it is not possible

to use TSC in Raspberry Pi due to its ARM structure. Therefore in our case the time is measured with the system call interface $clock\_gettime()$.

6. Integration of software modules

The integration of software modules is to integrate all tasks with the Hierarchical Scheduling Framework. As far only the tasks related to functionality path tracking (lane processing) are integrated. The functionality development of path tracking will be introduced since it is also a contribute of this thesis.

6.1. Path tracking

The objective of path tracking (also called lane processing) is that the car should keep its track in the middle of a road with clear board. A road shows in figure 6.1 and 6.2 is chosen as a test environment. The pictures are taken from one of the stereo cameras of the model car. Therefore the size of picture is only 320*240, which is enough for lane detection. Photos from this camera is the input of path tracking and steering angle of servos will be sent to micro-controller Atmega328 as output. All of the following steps are achieved with help of OpenCV libraries.

Figure 6.1.: Test road1

Figure 6.2.: Test road2

Since the road has high contrast between the road surface(grey) and lawns(green) on both side. It is possible to extract road contour with thresholding. But it is also observed that, shadows ,sunshine, fallen leaves as well as small rocks on the road will be big interferences for our algorithm. After anaylsis of the photos from camera, the following 5 steps are made as the process of lane processing.

(1) Get the bottom part of a picture. Not the whole picture is useful for lane detection. Since the camera is parallel to the ground, the road occupies only about 1/4 part of the

whole photo with about 15 meters' distance. Small size also makes the image processing much more faster.

Figure 6.3.: Get bottom part of a picture

(2) Split the RGB channels of a picture A RGB based picture consists of 3 channels, the red, green and blue channel. Since in our case the back ground of the bottom part is usually green while the road surface is grey, not every channel of the RGB photo has a good contrast. The figure 6.4 to 6.7 show the greyscales of three channels and the result after ostu(will be introduced in (4)) thresholding of a same picture. It is observed that the blue channel has the best contrast and result after thresholding.

Figure 6.4.: Original greyscale

Figure 6.5.: Blue channel

Figure 6.6.: Green channel

Figure 6.7.: Red channel

(3) Gaussian blur to filter noise A Gaussian filter is used to smooth the greyscale of a picture. It is effective to remove noises. The following pictures show the differences

between the results of thresholding with a GaussianBlur and without. The comparision shows that thresholding after Gaussian blur could reduce noises significantly.

Figure 6.8.: Greyscale of blue channel

Figure 6.9.: Greyscale after gaussian blur

Figure 6.10.: After ostu thresholding

Figure 6.11.: After ostu thresholding

(4) Thresholding with Otsu's binarization In global thresholding of a picture, an arbitrary value is used as a threshold value. But it is hard to know if the value we chosen is good or not. Since in our case the bottom part is mostly made of two different colors, it is a typical bimodal image, whose histogram has two peaks. e.g. the histogram of figure 6.9 shows as follows. The histogram is got by using the function $calcHist()$ provided by OpenCV. The Otsu's binarization tries to find the minimal point between two peaks and this value will be regarded as the optimal threshold value. We could say in figure 6.12 the blue point is the best value for thresholding.

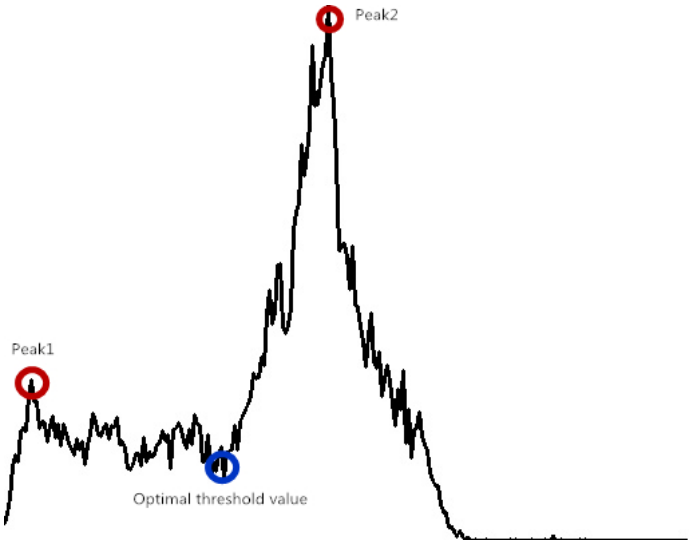


Figure 6.12.: Histogram of figure 6.9

(5) Steering control with pure pursuit method Figure 6.13 shows the camera view and the top view of a road situation. The goal point, also the destination point is determined

by calculating the middle point of the red part of road contour, e.g. the green point is the goal point.

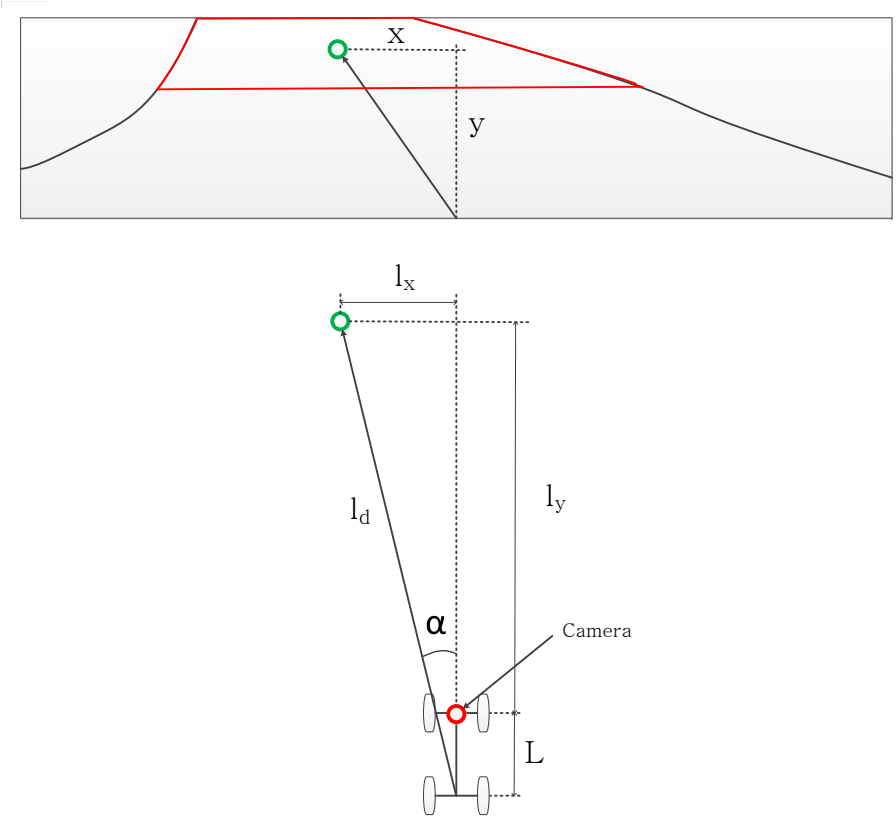


Figure 6.13.: Camera view and top view

From the figure 6.14 we could get the geometric relationship between the goal point position in photo from camera and the real goal point in top view.

$$\begin{cases} l_x = k_x \times x \\ l_y = k_y \times y \\ l_d = \sqrt{l_x^2 + (l_y + L)^2} \\ sin\,(\alpha) = l_x / l_d \end{cases} \tag{6.1}$$

where x, y is the number of pixels in camera view, l_x, l_y are the real distance in top view, k_x, k_y are proportional ratios between the number of pixels and real distance. l_d is the distance from the current rear axle position to the real goal point. L is the length between the front axle to rear axle fo the car.

