Master's Thesis

Fachhochschule Dortmund

University of Applied Sciences and Arts

IDiAL Institute for Digital Transformation of Application and Living Domains

Development and Software Parallelization Evaluation of a Distributed Multi-core Demonstrator with APP4MC

Mustafa Özçelikörs

A Master's Thesis submitted in fulfillment
of the requirements for the degree of
Master of Engineering in
Embedded Systems for Mechatronics
at the Information Technology Faculty
of Fachhochschule Dortmund

Author:
Mustafa Özçelikörs
born on 06.03.1992
Matr.-Nr. 7099750

Supervisors:
Prof. Dr. Carsten Wolff
M. Eng. Robert Höttger

Dortmund, October 9, 2017

This project has been completed at the IDiAL Institute
along with publicly funded projects
AMALTHEA4public and APPSTACLE.

This project has also been granted
by Google Inc. and The Eclipse Foundation
during Google Summer of Code 2017.

Abstract

Distributing software effectively to multi core, many core, and distributed systems has been studied for decades but still advances successively driven by domain specific constraints. Programming vehicle ECUs is one of the most constrained domains that recently approached the need for concurrency due to advanced driver assistant systems or autonomous driving approaches.

In this thesis, software distribution challenges for such systems are discussed and solutions are presented upon instruction precise modeling, affinity constrained distribution, and reducing task response times achieved by advanced software parallelization. Furthermore, issues regarding tracing, distributing and power features are discussed in order to create precise models with APP4MC.

A demonstrator system called A4MCAR has been developed which features not only low level functionalities such as sensor and motor driving but also high level features such as image processing, camera streaming, server-based wireless driving via Web, bluetooth connectivity via Android application, system core monitoring features and a touchscreen UI. Moreover, experiments along the multi-task heterogeneous demonstrator A4MCAR show that using APP4MC results instead of OS-based or sequential software scheduling on a distributed heterogeneous system improves its responsiveness in some cases in order to potentially reduce energy consumption and replaces error prone manual constraint considerations for mixed-critical applications.

Acknowledgements

I would like take the opportunity and thank all people without whom this research would not have been possible. First, my supportive supervisor Robert who have patiently answered the countless questions I have asked and who haven't hesitated to involve me in a great community with great community events. Your helpful criticism coupled with your encouraging words have given me great confidence as a researcher.

I would like to also express my gratitude to Prof. Carsten Wolff, who with Prof. Peter Schulz, made the ESM course and IDiAL institute possible, and who have been helping me whenever in need of help. I would like to thank Prof. Andreas Becker for guiding me get through every issue with his heart-warming moral support, aiding me with my carreer choices, and also for giving me the opportunity to work with him on my project thesis. I would also like to thank Fatemeh, our beloved course coordinator, for guiding us through our confusions during difficult times.

Big thanks to all of the IDiAL members for being awesome researchers. I would like to thank Lukas, Uwe, Pedro, Philipp, and Cristoph for their support and companionship.

This research would not have been possible without the APP4MC community. It has been great to meet great people from FH Dortmund, Eclipse Foundation, Bosch and Fraunhofer. I would like to thank the APP4MC community: Jörg, Harald, Susan, Gael, David along with Lukas and Robert for making APP4MC possible.

Research projects need financial support. This research also would not have been possible without our funders. Our project has been funded by BMBF Fund.Nb 01-S14029K under Project AMALTHEA4public. During our participation in the Google Summer of Code 2017, this research have also been partially funded by Google and Eclipse Foundation with the prestigious GSoC grant for our project called "A4MCAR: A Distributed and Parallel Demonstrator for Eclipse APP4MC". I would like to thank all our funders and partner companies for their faith in our project.

Finally, I would like say special thanks to my family, who have always been there for me and who have blessed me with excellent moral support through thick and thin.

Contents

1. Introduction

1.1. Motivation

Developing and distributing software effectively is one of the most important concerns of today's software-driven fields. Effective software is surely needed in almost every part of embedded systems, especially in the fields of automotive, robotics, defense, transportation, electrical instruments, autonomous and cyber-physical systems. Optimizing software quality in the above mentioned fields undoubtedly creates a great demand for parallel software development for the last decades. This great demand caused software engineers especially in the information technology and embedded system sector to study parallel computing along with multi- and many- core systems under special real-time constraints.

The digitalization of almost every aspect of our lives as we know it requires systems to be more and more complex each passing day. While decades ago the computers had single-core processors, today almost every single computer has at least a couple of cores within their processors because of the fact that frequency scaling is not possible anymore. The advancements in processors allowed the development of more advanced systems with efficient software. For example, NASA's super computers collect and process data just on the topic of climate change that will reach 350 Petabytes in size by 2030, which is expected to be the same to the amount of letters delivered by the US Postal service in 70 years [1]. This should show how complex applications can be in the century we are living in. Furthermore, one of the most trending topics, Cloud Computing, which is being studied to make use of complex computing power of super computers remotely to public users, is being developed day by day and it will benefit greatly from the advancements in the field of parallel computing.

While parallel computing is used to meet the demands of more complex software, it is also widely used in more basic and cheap processors in order to execute more tasks with less resource consumption and cost. This is achieved by proper scheduling techniques. Furthermore, with an efficient software distributed efficiently to a processor's cores, one can also make use of less energy consumption features by applying techniques such as underclocking a processor. To summarize, developing efficient parallel software is not only useful in achieving advanced computing capabilities but also can help to achieve less energy and resource consumption, thus decreasing the cost of systems and making them more advanced and environment-friendly.

1.2. Objective and Contributions

Even though achieving concurrency using parallel computing is crucial, it comes with certain concurrency issues and often has to introduce new mechanisms to cope with such issues. Developers have to choose appropriate technologies and also have to consider not only the hardware constraints but also the software constraints in order to create efficient, safe, secure and reliable systems.

Before its execution, parallel software has to be delicately planned. The first stage of the parallel software development involves the Modeling activity. The model is later used for Partitioning, Task generation and Mapping stages to come up with an efficiently distributed software. In the modeling stage, the hardware and software models have to be created. While the software model is described by defining runnables, labels, label accesses, runnable activations and software constraints, the hardware model is described by defining processor details, hardware system clock and core information.

After the modeling activity, partitioning is done that determines which group of runnables belong together and can potentially run in parallel. Partitioning results are combined with system constraints in order to generate tasks. Finally, the Mapping involves laying out the details about pinning generated tasks to available hardware units and their cores [2].

While there exist some commercial tools that provide easement in the parallel software development, recent study done in Germany, namely AMALTHEA4public [3] [4], aims to provide planning and tracing tools especially for multi-core developments in automotive domain with several open source development tools. The successor of AMALTHEA4public, the APP4MC project [2] provides an Eclipse-based tool chain environment and a de-facto model standard to integrate tools for all major design steps in the multi- and many-core development phase. A basic set of tools are available to demonstrate all the steps needed in the development process. The APP4MC project aims at providing [2]:

- A basis for the integration of various tools into a consistent and comprehensive tool chain.
- Extensive models for timing behaviour, software, hardware, and constraints descriptions (used for simulation / analysis and for data exchange).
- Editors and domain specific languages for the models.

- Tools for scheduling, partitioning, and optimizing of multi- and many-core architectures [2].

This thesis aims to investigate and evaluate APP4MC's performance with a real-world distributed multi-core system in several aspects. The objectives of this project are as follows:

- Development of a distributed multi-core demonstrator for the APP4MC platform that involves typical automotive application features.
- Investigation of new trends in parallel software development (such as Real-time Linux parallel programming, POSIX threads, RTOS, evaluation methods etc.)
- Researching techniques to retrieve information (number of instructions, communication costs) and system trace from platforms such as xCORE and Linux to achieve precise modelling with APP4MC.
- In order to achieve optimization goals such as reduced energy consumption and reduced resource usage, different affinity constrained software distributions will be evaluated and energy features will be invoked to see if the goals can be achieved.
- Developing a basic online parallelization evaluation software that will retrieve scheduling properties such as slack times, execution times, and deadlines from all the processes and that will tell which deadlines were met and which not. Also, the software distribution assessment is in focus as well as investigating methods to develop schedulable and traceable threads and processes.
- Recording detailed system traces in order to provide offline software evaluation and consequently figuring out means to balance the load on cores.
- Comparing the conventional schedulers of non-constrained affinity distribution (such as a Linux OS scheduler) to the affinity constrained distribution from APP4MC to see if performance can be improved.

With the help of the A4MCAR project, which is a demonstrator for the APP4MC project, it is intended that the Real-time Linux community will benefit from the published libraries and documentation that involve code snippets and information instructions on how to develop optimized distributed and parallel software. Furthermore, the Eclipse APP4MC community benefit from the A4MCAR via advanced tool support for RPI developments, open source example applications, and validations of APP4MC parallelization results in order to create a better tooling available to the public. Those results can be used to assess and compare different parallelization scenarios and consequently identify optimal solutions regarding timing efficiency for the A4MCAR. Thereby, a point of reference can be given as well as an easy starting point for developers approaching parallelism with their developments.

1.3. Methodology

Automotive or any vehicle control related field tends to require sophisticated systems. In a real-life automotive application, the amount of hardware nodes and software nodes is comparably high. Since the main focus of the APP4MC environment is to provide parallel computation tools for the automotive domain, a demonstrator is required that is closely related to automotive domain and can be further used for troubleshooting APP4MC. For that purpose, a demonstrator RC-Car called A4MCAR is developed. Although an RC-Car does not match up the number of nodes used in real vehicles, the A4MCAR has several nodes and a distributed architecture, thus getting close to a vehicle's distributed architecture such as the AUTOSAR used in vehicles while not imposing too much complexity. Furthermore, the A4MCAR can be used for automotive-like applications that involve motor driving, navigation, sensor driving, and autonomous features.