The model car could be simplified as a bicycle model with two wheels: the front steering wheel and the rear fixed wheel. We assume that the car could only move on a plane. From the figure 6.14 we could get the geometric relationship shows in equation 6.2 between the wheel steering and the curvature the rear axle will follow.

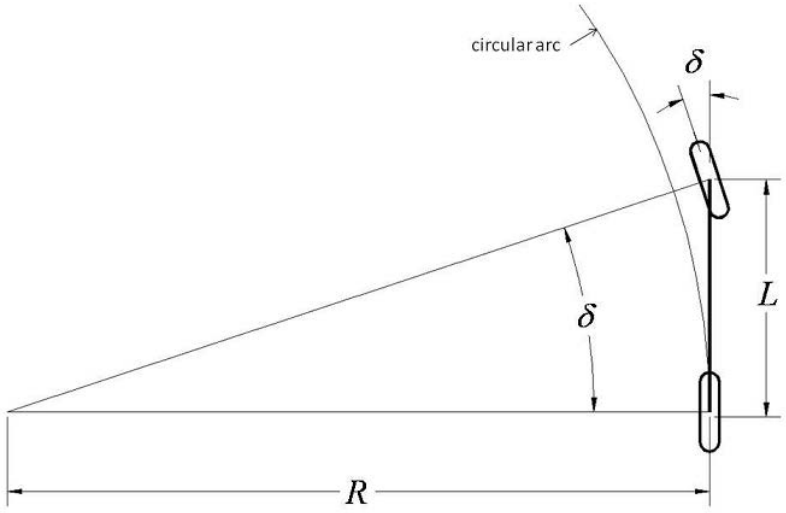


Figure 6.14.: Geometric movement model[33]

$$tan\,(\delta) = L/R \tag{6.2}$$

where δ is the steering angle of the front wheel, R is the radius of the circle that the rear axle will travel along at the given steering angle.

The pure pursuit [33] method is one of the most common approaches to the path tracking problem for mobile robots. The key part of the pure pursuit method is the calculation of the curvature of a circular, which connects the rear axle location to a goal point. Figure 6.15 shows the pure pursuit geometry.

$$\frac{l_d}{sin(2\alpha)} = \frac{R}{sin\,(\frac{\pi}{2} - \alpha)} \Rightarrow \frac{l_d}{2sin\,(\alpha)\,cos\,(\alpha)} = \frac{R}{cos\,(\alpha)} \tag{6.3}$$

$$R = \frac{l_d}{2sin\,(\alpha)} \tag{6.4}$$

From the equation 6.1, 6.2, 6.4 we could get the steering angle:

$$\delta = arctan\left(\frac{2k_x \cdot L \cdot x}{k_x^2 \cdot x^2 + (k_y y + L)^2}\right) \tag{6.5}$$

According to our test the value $k_x \approx 0.025\ m/pixel, k_y \approx 0.2\ m/pixel, y = 50\ pixels, L = 0.8m.$ we could get:

$$\delta = arctan\left(\frac{0.04 \cdot x}{0.000625 \cdot x^2 + 116.64}\right) \tag{6.6}$$

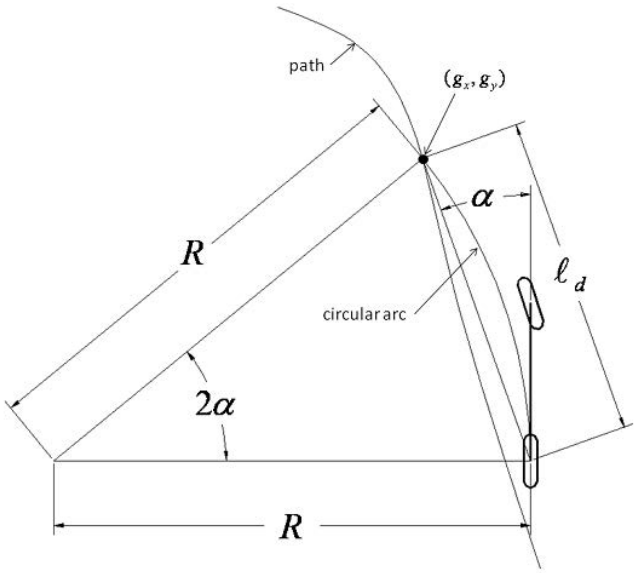


Figure 6.15.: Pure Pursuit geometry[33]

6.2. integration of tasks

For TTS-MC two task allocations in HI-critical mode and LO-critical mode are needed. In section 3 we have presented two solutions to solve the task allocation problem to minimize the makespan. And the task allocation for LO-critical mode has already been determined. In order to get the task allocation in high critical mode, the first thing needs to do is the classification of criticality of tasks.

Classification of tasks' criticality level According to the standard ISO 26262, the criticality levels in automotive industry are defined as the name Automotive Safety Integrity Level(ASIL). There are four ASILs identified by the standard: ASIL A, ASIL B, ASIL C, ASIL D. ASIL D dictates the highest integrity requirements on the product and ASIL A the lowest. QM is an additional level which has no safety requirements. In ASIL the classification of a taskis defined by three scales:
Severity scale (S)

- **S0** No Injuries
- **S1** Light to moderate injuries
- **S2** Severe to life-threatening (survival probable) injuries
- **S3** Life-threatening (survival uncertain) to fatal injuries

Exposure scale (E)

- **E0** Incredibly unlikely

- **E1** Very low probability (injury could happen only in rare operating conditions) injuries
- **E2** Low probability
- **E3** Medium probability
- **E4** High probability (injury could happen under most operating conditions)

Controllability scale (C):

- **C0** Controllable in general
- **C1** Simply controllable
- **C2** Normally controllable (most drivers could act to prevent injury)
- **C3** Difficult to control or uncontrollable

The table 6.1 shows the ASIL classification of tasks according to its severity, exposure and controllability scales.

		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Table 6.1.: ASIL Assignment Matrix

According to the table 6.1 we could get the following task criticality classification in table 6.2 of our system. the controllability scale of all functionalities are C3 since the car is autonomous driving, no driver is involved. Because ASIL QM has no safety requirement, the functionalities traffic signs processing, traffic lights processing and navigation are set as LO-critical, while path tracking, collision avoidance, and steering as well as speed control are set as HI-critical.