The demonstrator, A4MCAR, is equipped with a distributed architecture that involves a 16-core multi-core microcontroller development board (XMOS xCore-200 eXplorerKIT [5]) and a 4-core single board computer (Raspberry Pi 3 [6]) with Linux OS. The software nodes with respect to their priorities and low-level and high-level purposes are distributed along those hardware modules. The demonstrator is not only designed to match up the capabilities of a real vehicle but also involves parts that are related to semi-autonomous driving and control. It can handle wifi and bluetooth connection requests and drive itself accordingly over a web interface or an Android application. Since the A4MCAR is specifically designed as a demonstrator, it has the capability to monitor and visualize core utilization and display it using a touchscreen or its web interface. Furthermore, it is equipped with four ultrasonic sensors and a camera with image processing embedded to support its autonomous driving and web interface streaming functions.

In this thesis, the development and parallelism evaluation of the demonstrator A4MCAR as well as the studies on parallel computing and tracing options are discussed. Obtained results are used in APP4MC for better development. The remainder of this paper is organized as follows: Chapter 2 is dedicated to background information on concurrent programming and design with APP4MC while Chapter 3 is dedicated to explaining the demonstrator design and implementation. After Chapter 3, Chapter 4 and Chapter 5 will involve information tracing and system management and results, respectively. The paper will be concluded with Chapter 6.

1.4. Events and Publications

This project has been partially published in several forms. With the supervision of Robert Höttger, the author submitted a paper on Mixed-critical Parallelization of Distributed and Heterogeneous Systems [7] using A4MCAR as a demonstrator on the conference Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS'2017) held in 21-23 September 2017 in Bucharest, Romania.

Due to the close cooperation of the APP4MC project with the Eclipse Foundation, the author of this thesis published a contribution called *Developing a multi-core enhanced RC-CAR using APP4MC* at the EclipseCon 2016 IoT Day in Ludwigsburg, Germany. The author also attended to two APP4MC hackathons on the topic of demonstrators in 2016 and 2017 held at The Eclipse Foundation Europe GmbH with the participation of many partners such as Robert Bosch GmbH and Fraunhofer IML in Zwingenberg, Germany. The author also participated in the unconference event of the conference EclipseCon 2017 France to present the work done at FH Dortmund regarding the Rover project, which is one of the other demonstrators of APP4MC. Furthermore, the A4MCAR demonstrator and its initial parallelization outcomes were presented at Dortmund International Research Conference 2017.

The author, with the supervision of his mentor, has been accepted to receive a Google grant with the project called *A4MCAR: A Distributed and Parallel Demonstrator for Eclipse APP4MC* [8] from Google Inc. and The Eclipse Foundation during the Google Summer of Code 2017 (GSoC'17) event. During the event, outcomes of the project as well as the code that is created has been contributed to The Eclipse Foundation in an open-source manner under Eclipse Public License 1.0 (EPLv1). Thus, the intention of helping Real-time and parallel software development community was supported crucially during the participation on GSoC'17.

2. Background Information on Concurrent Programming and APP4MC

2.1. Introduction to Parallelism

High performance systems, scientific computations and multi-feature systems require well-established parallel software. The physical problems in our world are being solved by embedded systems especially by making use of simulations which are becoming more and more complex as the years go by. Graphical applications which involve big data operations also make use of the benefits of the parallel software. As the data that is involved in such systems increase, required amount of computing power, memory space and the need for accuracy and speed also increases. In the last decades, the improvements in high performance computing and the advancements in processors are significantly developed which allows the development of efficient parallel software in systems. Today, computers with multicore processors allow every desktop to be eligible for parallel software development. In [9], it is pointed out that the technological developments regarding multi-core processors and parallel computing was forced by physical reasons. As increasing the clock speed causes overheating which gets harder and harder to get rid of as the clock speed increases, the developments regarding multi-core technology allowed more computations to be achieved without having to increase the clock frequency [9].

While the hardware-related developments are out there, in order to make use of parallelism one should modify an existing software to increase its performance on a multi-core processor. Furthermore, the execution time of the parallel program should be lesser the execution time of the sequential program for it to worth the effort. Designing a parallel program, as compared to a sequential program can be time consuming. In this regard, one should know the parallel programming models and modern techniques to utilize a software in parallel manner. With this idea, it can be said that there is much research going on in the area of parallel programming languages and environments with the goal of utilizing parallel programs at the right level of abstraction [9].

General purpose computer systems and **embedded systems** are both evolving with parallel computing. A general purpose computer system can be described to be a system without any specific domain or specific constraint on power, performance and cost; whereas an embedded systems are parts of larger systems and are always dedicated to dealing with specific problems in specific fields [10]. In both of the mentioned systems, giving timely response to events (also called **real-time computing**) is key in computing process [10]. Although both general purpose computer systems and embedded systems have basic computing, storage, and I/O components, embedded systems are used due to their cost effectiveness in larger scale systems. Furthermore, the evolution of embedded systems has allowed for them to make use of parallel computation features just like general purpose computer systems in the last decades. Therefore, a focus on optimizing the software on embedded systems are studied in this work in order to achieve application-specific, cost-effective, performance-effective and power-effective applications.

The following sections in this chapter will give an introduction to parallelism, modern techniques, and will explain the theoretical part of parallelism which is addressed in this report.

2.2. Memory Architectures

Parallelization of a program is all about efficient usage of CPU time (also called as computation time) and memory accesses. Assuming some tasks are distributed through a distributed, many-core or multi-core system, the synchronization between the tasks' accesses to memory should be achieved. With these goals in mind, machines according to their memory architectures could be summarized as follows:

- **Distributed Memory Machines (DMM):** Distributed Memory Machines are the systems that use distributed memory architecture. A DMM is illustrated in the Figure 2.1. In the distributed memory architecture, the units which consist of a processor and a memory are interconnected using a network [9]. There are several topologies with which the nodes can be interconnected. Such topologies involve point-to-point topology, bus topology, crossbar topology, ring topology, mesh topology and hypercube topology which makes use of an external routing unit [11]. The advantage of distributed memory is that each processing unit has individual memory units. Thus, all local memory is private and can only be accessed via the respective local processor. Therefore, the communication between the individual processor memories are only handled via a **message passing interface (MPI)** [9]. However, DMMs often have very large communication costs compared to other memory architectures. DMMs often resemble networks of workstations according to [9]. Collection of complete computers with a dedicated interconnection network are called **clusters** and they also use message passing interfaces for communication. The difference between cluster systems and distributed systems is that a job scheduler is used in clusters as opposed to being addressed individually. Today's popular message passing libraries to address communication in cluster systems and distributed systems involve PVM and MPI libraries [9].

- **Shared Memory Machines (SMM):** Shared Memory Machines are the systems that use the shared memory architecture. Since a shared memory architecture has a **shared physical memory (global memory)**, the coordination of processor accesses to global memory must be considered. A global memory can be a collection of memory modules and the data access to the global memory is handled by reading or writing **shared variables**. A shared memory architecture is illustrated in the Figure 2.1. A typical example of a shared memory architecture can be found in **multi-core processors** in which there exist many physical processing units called **cores**, and shared memory modules. In computers, parallel programs for SMMs are often handled by using **threads** which are control flows that share data with other threads by accessing a global address space. While **kernel threads** are managed by the **operating system (OS)** and mapped to the system cores by the OS, the **user threads** are often mapped to cores by using specific programming environments. Today's user threads within a computer system often use frameworks and libraries such as *OpenMP* and *POSIX* thread (Pthread) for C language, Java Threads, and Python's *threading* library [9].

  While using shared memory architecture is advantageous in terms of costliness, the design of shared variables and mutual exclusion of shared variables might be effortful and time consuming. The coordination problems of shared address space may lead to error-prone non-deterministic systems which will have the issues such as **race conditions** which will be described in the coming sections.

Distributed Shared Memory (DSM) architecture is often used in systems as well. A depiction of such architecture is given in the Figure 2.1. In distributed shared memory architecture, the shared memory model is implemented in a distributed fashion. The shared memory model allows all nodes to have a virtual memory space. Since distributed systems have high communication costs, by moving data to the location of access DSM systems reduce the high communication cost [12].

2.3. Introduction to Multi-core Processors

The processor chip industry progressively evolves since its existence. However, according to [9], the fashion of increasing the chip performance has changed due to the physical limitations. As an example, pipelined parallelism of instruction execution is limited by the

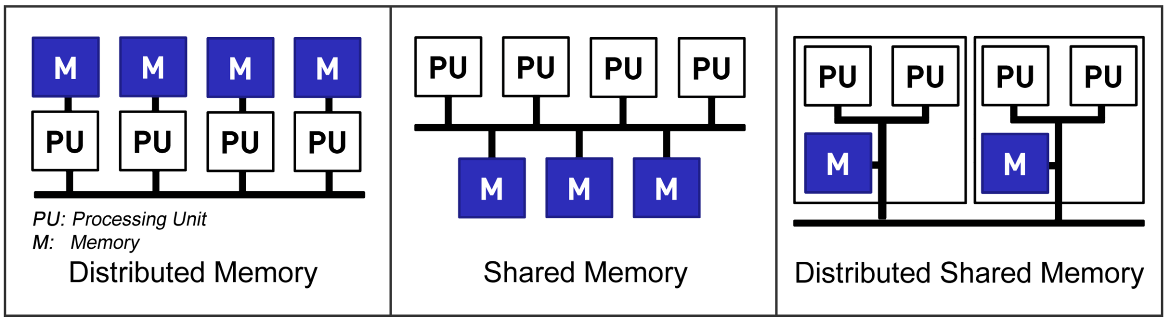


Figure 2.1.: Memory Architectures [13]

clock speed. The parallel programming book by Thomas Rauber et al. [9] explains these limitations as follows:

- Clock speed of a processor can no longer be increased significantly. Since the increase of the number of transistors on a chip is achieved by increasing the transistor density and this results in excessive heating due to current leakage, expensive cooling equipment are required. Furthermore, the heating on a chip should not exceed a certain limit to keep the chip functioning properly.
- Memory access times can not be reduced to the same rate as the processor clock increase. That results in a bottleneck when the processor is accessing the memory. **Cache memories**, which are temporary memories that are used in order to reduce the memory access times, are utilized in order to overcome this issue but further performance increase in processing units may not be resolved with caches.
- The speed of signal transfers within the wires could be a limiting factor.
- The physical size of a chip limits the number of pins that are used, which can limit the bandwidth between CPU and memory, which also creates a bottleneck at the memory communication.