From the classification in table 6.2 we could get the HI-critical task set $\tau_{\chi_i \in HI} = \{T4, T5, T6, T7, T9, T10\}$. And its task graph in figure 6.16.

Task allocation in high critical mode According to the figure 3.2 the period of the task set is 116 milliseconds. In order to keep the period after mode change same with in low critical mode, the high critical tasks could be assigned with larger budget execution time

Functionality	S	E	C	ASIL	Critical level
Collision avoidance	S3	E4	C3	D	HI
Signs processing	S1	E1	C3	QM	LO
Path tracking	S3	E3	C3	C	HI
Navigation	S1	E1	C3	QM	LO
Steering and speed control	S3	E4	C3	D	HI

Table 6.2.: Criticality classification

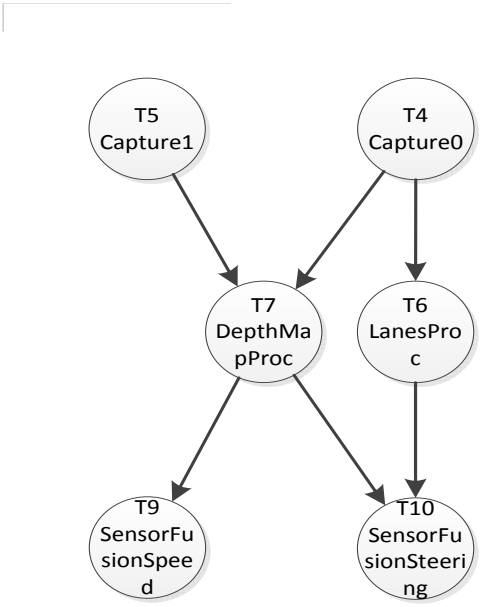


Figure 6.16.: task graph in high critical mode

as its $C_i(HI)$. The WCET assignment for high critical mode is flexible. The principle is to give the important tasks more budget time for execution. On this purpose we could first delete all low critical tasks in the task allocation. Then adjust the length of the high critical tasks. The adjusted length could be regarded as their $C_i(HI)$. The figure 6.17 and table 6.3 show the task allocation of the whole system.

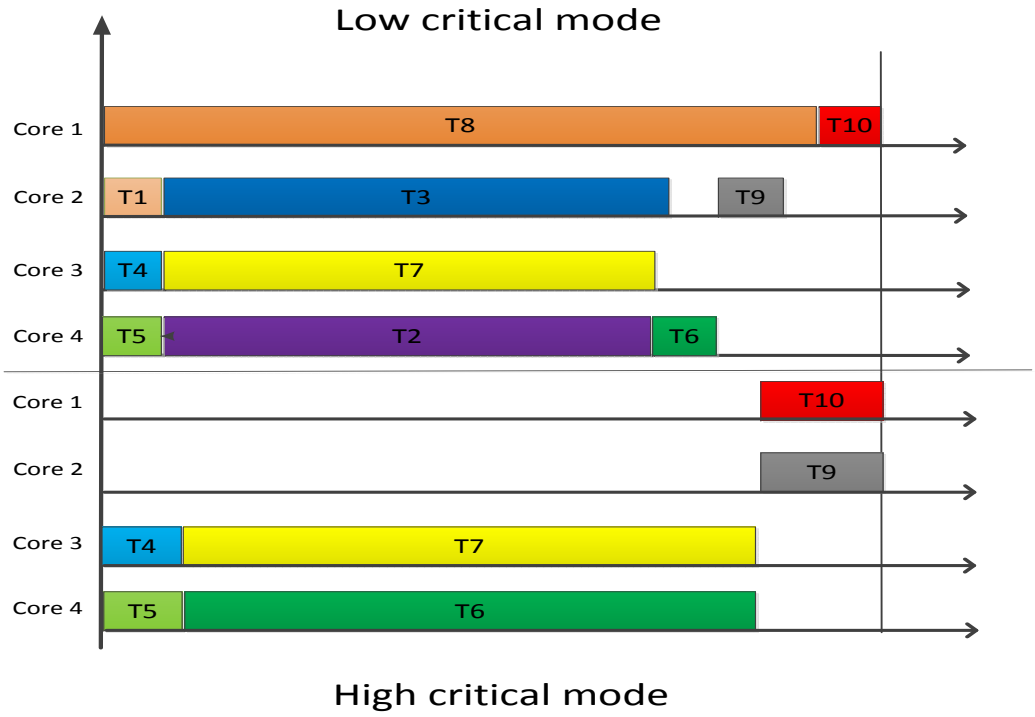


Figure 6.17.: task allocation of autonomous car

Task	name	Core Index	$\overrightarrow{C}(LO, HI))$	$\overrightarrow{R}(LO, HI))$
T1	Capture2	2	9	0
T2	SignsProc	4	70	10
T3	LightsProc	2	76	10
T4	Capture0	3	(9,13)	(0,0)
T5	Capture1	4	(9,13)	(0,0)
T6	LanesProc	4	(10,80)	(81,14)
T7	DepthMapProc	3	(72,80)	(10,14)
T8	GPSProc	1	106	0
T9	SensorFusionSpeed	2	(10,20)	(96,96)
T10	SensorFusionSteering	1	(10,20)	(106,96)

Table 6.3.: task allocation of autonomous car

7. Experiments

In experiments the whole task graph is scheduled with both TTS-MC and event scheduler-MC in Raspberry Pi 3 model B with a 1.2GHz 64-bit quad-core ARMv8CPU. The test time is one hour. The Linux version is raspbian 4.4.13. Scheduling overhead, number of missed deadlines, number of overruns as well as number of canceled tasks will be saved for statistical analysis. Tasks related to path tracking are applied with real tasks. Others are using busy-wait for simulation. The reason is that functionalities such as navigation, traffic signs processing need specific libraries, which are hard to integrate with HSF so far because the HSF is not using a CmakeLists file for compilation. The period of task set is set as all tasks' relative deadline, $D_i = T$. The execution time of busy waiting tasks are set as uniform distribution. The upper bound of uniform distribution is greater than $C_i(LO)$, so that tasks might overrun and the behaviour of scheduler after overrun could be studied.

$$\begin{cases} \chi_i \in LO,\ p_i = (C_i(LO)/2, C_i(LO) + (C_i(HI) - C_i(LO))\ /5) \\ \chi_i \in HI,\ p_i = (C_i(LO)/2, C_i(LO) \times 1.1) \end{cases} \tag{7.1}$$

Scheduling overhead is usually used to evaluate the performance of a scheduler. The scheduling overhead of both TTS-MC and event scheduler-MC consist of the following components: (1) job handling including the handle of job arrival and finishing. (2) overrun handling, including overrun trigger and overrun handle. (3) scheduler state updating. (4) The switch of critical mode. (5) Other processing in wrapper()(main process of scheduler). (6) The cancellation of overrun tasks. (7) Overrun checking since it is running independent as thread. The measurement of overhead is implemented by the Stopwatch Class with end-to-end measurement of code.