Due to the aforementioned limitations, rather than increasing the performance, multi-core processors have been developed which consist of several physical execution cores. Due to the physical cores, total parallelism as opposed to single-core threading is achieved by using multi-core processors. Cores of a multi-core processor consist of seperate execution resources such as functional units (Arithmetic Logic Unit (ALU), multiplier) and execution pipelines [9]. A multi-core processor or a microcontroller also fits in a single chip package same as a single-core processor. Proper software application examples in a multi-core computer system involve background applications and scientific cpu-intensive computations that distribute a task into several cores to improve their performance.

Within a multi-core processor, usage of threads should also be understood. It is known that the single-core processor threading can be used to achieve parallel software with higher performance. The threading is also possible in multi-core processors which will achieve even higher performance and will allow more features to be parallelized. The graphical illustration of this is given with the Figure 2.2. Using the physical core for only one thread or a process is related to **mapping**, whereas the timing organization of many threads in one core is handled by **scheduling** which is done by the operating system. These terms will be further explained in the coming sections, primarily at the Section 2.4.

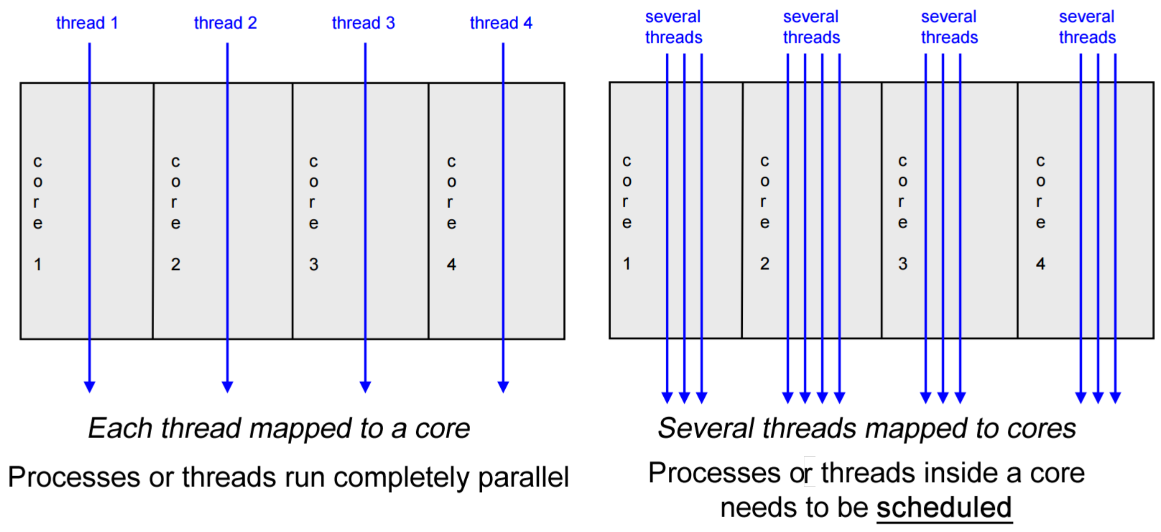

thread 1
thread 2
thread 3
thread 4
core 1
core 2
core 3
core 4
Each thread mapped to a core
Processes or threads run completely parallel
several threads
several threads
several threads
several threads
core 1
core 2
core 3
core 4
Several threads mapped to cores
Processes or threads inside a core needs to be scheduled

Figure 2.2.: Illustration of time-sliced execution paths in a multi-core processor [14]

2.4. Essential Concepts in Parallelization

Basic concepts in parallel programming should be understood to develop efficiently utilized software. The design of a parallel software starts by decomposing an application into its smallest pieces which are called **runnables**. This practice is especially used in the automotive domain. Using general terminology, One or more runnables combined constitute **tasks** which are functional pieces of an application that can be executed parallel across a multi-core or single-core processor. The size of tasks (mostly in terms of number of instructions) are called **granularity** [9]. Therefore, when decomposing an application into smaller pieces, granularity of the tasks should be considered for the load balancing. The tasks of an application are assigned to processes or threads for parallel execution on the hardware platform. Therefore, **process**es can be defined as programs that are assigned to execution resources to execute instructions concurrently. The switching between processes are called **context switches**, and the **scheduler** of the operating system manages those switches. A **thread** can be defined a continuous sequence of execution. A process can involve one or many independent control flows, i.e. threads [9].

Scheduling is the process of determining the order of execution of processes or threads on physical hardware whereas **mapping** is defined as the assignment of processes or threads to processing units. In other words, processes or threads are placed on cores using mapping, whereas their execution sequence is determined programmatically with the help of scheduling. The use of these techniques for parallel design introduce some challenges. Besides assigning processes or threads to cores, one should also manage assigning data to memories and communication to data paths. Constraints such as instruction set, locality, grouping, sizes, and deadline of the tasks can make this process effortly and time consuming [13]. **Synchronization** is also an important job which defines the organized communication between processes or threads [9]. Since the memory organization of the hardware matters, design should consider the hardware along with the software. The coordination and synchronization of processes and threads will be discussed in the next section, namely Section 2.6.

Finding a useful scheduling and mapping strategy is key to good parallel design. The practices involve keeping the **parallel execution time** of a task lower than the **sequential execution time**, keeping the load balanced through the cores of the system and keeping the communication overhead low. According to the book [9], for the quantitative evaluation of parallel programs, measures such as **speedup** and **efficiency** can be used which are measures that compare parallel execution time of a software with its sequential execution time.

2.5. Design Techniques in Parallelization

Parallelization can be defined as the transformation of a sequential program into a parallel program [9]. Although different sources ([9] and [13]) generalize the design techniques in parallelization differently, one can go through the following challenges in order to develop parallel software through parallelization (also illustrated with the Figure 2.3):

- **Partitioning of the problem:** The application that addresses a problem should be decomposed into smaller pieces, i.e. runnables that are considered to be the smallest pieces in a software. In some cases, this decomposition can be at task-level. A runnable or a small task can be a function that involves a single or more read or write accesses to a register or a shared variable, peripheral communication, or simple computations. In theory, a runnable is only executed in a single core and it has no dependencies to another runnables. However, the more complex tasks involve many runnables and introduce interdependency between them. The goal of this decomposition (according to [9]) is to make sure that all the applications are fine-grained enough that load balancing can be achieved. In other words, runnables should have small granularity and with the help of this the generated tasks can be distributed more efficiently.

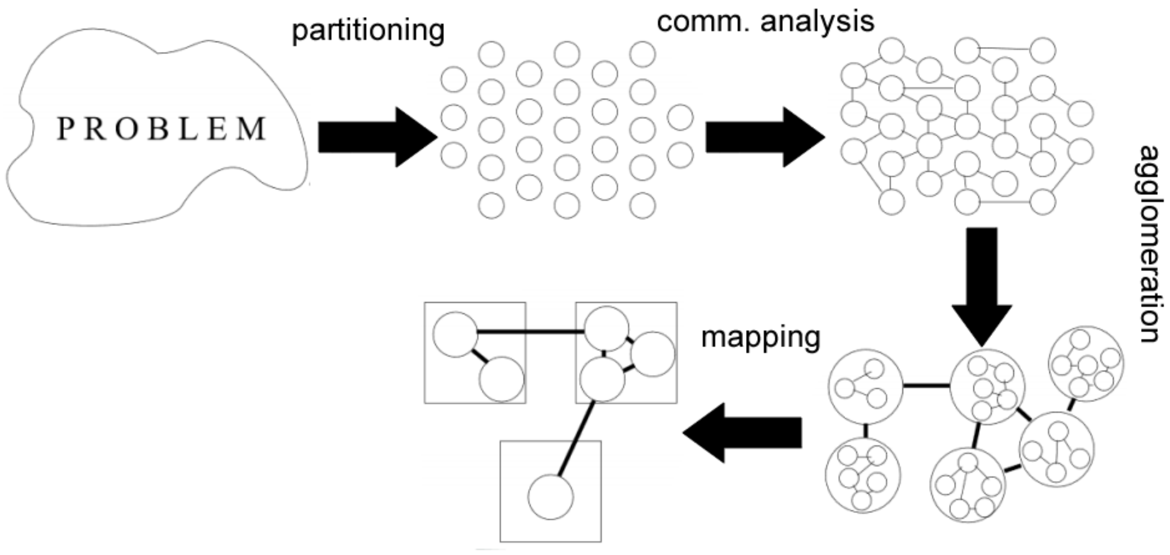


Figure 2.3.: Illustration of design techniques in parallelization [13]

- **Analysis of the communication:** Communication between runnables and tasks should be considered before the task generation. Communication is a big constraint in a parallel software, therefore all the dependencies between runnables or tasks, in terms of which runnable reads or writes data, and what is the communication cost (granularity) should be analyzed. Furthermore, it needs to be made sure that the inter-task communication produces low overheads as possible.
- **Agglomeration of executables to tasks (Task Generation):** In the task generation phase, runnables and some tasks are grouped together in order to constitute tasks. This is done with the consideration of dependent runnables in terms of communication. Also, the grouping should consider functional unification as well as load balancing.
- **Assignment of tasks to processes or threads:** According to [9], this intermediate phase is involved in computer systems where processes and threads are involved. The tasks that are generated can be grouped in order to constitute processes or threads. This step makes the software mapping-ready. In the cases of some multi-core microcontrollers, such as the design that is presentated in this work using APP4MC, the mapping is done at the task-level. Therefore, this intermediate step is not required.
- **Mapping of processes or threads to physical processes or cores:** Each process, thread, or task in some cases are assigned to a seperate processor or core after the final agglomeration phase, i.e. when the runnables are grouped sensibly. In cases where there are more process or thread than a core, multiple threads are assigned to some cores. In this case, as mentioned scheduler of the operating system takes care of the execution order of multiple processes or threads on a core. The mapping phase in computer systems is usually done by the operating system but the users can

intervene. Usually, the main goal of the mapping step is to get an equal utilization while keeping the communication overhead smallest [9] [13]. However, one can set-up different optimization goals such as reducing the power consumption for the mapping stage.

2.6. Coordination and Mutual Exclusion

A **critical section** can be defined as a shared variable that is accessed by two or more processes or threads. When developing parallel programs, not managing critical sections correctly might lead to error-prone systems. The actions of the processes or threads, when accessing a critical section, should be co-ordinated. For that purpose, **mutual exclusion** (also called **mutex**) concept must be investigated which refers to the requirement of ensuring that no two concurrent processes are in their critical section at the same time [13]. Generally, mutual exclusion is handled by using mutex algorithms such as Lamport's Bakery Algorithm [13], ensuring that every process is live, safe (no non-deterministic behavior is involved) and granted a fair amount of CPU time.

There are many problems that can occur in case the coordination of the process communication via accessing critical sections is not handled properly. A problem that is called **race condition** can occur in case mutual exclusion is not properly provided. Race conditions can be defined as the behavior of a system where the system behavior depends on the order or timing of the uncontrollable (sporadic) events [13]. In other words, race conditions happens when the system behaves non-deterministically due to the non-fixed order of accesses to critical region. Another problem that might arise is the **deadlock** problem which is the issue of processes not releasing the shared variables and waiting to acquire other shared variables that are being used by other processes [13]. In a deadlock scenerio, neither process can continue working therefore the execution does not progress. In a **livelock** situation, however, two or more processes access a shared variable but does inverting actions, thus even though the processes are alive the execution does not progress and has a non-deterministic behavior. The **spinlock**, a mechanism in which a task is continuously waiting for a shared variable to be released, can be given as another example of the problems that might occur if the mutual exclusion is not properly implemented.

Today, many thread and concurrent programming libraries involve functions to use mutual exclusion. A simple way to access a shared variable in a programming language could be to define a mutex variable and lock it before writing to this critical section. After the process writes to the critical section, one must release the section by unlocking the mutex variable. However, it must be remembered that while handling complex critical sections where many processes are involved, coordination of process access' to the critical section should be provided by using mutual exclusion algorithms such as Lamport's Bakery Algorithm. **Semaphores** (which are data types that control the access of multiple processes to a shared resource), **monitors** (which provides buffered locking mechanism to concurrent accesses), and synchronization methods (such as message passing) can also be used in order to coordinate the actions of multiple processes to critical sections [13][9].

2.7. Analysis of Scheduling

For the purpose of analysis and evaluation, one can use system traces that involve information such as timing, mapping, and priority in order to see if the tasks, processes or threads behave as expected. To meet the optimization goals, this work involves the evaluation of several software distributions.

According to [13], scheduling quality of service (QoS) could involve the following goals:

- Even load-balancing
- Efficient resource usage
- Maximal throughput (completed processes per time unit) and utilization (percentage a processor is used)
- Minimal response time and latency
- Maximal fairness (every process receive fair amount of cpu time depending on their granularity)
- Avoiding starvation (every process is guaranteed to receive cpu time eventually)

In the following section (Section 2.8), the optimization goals for parallel software are discussed. It can be argued that an important portion of the optimization goals for a parallel software can be achieved through the scheduling goals.

To understand how tasks are scheduled, one must carefully study the task graph given in the Figure 2.4. In the figure, it is seen that two tasks A and B are given with task B having a higher priority but lesser execution time than task A. Timing properties of the given timeline are defined as follows [15] [13]:

- **Initial Pending Time (IPT)**: IPT is defined as the preparation time before a task is ready to be started execution.
- **Core Execution Time (CET)**: CET is the absolute time elapsed at which the task is executed on the processor. Therefore, the amount of time the task is preempted are not counted when calculating the CET. In the figure, it is seen that Task A is preempted once, thus the overall CET is found by adding the times CET1 and CET2.

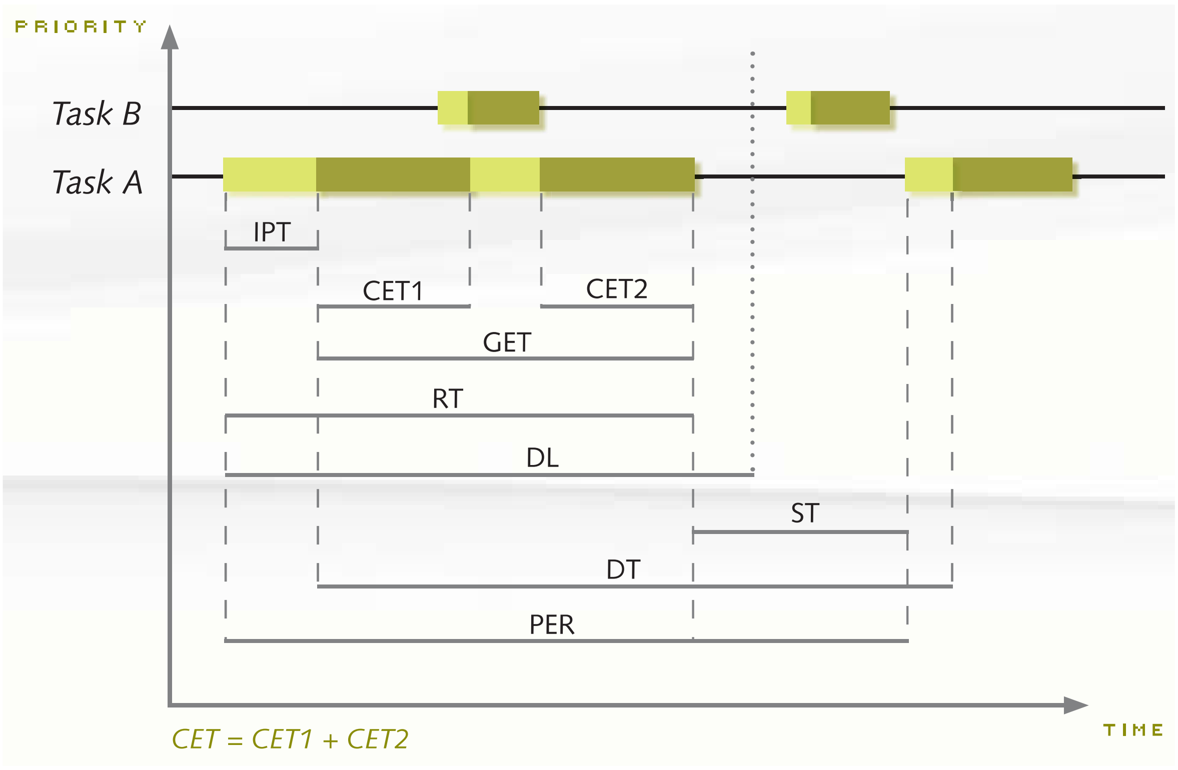


Figure 2.4.: Timing properties for scheduling in multi-tasked systems [15]

- **Gross Execution Time (GET)**: The GET is the amount of time it passes for an iteration of a task or an event is executed. In other words, it is the elapsed time the execution is done until the task goes into the sleeping state.
- **Response Time (RT)**: The response time is calculated by adding IPT and GET, i.e. $RT = IPT + GET$.
- **Deadline (DL)**: The deadline of a task or an event is the amount of time a task has to be completed in order to meet the real-time requirements of the task. Therefore, a reliable task must not miss its deadline even in the worst case. The deadline misses (DLM) can be used as an evaluation measure for the reliability of a task. It can also be used as a measure to compare the quality of software distributions as seen in Section 5.
- **Delta Time (DT)**: The delta time of a task can be defined as the time between two gross executions of a task.
- **Slack Time (ST)**: The slack time is the amount of time a task is at a sleeping state. In other words, slack time is the time between the last response of a task and the start of the pending state (IPT). Slack time describes the flexibility of a task, thus it is the time that a CPU can be used for other tasks. Therefore, if the slack time is higher in a task,

that task is considered to be less stressed. Slack time is also a measure that is used in the evaluation of different software distributions.

- **Period (PER)**: The period is one of the most basic properties of a task which defines how long it takes before a task repeats its instructions. The period is defined for tasks that are periodic.

Although the given timeline depicts the overall timing properties of task scheduling, due to tracing and instrumentation limitations not all of the information is extractable from the system trace. IPT, for example is a timing property which is not present in the tracer that is used in the work that this thesis explains. Therefore, IPT is neglected due to the aforementioned limitations and the fact that it is bound to be a rather small amount of time. Scheduling analysis is implemented in this work in Section 3.3.2. Furthermore, how information of what kind of evaluation metrics are used are further explained in Section 2.8.

2.8. Optimization and Evaluation of Parallel Software

Not every parallel program is beneficial. In order for a parallel program to be useful, it should be optimized. There are four main reasons for a parallel software to be optimized. Each optimization process focus on a goal and the design and development of the software should be carried out by considering that goal. The optimization goals (depicted in the Figure 2.5) involve achieving lowest energy consumption, achieving lowest computation time, achieving highest resource utilization, and achieving highest reliability. The way these goals are achieved are also shown in the Figure 2.5. It should be added that maximizing or minimizing any software property in a positive way can be generally seen as an optimization problem that deals with e.g. safety, security, distributivity, or scalability demands among others [7]. However, the figure given shows the basic software optimization goals that are related to this work.