7.1. Evaluation of TTS-MC

According to the statistics, there is no deadline misses during scheduling. Table 7.1 shows the details of different parts of the scheduling overhead. The relative overhead in individual core is defined as $relative\ overhead = \frac{absolute\ overhead}{duration}$. The absolute overhead of all cores is 9683.58ms. We could get $relative\ overhead = \frac{absolute\ overhead}{running\ duration \times m} = 0.067\%$, where m is the number of cores.

Figure 7.1 shows the overrun and cancellation rate of each core. The overrun rates are around 1% to 2%. There is no big difference between different cores since every task has the possibility to overrun. Different from the overrun rate, the cancellation rate varies

Components	Core 1	Core 2	Core 3	Core 4	total
Job handling	338.328	510.768	356.076	498.6	1703.772
overrun handling	24.552	21.828	32.94	35.808	115.128
scheduler state updating	3.312	0.192	0.228	0.144	3.876
mode switch	84.324	125.028	164.4	152.292	526.044
task cancellation	115.752	107.076	3.504	117.72	344.052
overrunchecker	887.424	1200.036	970.188	1303.296	4360.944
wrapper	534.816	783.672	530.304	780.972	2629.764
absolute overhead	1988.508	2748.6	2057.64	2888.832	9683.58
relative overhead	0.055%	0.076%	0.057%	0.080%	0.067%

Table 7.1.: Overhead measurement of TTS-MC(ms)

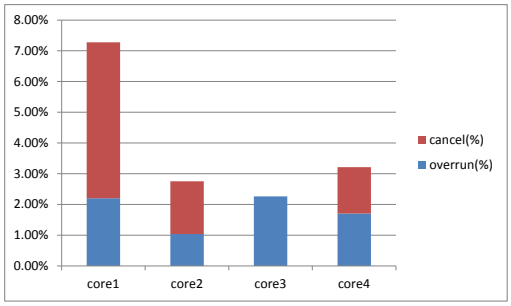


Figure 7.1.: Overrun and cancellation rate of TTS-MC

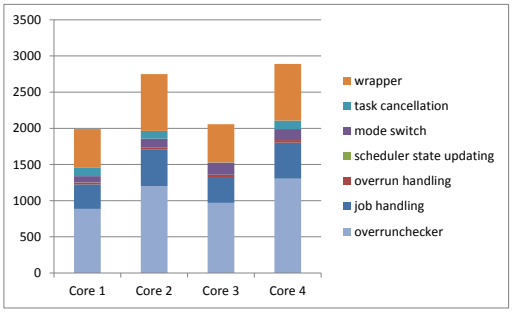


Figure 7.2.: Overhead distribution of TTS-MC

from different cores significantly. In core 1 there are more than 5% tasks are canceled while in core 3 no tasks are canceled. The task cancellation rate is related to the task allocation. Since in core 3, all of tasks are high critical, no task will be canceled. But in core 1, the task "GPSProc" with LO-criticality level has the longest WCET. Overrun of any task might lead to the cancellation of task "GPSProc".

The figure 7.2 and figure 7.3 to 7.6 show the scheduling overhead distribution of different components. The absolute overhead varies from cores. Core 2 and core 4 have greater overhead than core 1 and core 3. It could be concluded that cores with more assigned tasks have greater absolute overhead since core 2 and core 4 have three assigned tasks while core 1 and core 3 have only two. It could be observed that the overheads of overrun checker, Job handling and wrapper are related to the number of assigned tasks. They occupy more than 60% of the scheduling overhead. The scheduling overhead of mode switch, task cancellation and overrun handling are related to the overrun and cancellation rate of tasks. Although these parts of overheads are unpredictable, they occupy only a small part of overhead. But with increase of cancellation and overrun rates, this part of overhead could also increase. In conclusion we could say that the mode change mechanism of TTS-MC has small impact on the performance of scheduler. The relative overhead is totally acceptable. With no deadline miss the TTS-MC scheduler could be used for real-time scheduling.

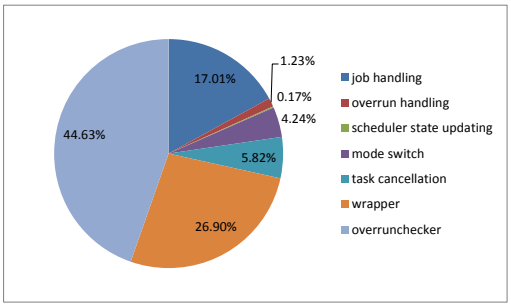


Figure 7.3.: Scheduling overhead distribution statistics in core 1

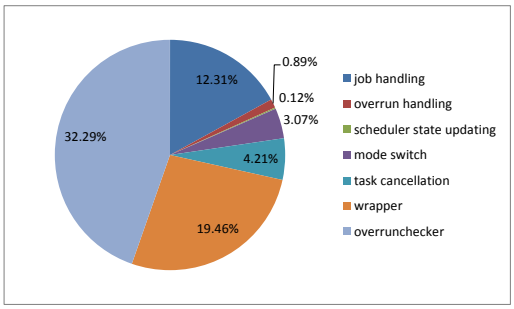


Figure 7.4.: Scheduling overhead distribution statistics in core 2

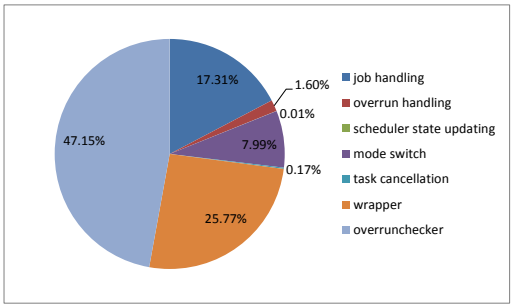


Figure 7.5.: Scheduling overhead distribution statistics in core 3

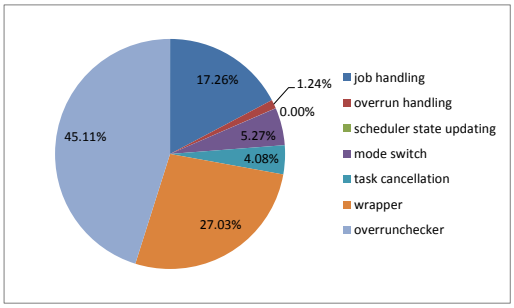


Figure 7.6.: Scheduling overhead distribution statistics in core 4

7.2. Evaluation of Event scheduler-MC

According to the statistics, there is also no deadline miss during scheduling. Table 7.2 shows the details of different parts of the scheduling overhead. The absolute overhead in all of cores is 8292.758ms, and the relative overhead is 0.058%. The overhead of event scheduler-MC is smaller than the TTS-MC. A main reason is that there is only one dispatcher in event scheduler while in TTS-MC every task has a related dispatcher.