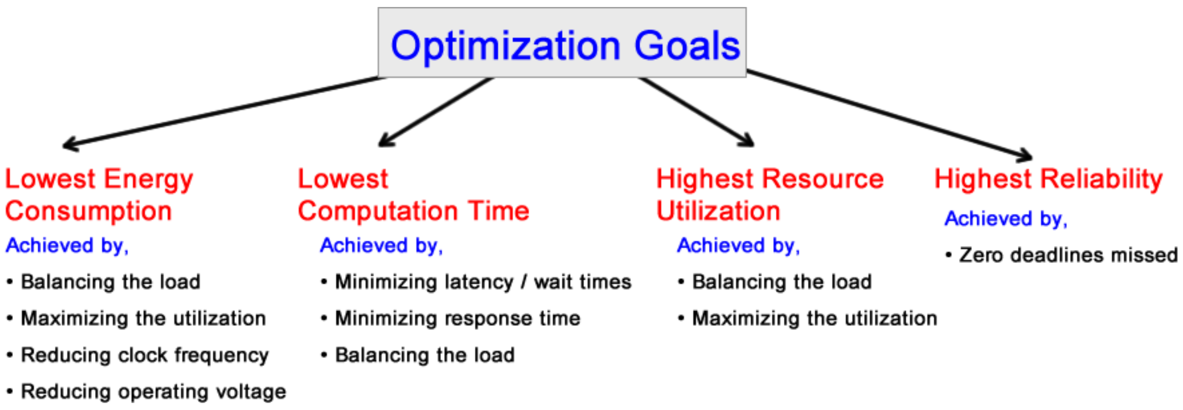

Optimization Goals
Lowest Energy Consumption
Achieved by,
• Balancing the load
• Maximizing the utilization
• Reducing clock frequency
• Reducing operating voltage
Lowest Computation Time
Achieved by,
• Minimizing latency / wait times
• Minimizing response time
• Balancing the load
Highest Resource Utilization
Achieved by,
• Balancing the load
• Maximizing the utilization
Highest Reliability
Achieved by,
• Zero deadlines missed

Figure 2.5.: Optimization goals of parallel software [13]

On a higher level, there is always a question whether the optimized software is really optimal or not. There are strategies with which this could be measured and true optimization could be achieved [13]. Those strategies involve Linear Programming, Integer Linear Programming, Genetic Algorithms, Mixed Integer Linear Programming, Quadratic Programming, Evolutionary Algorithm, and Simulated Annealing [13]. Although it is useful to know such strategies exist, the optimization of an parallel software on a higher abstraction layer will not be a part of this work.

As mentioned before, one of the most basic evaluation techniques to determine if a parallelized software is beneficial as opposed to a sequential software is to check the run-times of the processes. If the parallel run-time (overall execution time) of a program is less than the sequential run-time, then it could be said that parallelization benefits in terms of computation time. The following list involved the parallelization evaluation techniques that will be a part of this work in evaluating software distributions.

- ST_{avg}: Average slack time of all traceable processes
- DLM: Percentage of deadlines missed
- $U_{\text{0-p}}$: Percentage of utilization of each core
- $S_{\text{p}}(n)$: **Speedup** value which quantitatively compares the execution time of a sequential implementation with that of the parallel implementation [9]. Speedup is calculated as follows [9], given that $T^{*}(n)$ is the sequential run-time and $T_{\text{p}}(n)$ is the parallel run-time:

  $$S_{\text{p}}(n) = \frac{T^{*}(n)}{T_{\text{p}}(n)} \tag{2.1}$$

  It can be said that the bigger the speedup value, the better the parallel utilization.

The aforementioned evaluation techniques are mostly used to evaluate load balancing ($U_{\text{0-p}}$), reliability (DLM) and resource utilization (ST_{avg}, $U_{\text{0-p}}$ and $S_{\text{p}}(n)$).

2.9. New Trends in Parallelization

New practical trends evolve in software development every passing day. Since today's applications are more complex than they were before, software development field tends to grow. Surely, with the applications getting more complex, more advanced development environments, platforms, and frameworks are needed. Software engineers have been trying to fulfill this need especially since embedded, internet-based and cyber-physical systems became more common. Since some systems require hard real-time requirements, real-time development has became an important field of study.

One of the most common concurrent programming libraries involve **Real-time Operating Systems (RTOS)**. Real-time operating systems are the operating systems or more practically frameworks that are developed for microcontrollers or processors. They provide real-time computing to the viable hardware in order to be used in the applications that requires deterministic and predictive task behavior [16]. RTOS' implement kernel components which are scheduler, thread managers, message queues, semaphores, mutexes, timers, and memory pools [16]. In the basic sense, an RTOS can be used to schedule applications, send and receive messages, get and release semaphores, and handle events in order to manage a parallel system [16]. RTOS' are common for both single-core and multi-core processors. With a single-core processor, many threads are scheduled within a single core whereas in a multi-core processor, the mapping features can also be used. As an example, *FreeRTOS* (also currently called as *CMSIS-RTOS* [17]) is a common RTOS library that is developed for *ARM Cortex M* series microcontrollers (such as *STM32F407* or *STM32F103C8T6*). Other examples involve *OSEK/VDX*, *RTX*, *FreeOSEK* etc.

As introduced, multi-processor and multi-core solutions are also common. For distributed systems, MPI C/C++ library uses compiler directives to parallelize applications. Furthermore, OpenMP is a C/C++ library that is developed for systems with shared memory architecture and it can be used as a different solution to parallelize applications and manage message passing.

Since thread-level parallelism, i.e. using virtual cores to manage many-tasked, advanced, featured systems, is very common since the multi-core processors became popular, almost every programming language has its own separate libraries or frameworks to support multi-threaded software development. A few examples could be given: Java's *Thread* library, C/C++ *POSIX Thread (Pthread)* library, and Python's *threading* library [18]. One could also make use of the mapping features of Linux kernel to develop parallel applications on the process level.

The tendency of applications in the embedded computing domain also made concurrent programming popular in single board computers such as the Pandaboard, Beagleboard, Beaglebone, and Raspberry Pi etc. Since a single board computer can run many programming languages and they provide multiple cores nowadays, one can easily use the aforementioned libraries and frameworks to develop concurrent applications for them. However, although parallelization is not an issue, some single board computers are not very beneficial in terms of real-time computing. Therefore, since Raspberry Pi is used in A4MCAR due to being low-cost, real-time aspects of it are also be discussed.

According to [19] and [20], a **real-time application** can ensure guaranteed response within strict timing constraints. Real-time nature of an operating system is related to **interrupt latency**, the time to process an interrupt, and **scheduling latency**, the time to start a processing task [20]. Although Raspberry Pi is a good embedded platform for most of applications, its real-time capabilities are not satisfying in terms of the following [20]:

- Current Linux or BSD kernels for Raspberry Pi do not support real-time.
- Current Linux or BSD kernels for Raspberry Pi have a lot of overheads.
- Raspberry Pi does not have a real-time clock.

The solutions offered in [20] involve adding pulse generators or real-time clocks, building new Linux kernel without unnecessary services to remove overhead, or using real-time operating systems such as *RTEMS* or *PREEMPT-RT* that are ported to Raspberry Pi. However, the work that is presented with this thesis doesn't use such system because of the complexity of applications and the need for using multiple languages and runtime environments for applications.

Since this work targets the automotive domain, the reader would benefit from knowing what concurrency solutions are used for automotive systems. Since an embedded system in an automobile consists of many ECUs (Electronic Control Units) with multi-processor capabilities, it is crucial to handle synchronization and communication between the distributed and parallel nodes. A software solution that is used by the most automotive OEMs and part suppliers is the **AUTOSAR** platform[21]. AUTOSAR is a framework that is used for microcontrollers or processors which have services on many abstraction layers to deal with memory, communication, I/O operations. A real-time runtime environment (RTE) deals with scheduling (OSEK scheduler), events, timers and semaphores apart from the application. The AUTOSAR architecture, therefore, intends to provide modular applications that only communicate using hardware and software ports with other micro-controllers and other embedded ECUs. The hierarchical representation of AUTOSAR architecture can be seen in the Figure 2.6. Figure shows that there are many drivers, hardware abstraction layers and services before application is scheduled and executed.

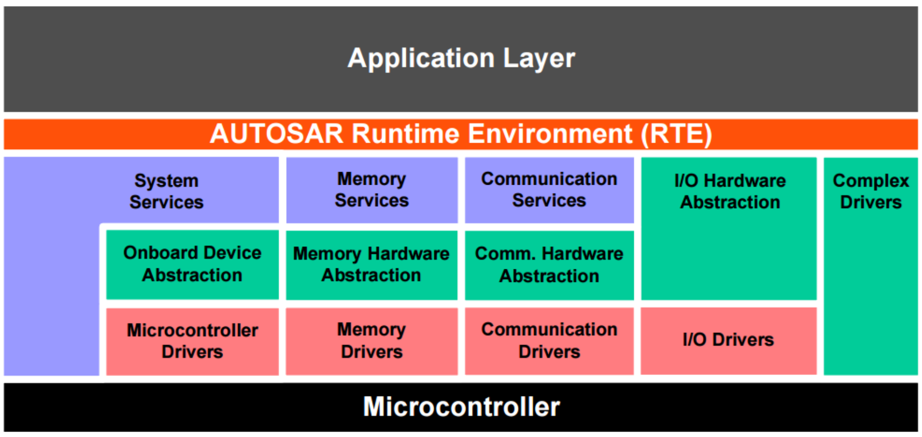


Figure 2.6.: AUTOSAR Architecture [21]

2.10. Parallel Design using Eclipse APP4MC

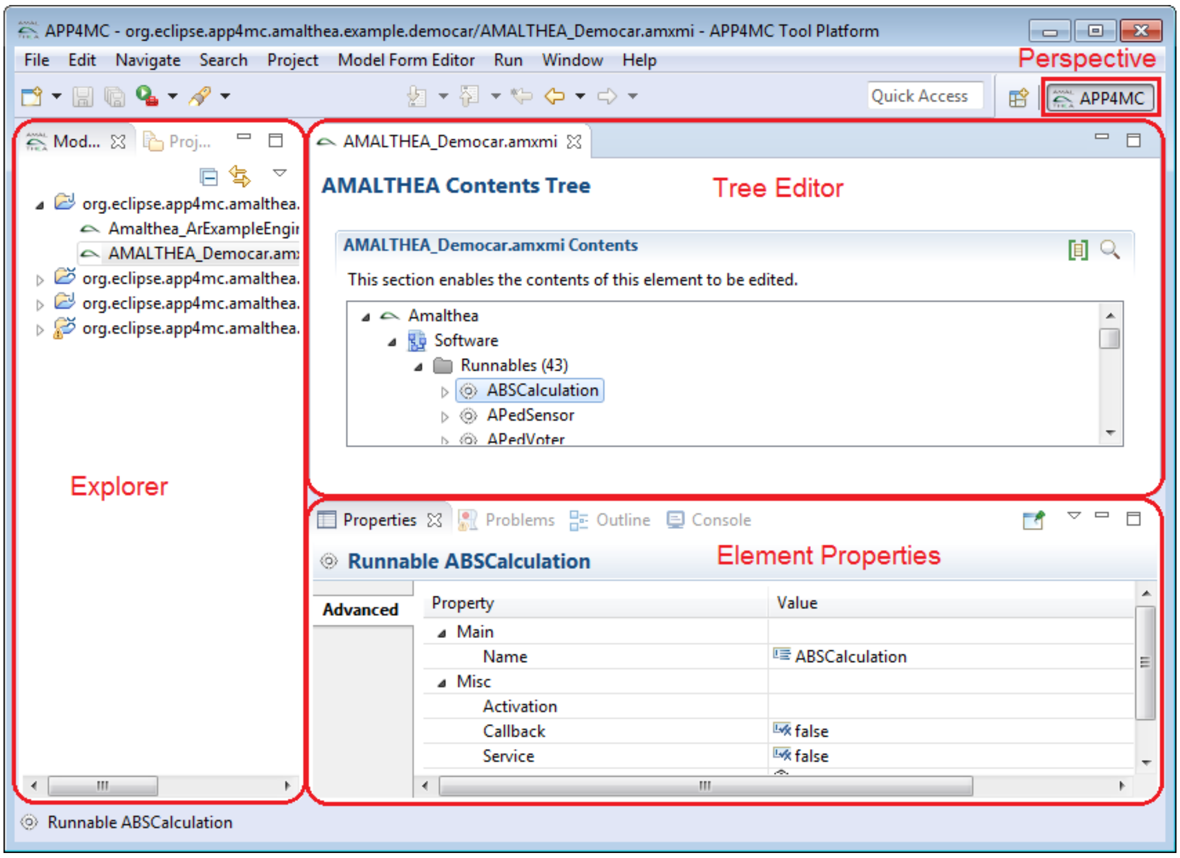
As introduced in the Chapter 1, this work uses the Eclipse APP4MC development tool for software parallelization. **APP4MC (Application Platform Project for MultiCore)** [3] [22] is an Eclipse-based project that aims to achieve an open, consistent, expandable tool platform for embedded software engineering [23]. APP4MC targets multi-core and many-core platforms, while the main focus is the optimization of embedded multi-core systems [23]. Due to its focus, APP4MC is partnered with many automotive OEMs and part suppliers that deal with embedded software engineering. Furthermore, it supports interoperability and extensibility and unifies data exchange in cross-organizational projects [22]. Additionaly, since APP4MC uses Eclipse platform to its purposes, the development environment has a complete open-source nature under Eclipse Public License 1.0 (EPLv1) [24].

The Eclipse APP4MC platform editor window can be seen in the Figure 2.7 [23]. In the figure, the *Explorer* window is used for finding models, performing operations such as partitioning, task generation, mapping, and model migration. The tree editor shows the hierarchical structure of the selected AMALTHEA model, whereas the *Element Properties window* is used for editing the properties of AMALTHEA model elements selected in the *Tree Editor* [23].

APP4MC is a project founded by a publicly founded project AMALTHEA4public [4]. APP4MC uses AMALTHEA models, which are XML-based EMF models that describe software components and hardware platforms. The main operation of APP4MC involves modeling the system by creating AMALTHEA models and performing partitioning, mapping, optimization on parallel programs [3]. APP4MC also has the ability to trace and simulate parallel programs. Basic ingredients for an AMALTHEA model are illustrated in the Figure 2.8 [22]. It is seen that AMALTHEA model can contain software decisions, costs, constraints, as well as hardware platform information [22]. Constraint models are used to define process groups to make sure some processes belong together. Furthermore, a target platform dependency of a process group is also modeled using constraints model. More information on modeling will be discussed later in this section.

An illustration of how parallel software can be designed for embedded multi-core platforms are given in Figure 2.9. Studying the illustration given in Figure 2.9 in combination with the parallel design techniques that were given in the Section 2.5, the following remarks can be made regarding the design procedure with APP4MC platform:

- **Modeling:** Design of a parallel software starts with modeling in APP4MC. An AMALTHEA model is constructed that involves three seperate models: (1)- hardware model, (2)- software model, (3)-constraints model. In the hardware model, each distributed ECU is modeled in a hierarchical manner. The hardware model involves information such


APP4MC - org.eclipse.app4mc.amalthea.example.democar/AMALTHEA_Democar.amxmi - APP4MC Tool Platform
File Edit Navigate Search Project Model Form Editor Run Window Help
Perspective
Quick Access
APP4MC
Mod... Proj...
org.eclipse.app4mc.amalthea.
Amalthea_ArExampleEngi
AMALTHEA_Democar.am
org.eclipse.app4mc.amalthea.
org.eclipse.app4mc.amalthea.
org.eclipse.app4mc.amalthea.
Explorer
AMALTHEA_Democar.amxmi
AMALTHEA Contents Tree
Tree Editor
AMALTHEA_Democar.amxmi Contents
Amalthea
Software
Runnables (43)
ABSCalculation
APedSensor
APedVoter
Properties Problems Outline Console
Runnable ABSCalculation
Element Properties
Advanced
Runnable ABSCalculation

This section enables the contents of this element to be edited.

Property	Value
Main	
Name	ABSCalculation
Misc	
Activation	
Callback	false
Service	false

Figure 2.7.: Eclipse APP4MC platform Editor Window [23]

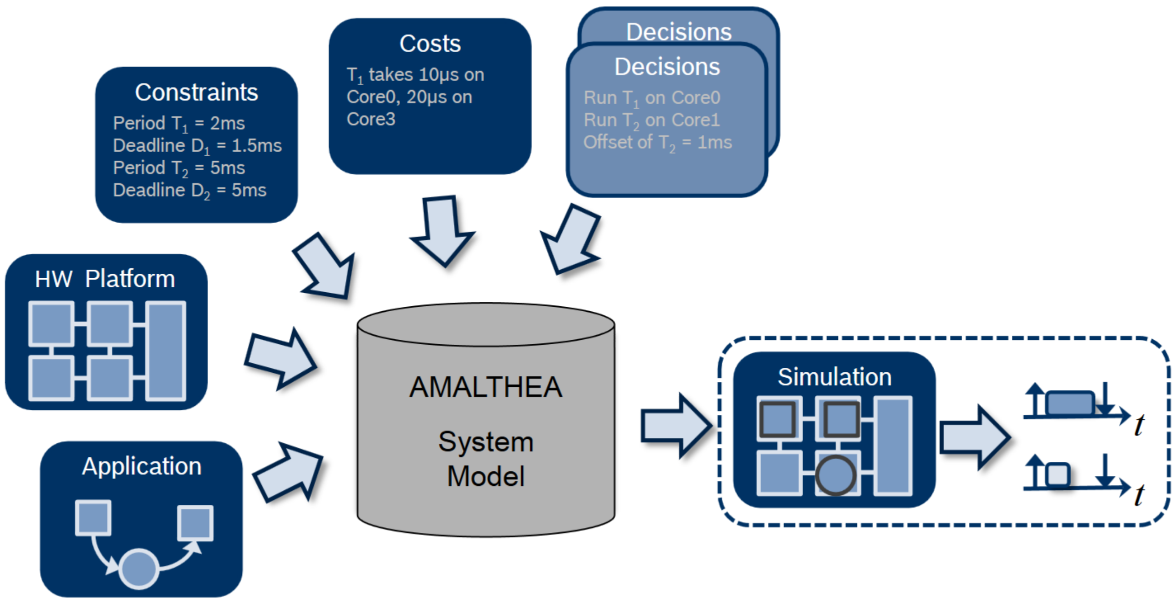


Figure 2.8.: AMALTHEA Model for APP4MC [22]

as number of processor cores, system clock frequency for processors, and memory details. In the software model, runnables are modeled among others. Runnables are found with the help of binary analysis tools and by using the decomposition technique mentioned in the Section 2.5. Each runnable is modeled by making use of information such as granularity (number of instructions) and label accesses (memory read-write). In the early development stages, the model contains a rough model of the software, but the model is constantly improved by using either the Tracing functionality of the APP4MC or other tracing software.