Figure 7.7 shows the overrun and cancellation rate of each core. The overrun rates are around 1% to 2%. Same with TTS-MC, the core 1 has the highest cancellation rate. The figure 7.8 and figure 7.9 to 7.12 show the scheduling overhead distribution of different components. Overheads of overrun checker, job handling and wrapper occupy the biggest part, while the scheduling overhead of mode switch, task cancellation and overrun handling occupy only a small part. In conclusion the event scheduler has relative lower task overrun and cancellation rate also less scheduling overhead than the TTS-MC.

Components	Core 1	Core 2	Core 3	Core 4	total
Job handling	208.216	289.183	211.644	424.350	1133.394
overrun handling	9.338401	23.122	34.889	13.134	80.486
scheduler state updating	1.202	2.757	0.720	0.964	5.645
mode switch	85.850	103.873	87.394	125.860	402.979
task cancellation	175.4237	187.535	13.562	93.299	469.821
overrunchecker	790.167	1320.409	690.171	1139.203	3939.951
wrapper	485.725	642.123	369.142	763.488	2260.48
absolute overhead	1755.924	2569.005	1407.527	2560.302	8292.758
relative overhead	0.049%	0.071%	0.039%	0.071%	0.058%

Table 7.2.: Overhead measurement of Event scheduler-MC(ms)

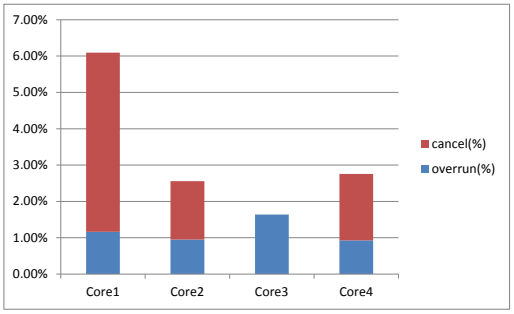


Figure 7.7.: Overrun and cancellation rate of Event scheduler-MC

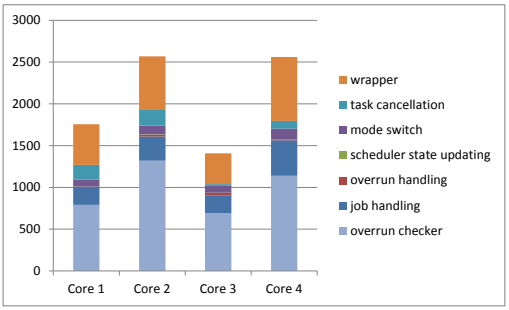


Figure 7.8.: Overhead distribution of Event scheduler-MC

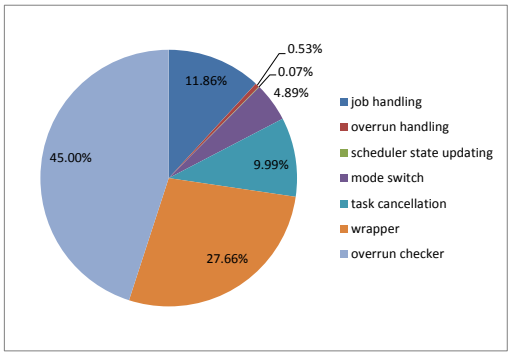


Figure 7.9.: Scheduling overhead distribution statistics in core 1

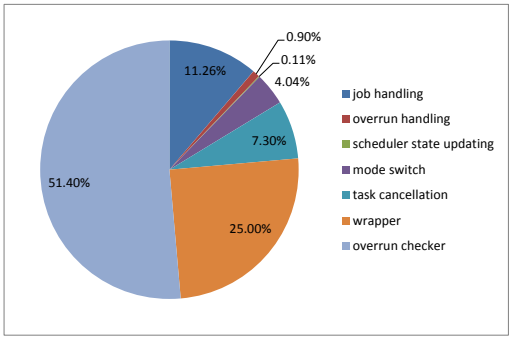


Figure 7.10.: Scheduling overhead distribution statistics in core 2

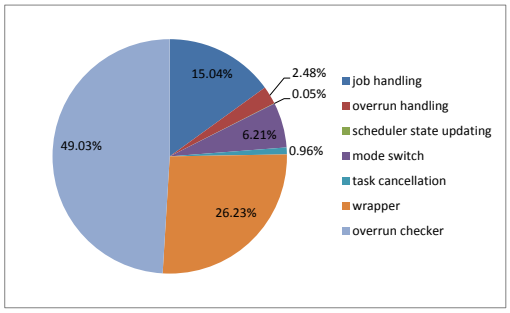


Figure 7.11.: Scheduling overhead distribution statistics in core 3

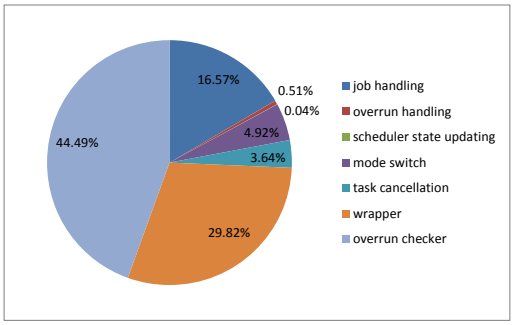


Figure 7.12.: Scheduling overhead distribution statistics in core 4

8. Conclusion and problems

8.1. Conclusion

In this thesis two real-time multi-core schedulers for mixed-criticality task graph with precedence constrained are designed and implemented. The implementation was done by extending the Hierarchical Scheduling Framework(HSF)[25][16]. Both software and hardware of an autonomous driving model car were designed and implemented. The two designed schedulers named ”TTS-MC” and ”Event scheduler-MC” were tested by integrating the task graph of this car with its path tracking functionality. The result showed that both of the two schedulers have good performance with less than 0.1% scheduling overhead. And the real-time requirement of safety-critical tasks could be guaranteed. Compared to ”TTS-MC”, the ”Event scheduler-MC” has lower scheduling overhead and usually shorter response time.

After all, the procedure of applying a real-life task graph with HSF could be made:

- (1) Draw a task graph of the real-life system.
- (2) Classification of the Automotive Safety Integrity Level(ASIL) and get criticality level of tasks(HI or LO).
- (3) Measure the Worst Case Execution Time(WCET) of each task, where the LO-critical tasks, which could not be canceled, should be assigned with more pessimistic WCET.
- (4) Using the TORSCHE scheduling toolbox to get optimal task allocation with minimized makespan.
- (5) Create task class in HSF for each real-life task.
- (6) Write XML file according to task allocation.
- (7) Test the scheduler, adjust the task allocation or task set period if the scheduler works not well.

8.2. Problems

Although the designed schedulers work as wished in my case, there are still problems. I list some tips that may help the people who want to extend this framework with other functionalities.

- (1) Time measurement mechanism of HSF is different in x86 platform from in ARM based platform such as raspberry pi. In x86 platform the time stamp counter is accessed to get a global timing with an accuracy of one CPU cycle, while in ARM based platform we could only access the operating system timer with an accuracy of several microseconds. The disadvantage of this way is that there are some blocking delays when one core tries to access the timer, while another is already inside the system call[25]. Therefore the time measurement of scheduling overhead might have errors, which should be considered in future work.
- (2) The periodic dispatcher of both TTS-MC and Event scheduler-MC is some times delayed in about 10 ms, the reason caused this phenomena remains unknown.
- (3) Real-tasks are usually not be able to be canceled. Pessimistic WCET assignment will increase the makespan significantly. Therefore the cancellation of tasks needs more works.
- (4) In event scheduler-MC, every miss of task finish will block the whole system. In experiment this situation also happened sometime. So far the reason still remains unknown. Therefore using event scheduler-MC to scheduling real-life task set still needs a lot of tests.
- (5) The task allocation get from Torsche toolbox did not consider the criticality of tasks. In practice high critical tasks usually should be allocated earlier than low critical tasks. Also the high critical task load balance should be considered since after the mode change only high critical tasks will be scheduled.

Appendix

A. Installation of Hierarchical Scheduling Framework

The whole project locates in gitlab address: git@gitlab.lrz.de:autonomous-car-ss16/Collision-Avoidance.git, you could find both the original version of HSF and the modified version. For installation please check the README file.

Listing A.1: Installation of Hierarchical Scheduling Framework

```
Hierarchical Scheduling Framework {#mainpage}
================================


Requirements
------------

HSF requires a *NIX kernel with standard libraries. To compile all
    sources,
these packages are needed:

* g++ (>= 4.7)
* make
* octave
* php
* (Lu Cheng)OpenCV (>=3.1.0) (only needed when you are applying with
    the autonomous car task set)

If you want to use the initial support for the hwloc library for
    setting the
processor affinities of the threads, the hwloc tools and library are
    also needed:

* hwloc

You can also directly link the sources without compiling and installing
     them. For
more details see: <http://www.open-mpi.org/projects/hwloc/>\n
To compile hsf without hwloc support, check that `USE_LINUX_AFFINITY`
    is set to
`1` in `src/pthread/Thread.cpp` and uncomment `HSFLIBS += -lhwloc` in
    the Makefile.
```

```
For generating the output figures with the ‘simfig‘ tool you need the
    following
additional libraries:

* libmgl-dev (>= 2.0)
* libX11-dev

Mathgl library should be compiled, and libmgl.so.7.0.0 should be placed
     in
‘/usr/local/lib‘ (otherwise the MATHGL variable in the makefile
should be changed to the appropriate location). If you follow the
    compile
instructions of Mathgl this should happen automatically. For more
    information
on Mathgl, please visit: <http://mathgl.sourceforge.net/>

For building the documentation of the Hierarchical Scheduling Framework
     you will
also need a recent version of doxygen:

* doxygen (>= 1.8)

To build the documentation simply run ‘make doc‘ in the HSF directory.
    The
documentation will then be generated in the ‘doc‘ directory.


Installation and Configuration
------------------------------

1. If your HSF folder is not located in ‘~/git‘, then please change
    line 3 of
‘hsf_paths.sh‘ to the path of your HSF folder.
2. In the terminal, type:

source hsf_paths.sh

This will set a new ‘$HSF‘ variable, and add it to your ‘$PATH‘
    variable. You can
also add it to your ‘~/.bashrc‘ file, to have it load automatically
3. Privileges for executing with real-time priorities
To execute hsf the user needs to have the privileges to switch the
    applications
scheduling policy to real-time priority based scheduling and execute it
     a with
real-time priority. You can do this in two different ways:

1. (RECOMMENDED) Allow the user to execute applications with real-time
    priority
using the limits.conf configuration file. To allow the user with name ’
    user’
```

```
to execute its applications with real-time priority, simply add the
    file
‘/etc/security/limits.d/50-rtprio.conf‘ with the following two lines (
    replace
’user’ with your username):

user   -   rtprio  unlimited
user   -   nice    -20

To apply these changes a reboot of your machine is required.
Please note that this user can execute any application with real-time
    priority.
More details about this can be found in the manpage of limits.conf
2. (_NOT_ RECOMMENDED) To execute hsf with real-time priority the
    command could
simply executed with root privileges for example be using ‘sudo‘.
    However we
do not recommend executing the framework as root.
If you still want to use the sudo approach please note the following:
    On some
older systems, you might have to add the following line to your bash
    profile
in order to inherit you PATH variable when using ’sudo’:

alias sudo=’sudo env PATH=$PATH $@’
3. (Lu Cheng)If you are compiling the HSF in Raspberry pi, the
    following three
files should be replaced to disable the use of TSC: src/util/TimeUtil.h
    , src/util/TimeUtil.cpp, src/util/RDTSC.h.
4. (Lu Cheng)The makefile is already added with OpenCV library, if
    there is any problem, please use the original one.

4. Then type:

./install.sh
5. Run HSF!
You can now type in you terminal the following commands:

hsf [filename(.xml)]
simulate [filename(.xml)]
calculate [metric] [filename]
show [metric] [filename]
simfig [filename]
publish [filename]

[metric] can be one (or more) of the following:

* exe|exec -> Execution Times
* resp     -> Response Times
* throughput -> Throughput
* util     -> Utilization
* alloc    -> Resource allocation costs
```

```
* sys      -> System allocation costs
* worker   -> Worker costs
* missed   -> Missed deadlines


Simulations on Xeon Phi (and Other Systems With Network File Systems)
-----------------------------------------------------------------------

When simulating on Xeon Phi, please notice that the underlaying network
     file system
is likely to be blocked by the HSF framework, because the framework
    runs at higher
priority than the file system daemon. This will cause undefined
    behavior and blocking
of the framework, when it reads and/or writes to files.

To avoid these blocking, you need to copy all files needed for
    simulations to a
local subdirectory in `/tmp/` and start simulation in that directory.

This setup is required if one of the following criteria is met (know
    cases):

- Simalations using the partitioned EDF-VD scheduler (PartitionedEDF_VD
    )
- Simulations with flight management system tasks (FlyanceTask)
```