- **Partitioning:** The partitioning stage in APP4MC-aided parallel design corresponds to identification of initial tasks. After an initial AMALTHEA model is constructed, one can perform partitioning in APP4MC by simply selecting the model and pressing the *Perform Partitioning* button. At this stage, APP4MC will analyze the runnables and runnable label accesses in order to suggest how tasks should look like for a balanced parallel distribution. APP4MC currently uses two partitioning algorithms that are ESSP (Earliest Start Schedule Partitioning) (performed by default) and CPP (Critical Path Partitioning) in order to find the partitions of the system. ESSP and CPP algorithms are based on the graph theoretical analyses [25] which are commonly used in hardware and software co-design. Partitioning algorithms are used for analyses of the granularity and communication costs of individual runnables and create best possible parallel partitions.
- **Task Generation:** Initial tasks (partitions) are finalized by pressing *Generate Tasks* button. By making use of the dependencies between partitions and by grouping them, APP4MC generates a model that contains the desired amount of tasks with the help of *Task Generation* phase.
- **Mapping:** As known, mapping is the stage of placing the software distributions (tasks, processes, threads) into the processors. By making use of the hardware capabilities and using optimization strategies (such as Integer Linear Programming or Genetic Algorithms), APP4MC is able to find a mapping model of the system. The utilization details of the simulations will be seen at the end of the mapping stage.
- **Code Generation:** Since APP4MC provides model-based development, code generation features for C language could be written for proprietary platforms.
- **Tracing:** By making use of binary tracing, AMALTHEA trace model can be observed and re-used to update the system model.

APP4MC promises beneficial set of tools for embedded parallel software development. However, the demonstration of its features is needed for further improvement. Therefore, in this work APP4MC is used for system parallelization for the A4MCAR, a demonstrator RC-Car. Further sections will involve design, modeling, and evaluation of software distributions for this demonstrator.

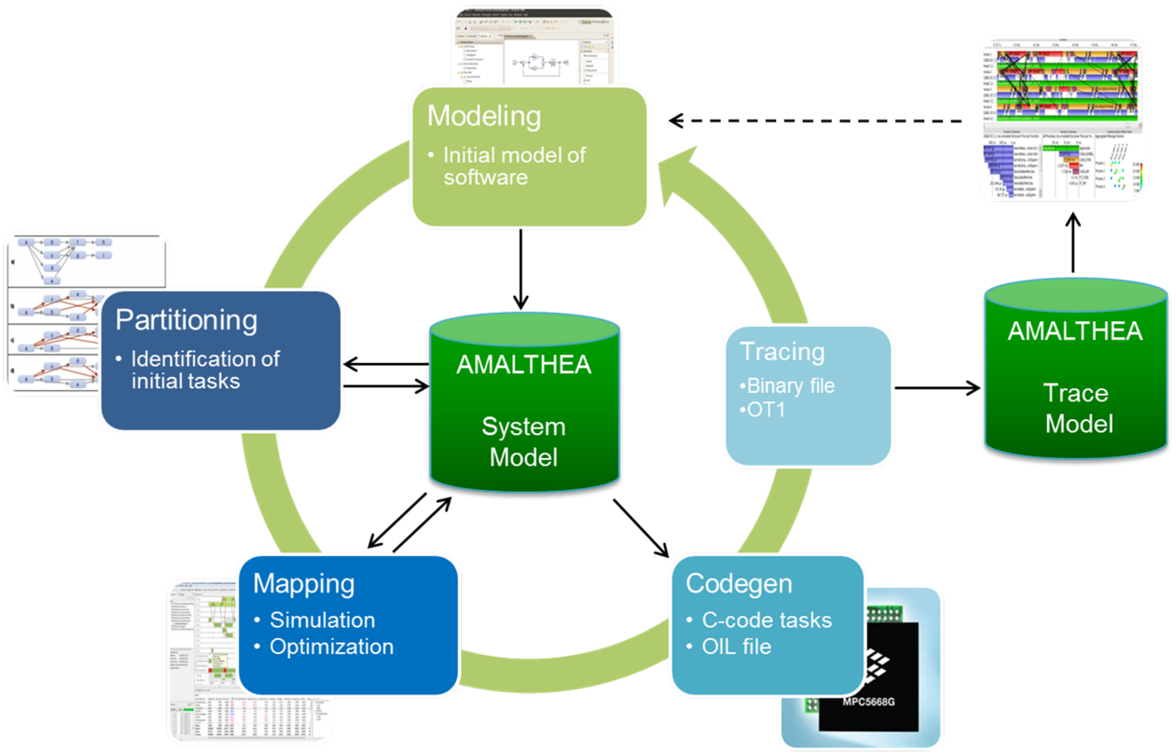


Figure 2.9.: Illustration of how parallel software are designed using APP4MC platform [4]

2.11. Related Work

There are many studies done in the direction of efficient software parallelization methodology and tooling such as [26], [27], [28], [7], [29], [30], [31], [32], [33], and [34]. The researches that involve APP4MC-based and AMALTHEA model-based parallelization are e.g. [26], [27], [28], [7], whereas some useful publications in the same direction that doesn't involve APP4MC or AMALTHEA can be given as [29], [30], [31], [32], [33], and [34].

The work by Carsten Wolff et al. [26] introduces the evolution of the AMALTHEA tool project between 2011 and 2013. Many aspects such as standardization, tool-chain support and evolution, software development methodology in the tool-chain are explained. The paper also introduces Eclipse-based open source framework development. Due to AMALTHEA's close relevance to APP4MC, it is crucial to understand how AMALTHEA is tailored and how it became an open-source embedded multi-core development platform.

While the aforementioned paper explains the standardization and tool-chain aspects, the paper by Robert Höttger et al. [27] describes the model-based partitioning and mapping features of AMALTHEA (and by extension APP4MC) with the emphasis of automotive domain. In the paper, novel approaches to partitioning and mapping in terms of model-based embedded multi-core system engineering are introduced. This work shows that the performance, energy efficiency and timing requirements are improved using their partitioning and mapping methodologies. The work compares approaches such as Critical Path Partitioning and Earliest Start Scheduling Partitioning regarding partitioning and GA- (Genetic Algorithm) based optimization approaches towards mapping. How specific goals such as energy consumption and load balancing are addressed are also included in this paper. The paper also discussed benefits, industrial relevance and features of the presented model-based multi-core partitioning and mapping techniques in common with existing approaches.

The publication by Andreas Sailer et al. [28] gives an extended practical comparison of distributed multi-core development standards in the automotive domain. The XML-based automotive software standards ASAM MDX, AUTOSAR and AMALTHEA are compared with regard to their model, methodology, and reference implementation. It is mentioned that out of three standards only AMALTHEA is open-source and has special focus on multi-core development. It is also mentioned that AMALTHEA exchange format allows the detailed specification of dynamic software architecture properties. Regarding the overall comparison results using a case study, although it is stated that AUTOSAR is much more than just a standard to automotive software development and is capable of many things, the paper concludes that AMALTHEA is able to address some of the deficits of AUTOSAR within the scope of multi-core. Also, the AUTOSAR compatibility of AMALTHEA within the project AMALTHEA4public as a relatively new but open-source supplementary tool to the automotive world is highlighted in the paper [28].

One work that the author of this thesis is also involved [7] addresses constrained mixed-critical parallelization for distributed heterogeneous systems using APP4MC. This work explains addressing software parallelization via precise modeling and affinity constrained distribution. In the research, the precise modeling, partitioning and mapping features of APP4MC are used in order to achieve software parallelization on the demonstrator A4MCAR. Experiments along A4MCAR show that using new distributions from APP4MC creates significant improvement regarding proper parallelization and energy efficiency compared to the sequential distributions or distributions that are created by the operating system, which are the experimental conformance to the results discussed in the work [27]. In the work, it is also addressed that due to less context switching compared to OS-based distributions, *affinity constrained distribution* using the results from APP4MC could help to achieve a software that consumes less energy on the hardware system. The work states that by underclocking the CPU without creating and deadline misses could also decrease the energy consumption significantly. Besides the aforementioned aspects, the paper [7] also addresses the APP4MC's model-based technique and capabilities regarding partitioning and mapping along with tools and methods on Linux platform to gather information to create precise software models. Safety considerations such as *ASIL-level based partitioning and mapping* is explained in the work which shows the APP4MC's relevance to the automotive domain [7].

As mentioned in the first paragraph of this section, there is several other work that targets the same direction as AMALTHEA or APP4MC. As an initial example, Devika et al. [29] explains the implementation of an AUTOSAR Multicore Operating System. In their work, Devika et al. talk about real-time operating system OSEK/VDX, standards set by the AUTOSAR standard, and the implemented multi-core features of AUTOSAR. Also the challenges such as spin-locks, deadlocks, starvation, fairness are investigated. In order to understand how such challenges are tackled in industry, the work done by Devika et al. is surely a good starting point. Another example of good reading materials could be given as the contributions done by Navet et al. [30] and Alfranseder et al. [31]. In their paper, Navet et al. talk about safety critical aspect of multi-core automotive ECUs, i.e. operating system protection mechanisms. Strategies toward scheduling and load-balancing are also explained in their work. The work by Alfranseder et al. [31] however try to find solutions to two crucial questions. In their words, those questions are *"How can one schedule real-time tasks to the available cores in an optimal way?"* and *"How can one handle synchronization of shared resources with minimal overhead?"*. They present a spin-lock and busy-wait based resource sharing protocol to answer these questions. As for more examples for work done in the direction of timing synchronization and tracing, the works by Yu et al. [32], Lu et al. [33], and Nilakantan et al. [34] could also be beneficial to the reader.

3. Distributed Multi-core Demonstrator (A4MCAR) Design and Implementation

3.1. System Overview

As introduced in the Chapter 1, the A4MCAR is a demonstrator RC-Car for the APP4MC development environment. The A4MCAR provides a distributed multi-core architecture that allows the demonstration of embedded low-level and high-level applications. Pictures of the A4MCAR can be seen in Figure 3.1.