B. XML file for experiments

B.1. XML file for TTS-MC experiment

Listing B.1: XML file for TTS-MC

```
<simulation name="car_experiment">
<duration value="1" units="sec" />
<runnable type="scheduler" algorithm="TTS_MC" cores="4"
   criticality_levels="2">

<runnable type="worker" periodicity="periodic" task="busy_wait"
   monitoring="true">
<offset value="0" units="ms" />
<name value="Capture2" />
<core value="1" />
<period value="118" units="ms" />
<criticality_level value="1" />
<wcet criticality_level="1" value="9" units="ms" />
<wcet criticality_level="2" value="9" units="ms" />
<distribution value="test"/>
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic" task="busy_wait"
   monitoring="true">
<offset value="10" units="ms" />
<name value="SignsProc" />
<core value="3" />
<period value="118" units="ms" />
<criticality_level value="1" />
<wcet criticality_level="1" value="70" units="ms" />
<wcet criticality_level="2" value="70" units="ms" />
<distribution value="test"/>
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic" task="
   car_SensorFusionSpeed" monitoring="true">
<offset value="105" units="ms" />
<offset_hi value="96" units="ms" />
<name value="SensorFusionSpeed" />
<core value="1" />
<period value="118" units="ms" />
```

```
<criticality_level value="2" />
<wcet criticality_level="1" value="10" units="ms" />
<wcet criticality_level="2" value="20" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic" task="busy_wait"
   monitoring="true">
<offset value="10" units="ms" />
<name value="LightsProc" />
<core value="1" />
<period value="118" units="ms" />
<criticality_level value="1" />
<wcet criticality_level="1" value="76" units="ms" />
<wcet criticality_level="2" value="76" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic" task="busy_wait"
   monitoring="true">
<offset value="10" units="ms" />
<offset_hi value="14" units="ms" />
<name value="DepthMap" />
<core value="2" />
<period value="118" units="ms" />
<criticality_level value="2" />
<wcet criticality_level="1" value="72" units="ms" />
<wcet criticality_level="2" value="80" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic" task="
   car_SensorFusionSteering" monitoring="true">
<offset value="106" units="ms" />
<offset_hi value="96" units="ms" />
<name value="SensorFusionSteering" />
<core value="0" />
<period value="118" units="ms" />
<criticality_level value="2" />
<wcet criticality_level="1" value="10" units="ms" />
<wcet criticality_level="2" value="20" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic" task="busy_wait"
   monitoring="true">
<offset value="0" units="ms" />
<name value="Capture1" />
```

```
<core value="3" />
<period value="118" units="ms" />
<criticality_level value="2" />
<wcet criticality_level="1" value="9" units="ms" />
<wcet criticality_level="2" value="13" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic" task="car_Capture0"
   monitoring="true">
<offset value="0" units="ms" />
<name value="Capture0" />
<core value="2" />
<period value="118" units="ms" />
<criticality_level value="2" />
<wcet criticality_level="1" value="9" units="ms" />
<wcet criticality_level="2" value="13" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic" task="car_LanesProc"
   monitoring="true">
<offset value="81" units="ms" />
<offset_hi value="14" units="ms" />
<name value="LanesProc" />
<core value="3" />
<period value="118" units="ms" />
<criticality_level value="2" />
<wcet criticality_level="1" value="10" units="ms" />
<wcet criticality_level="2" value="80" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="periodic" task="busy_wait"
   monitoring="true">
<offset value="0" units="ms" />
<name value="GPSProc" />
<core value="0" />
<period value="118" units="ms" />
<criticality_level value="1" />
<wcet criticality_level="1" value="105" units="ms" />
<wcet criticality_level="2" value="105" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

</runnable>
</simulation>
```

B.2. XML file for Event scheduler-MC experiment

Listing B.2: XML file for TTS-MC

```
<simulation name="car_experiment_event">
<duration value="2000" units="ms" />
<runnable type="scheduler" algorithm="EventScheduler_MC" cores="4"
   criticality_levels="2">

<runnable type="worker" periodicity="eventschedulerperiodic" task="
   busy_wait" monitoring="true">
<offset value="0" units="ms" />
<name value="GPSProc"/>
<order value="0"/>
<core value="0" />
<period value="118" units="ms" />
<criticality_level value="1" />
<wcet criticality_level="1" value="105" units="ms" />
<wcet criticality_level="2" value="105" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="eventschedulerperiodic" task="
   car_SensorFusionSteering" monitoring="true">
<dependency value="DepthMapProc" />
<dependency value="LanesProc" />
<dependency value="GPSProc" />
<name value="SensorFusionSteering"/>
<order value="1"/>
<core value="0" />
<criticality_level value="2" />
<wcet criticality_level="1" value="10" units="ms" />
<wcet criticality_level="2" value="20" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="eventschedulerperiodic" task="
   busy_wait" monitoring="true">
<offset value="0" units="ms" />
<order value="0"/>
<name value="Capture2" />
<core value="1" />
<criticality_level value="1" />
<wcet criticality_level="1" value="9" units="ms" />
<wcet criticality_level="2" value="9" units="ms" />
<distribution value="test"/>
<relative_deadline value="118" units="ms" />
</runnable>
```

```
<runnable type="worker" periodicity="eventschedulerperiodic" task="
   busy_wait" monitoring="true">
<dependency value="Capture2" />
<name value="LightsProc" />
<order value="1"/>
<core value="1" />
<criticality_level value="1" />
<wcet criticality_level="1" value="76" units="ms" />
<wcet criticality_level="2" value="76" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="eventschedulerperiodic" task="
   car_SensorFusionSpeed" monitoring="true">
<dependency value="SignsProc" />
<dependency value="LightsProc" />
<dependency value="DepthMapProc" />
<order value="2"/>
<name value="SensorFusionSpeed"/>
<core value="1" />
<criticality_level value="2" />
<wcet criticality_level="1" value="10" units="ms" />
<wcet criticality_level="2" value="20" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="eventschedulerperiodic" task="
   car_Capture0" monitoring="true">
<name value="Capture0" />
<order value="0"/>
<core value="2" />

<criticality_level value="2" />
<wcet criticality_level="1" value="20" units="ms" />
<wcet criticality_level="2" value="30" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="eventschedulerperiodic" task="
   busy_wait" monitoring="true">
<dependency value="Capture1" />
<dependency value="Capture0" />
<name value="DepthMapProc" />
<order value="1"/>
<core value="2" />
<criticality_level value="2" />
<wcet criticality_level="1" value="72" units="ms" />
<wcet criticality_level="2" value="80" units="ms" />
<distribution value="test" />
```

```
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="eventschedulerperiodic" task="
   busy_wait" monitoring="true">
<name value="Capture1" />
<order value="0"/>
<core value="3" />
<criticality_level value="2" />
<wcet criticality_level="1" value="9" units="ms" />
<wcet criticality_level="2" value="13" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="eventschedulerperiodic" task="
   busy_wait" monitoring="true">
<dependency value="Capture2" />
<name value="SignsProc" />
<order value="1"/>
<core value="3" />
<criticality_level value="1" />
<wcet criticality_level="1" value="70" units="ms" />
<wcet criticality_level="2" value="70" units="ms" />
<distribution value="test"/>
<relative_deadline value="118" units="ms" />
</runnable>

<runnable type="worker" periodicity="eventschedulerperiodic" task="
   car_LanesProc" monitoring="true">
<dependency value="Capture0" />
<name value="LanesProc" />
<order value="2"/>
<core value="3" />
<criticality_level value="2" />
<wcet criticality_level="1" value="10" units="ms" />
<wcet criticality_level="2" value="15" units="ms" />
<distribution value="test" />
<relative_deadline value="118" units="ms" />
</runnable>

</runnable>
</simulation>
```

Bibliography

[1] Makespan scheduling. https://www2.informatik.hu-berlin.de/alcox/lehre/lvws1011/coalg/makespan_scheduling.pdf.

[2] J.H. Anderson, S.K. Baruah, and B.B. Brandenburg. Multicore operating-system support for mixed criticality. *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification, San Francisco*, 2009.

[3] M. Asberg, T. Nolte, S. Kato, and R. Rajkumar. ExSched: An External CPU Scheduler Framework for Real-Time Systems. *Proc. RTCSA*, pages 240–249, 2012.

[4] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The Preemptive Uniprocessor Scheduling of Mixed-Criticality Implicit-Deadline Sporadic Task Systems. *24th Euromicro Conference on Real-Time Systems*, pages 145–154, July 2012.

[5] G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real-time systems. *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 279–288, 2002.

[6] Enis Bilgin and Stefan Robila. Road sign recognition system on Raspberry Pi. *2016 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pages 1–5, 2016.

[7] M. Bommert. Schedule-aware distributed of parallel load in a mixed criticality environment. *JRWRTC, RTNS*, page 21–24, 2013.

[8] Catalin-Virgil Briciu, Ioan Filip, and Franz Heininger. A new trend in automotive software: AUTOSAR concept. *Applied Computational Intelligence and Informatics (SACI), 2013 IEEE 8th International Symposium*, pages 251–256, 2013.

[9] A. Burns and R.I.Davis. Mixed criticality systems: A review. Technical Report SA-2013-57, Department of Computer Science, University of York, 2014.

[10] J.M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS : A Testbed for Empirically Comparing RealTime Multiprocessor Schedulers. *in Proc. RTSS*, pages 111–126, 2006.

[11] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari. Hierarchical multiprocessor cpu reservations for the linux kernel. *5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009)*, 2009.

[12] R.I. Davis and A. Burns. Hierarchical fixed priority preemptive scheduling. *IEEE Real-Time Systems Symposium (RTSS)*, pages 389–398, 2005.

[13] D. Faggioli, M. Trimarchi, and F. Checconi. An Implementation of the Earliest Deadline First Algorithm in Linux. *in Proc. SAC*, pages 1984–1989, 2009.

[14] G. Farmer and R. West. Hijack: Taking Control of COTS Systems for Real-Time User-Level Services. *Proc. RTAS*, pages 133–146, 2007.

[15] Paolo Gai and Massimo Violante. Automotive embedded software architecture in the multi-core age. *2016 21th IEEE European Test Symposium (ETS)*, pages 1–8, 2016.

[16] Andres Gomez, Lars Schor, Pratyush Kumar, and Lothar Thiele. SF3P: a framework to explore and prototype hierarchical compositions of real-time schedulers. *2014 25nd IEEE International Symposium on Rapid System Prototyping*, pages 2–8, Oct 2014.

[17] Giovani Gracioli. Implementation and evaluation of global and partitioned scheduling in a real-time OS. *Real-Time Systems*, pages 669–714, November 2013.

[18] Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Improving the scheduling of certifiable mixed-criticality sporadic task systems. Technical Report 2013-008, Department of Information Technology, Uppsala University, April 2013.

[19] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH.

[20] K.Lakshmanan, D.de Niz, and G.Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. *In ICDCS*, pages 167–178, 2010.

[21] A. Kritikakou, C. Pagetti, C. Rochange, M. Roy, M. Faugre, S. Girbal, and D.G. Prez. Distributed run-time WCET controller for concurrent critical tasks in mixedcritical systems. *RTNS*, 2014.

[22] Graham. R. L. Bounds for certain multiprocessing anomalies. *Bell Syst. Techn. J.*, 45:1563–1581, 1966.

[23] A. Lackorzynski, A. Warg, and M. Voelp. Flattening hierarchical scheduling. *ACM EMSOFT*, pages 93–102, 2012.

[24] Haohan Li and Sanjoy Baruah. Improving the Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems. pages 166 – 175, July 2012.

[25] Lukas Sigrist. Implementation and evaluation of mixed-criticality scheduling algorithms for multi-core systems. Technical Report SA-2013-57, Swiss Federal Institute of Technology Zurich, 2014.

[26] Alessandra Melani. Global scheduling in multiprocessor real-time systems. `http://retis.sssup.it/~giorgio/slides/cbsd/mc3-global-2p.pdf`. Retrieved on 2015.04.15.

[27] M. Mollison, J. Erickson, J. Anderson, S.K. Baruah, and J. Scoredos. Mixed criticality real-time scheduling for multicore systems. *the 7th IEEE International Conference on Embedded Software and Systems*, page 1864–1871, 2010.

[28] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. An Implementation of the Earliest Deadline First Algorithm in Linux. *Software: Practice and Experience*, 39(1):1–31, 2009.

[29] Roger S Rivett. Hazard identification and classification: ISO26262- the application of IEC61505 to the automotive sector. *SIL Determination, 2009 5th IET Seminar*, pages 1–24, 2009.

[30] Roger Rösch. Development and Construction of an Autonomous Car with Low Cost Components. Master's thesis, Technische Universität München, 2016.

[31] N. R. Satish, K. Ravindran, and K. Keutzer. Scheduling task dependence graphs with variable task execution times onto heterogeneous multiprocessors. Technical Report UCB/EECS-2008-42, EECS Department, University of California, Berkeley, April 2008.

[32] Adam Sherer, John Rose, and Riccardo Oddone. Ensuring functional safety compliance for ISO 26262. *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–3, 2015.

[33] J.M. Snider. Automatic Steering Methods for Autonomous Automobile Path Tracking. *Carnegie Mellon University*, Feb 2009.

[34] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Multiprocessor scheduling of precedence-constrained mixed-critical jobs. Technical Report TR-2014-11, Verimag - Universit´e Joseph Fourier - Grenoble, 2014.

[35] S. Sören, G. Kim, S. Rosinger, and A. Rettberg. utonomous Flight Control Meets Custom Payload Processing: A Mixed-Critical Avionics Architecture Approach for Civilian UAVs. *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 348–357, 2014.

[36] Premysl Sucha and Michal Kutil. TORSCHE Scheduling toolbox for Matlab. *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, Oct 2006.

[37] D. Tamas-Selicean and P. Pop. Optimization of time-partitions for mixed criticality real-time distributed embedded systems. *In 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 2–10, March 2011.

[38] D. D. Ward and S. E. Crozier. The uses and abuses of ASIL decomposition in ISO 26262. *System Safety, incorporating the Cyber Security Conference 2012, 7th IET International Conference*, pages 1–6, 2012.

[39] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, and Niklas Holsti. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, April 2008.

[40] Tianyu Zhang, Nan Guan, Qingxu Deng, and Wang Yi. On the analysis of EDF-VD scheduled mixed-criticality real-time systems. *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 179–188, 2014.

[41] X. Zhang, J. Zhan, W. Jiang, Y. Ma, and K. Jiang. Design optimization of security-sensitive mixed-criticality real-time embedded systems. *In L. George and G. Lipari, editors, Proc. ReTiMiCS, RTCSA*, pages 12–17, 2013.