3.1.1. System Features

As the A4MCAR targets automotive industry and parallelization studies done via APP4MC, it features not only sensing and actuation related features but also applications that would help with task to core distributions and parallelization performance evaluation. The featured applications for the A4MCAR are illustrated in Figure 3.2. In the figure, it is shown that the low level module of A4MCAR, built using xCore-200 eXplorerKIT targets mostly actuation and sensing related applications. The full list of tasks developed for the low-level module includes:

- Core monitoring applications for each tile (two exist) that calculates the average core utilization.
- Bluetooth task to configure the bluetooth module in slave mode and receive data over UART interface.
- Proximity measurement task that obtaines the distance sensor information from four SR-04 sensors over an I2C sensor network.
- Speed control task in order to use PWM to control the speed controlling motor.
- Steering task that controls a servo motor using PWM signaling in order to steer the A4MCAR using external inputs.

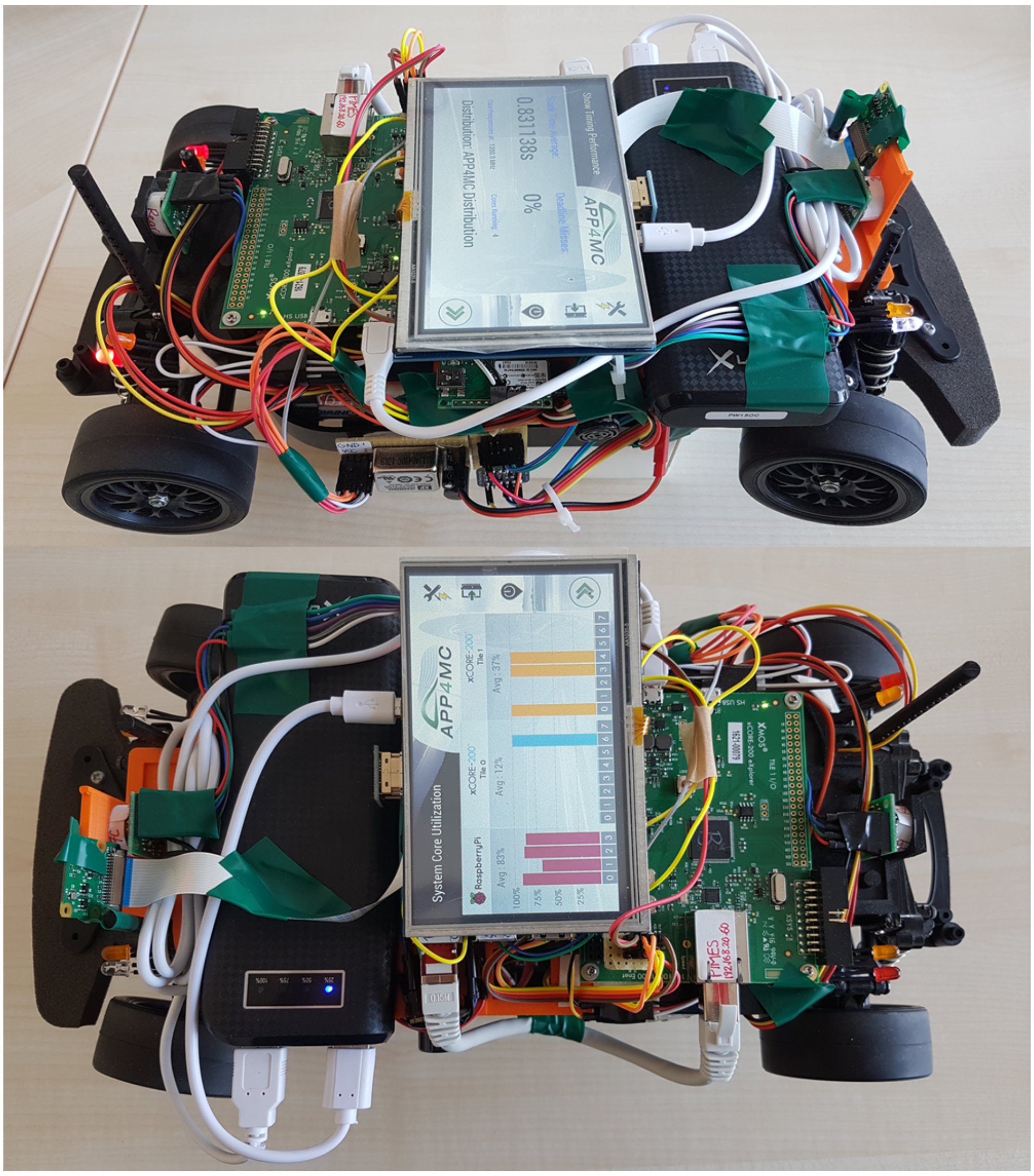


Figure 3.1.: A4MCAR

- Light system task in order to control a light module for certain conditions.
- Ethernet server task to maintain a TCP server for data reception and transmission from the high level module.
- TCP task and several other ethernet configuration related tasks to configure the ethernet module (PHY) drivers and establish proper TCP connection.

In order to investigate parallelization outcomes on a GNU/Linux platform and make use of high level features such as web server, image processing and touchscreen interface high-level module is introduced to the system. The high-level module is designed so that it can

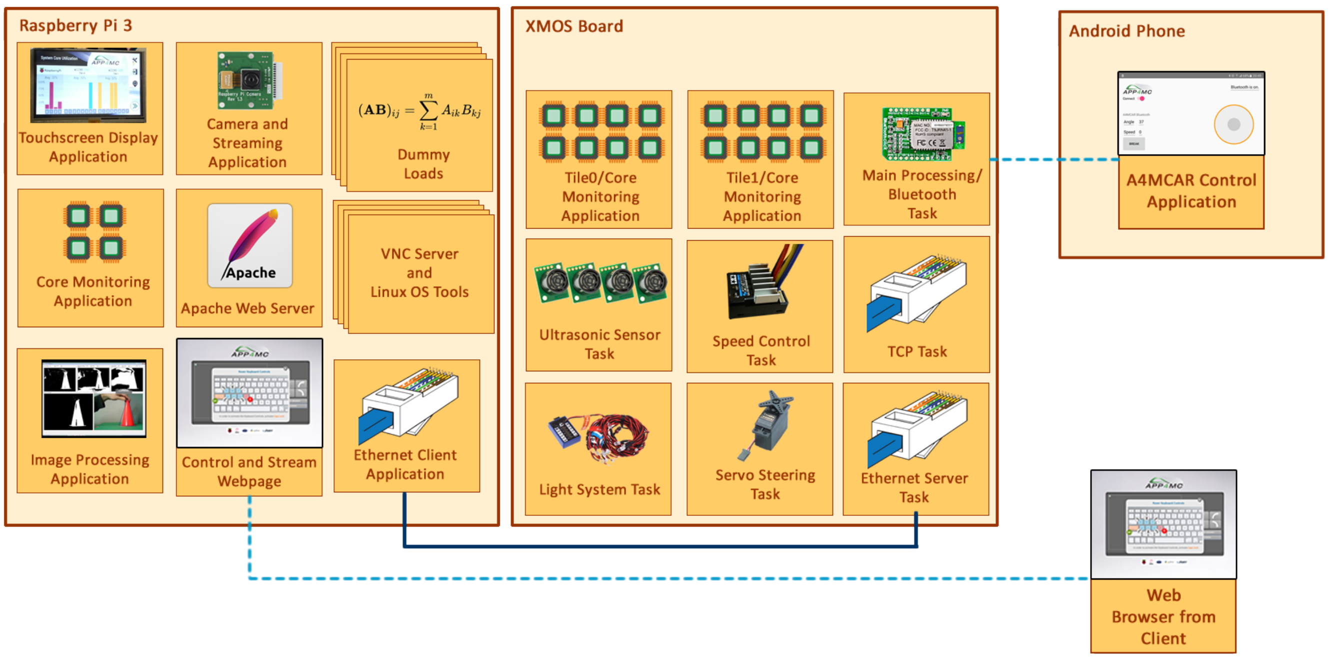

Raspberry Pi 3
Touchscreen Display Application
Camera and Streaming Application
$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj}$
Dummy Loads
Core Monitoring Application
Apache
Apache Web Server
VNC Server and Linux OS Tools
Image Processing Application
Control and Stream Webpage
Ethernet Client Application
XMOS Board
Tile0/Core Monitoring Application
Tile1/Core Monitoring Application
Main Processing/ Bluetooth Task
Ultrasonic Sensor Task
Speed Control Task
TCP Task
Light System Task
Servo Steering Task
Ethernet Server Task
Android Phone
A4MCAR Control Application
Web Browser from Client

Figure 3.2.: Applications developed and/or maintained for A4MCAR

communicate with the low-level module over TCP in order to send driving information and retrieve core information from the low-level module. It is important to mention that high-level module uses Raspberry Pi 3 in order to achieve high level tasks using a robust Debian-based OS, namely Raspbian. Although the features of the high-level module is illustrated in the Figure 3.2, a full feature list can be given as follows:

- Core monitoring application that calculates the average core utilization on each core.
- Image processing application that helps to find a traffic cone.
- Apache Web Server that is used to host a web page which shows average core usage, show Raspberry Pi 3 camera (Raspicam) stream and helps to drive the A4MCAR via web page controls.
- Ethernet client application that handles data transmission and reception to and from server using file operations and data parsing.
- Camera and streaming application that starts the Raspicam and maintains the stream using configuration parameters such as resolution, quality, frame rate, port and etc.
- A webpage which is used for driving the A4MCAR as well as display core utilization on each core and Raspicam stream.
- A touchscreen display application which is used for displaying all cores and their utilization, starting and killing all applications on the high-level module, allocation of processes for the high-level module to cores dynamically, visualization of timing related

performance indicators such as average slack time and deadline misses percentage, selecting different distributions, and configuration of the IP addresses on the high-level module.

- Dummy load processes and a dummy software graph that perform random matrix multiplication in order to investigate performance indicators in full utilization.
- Several Linux processes that run the Linux OS kernel and a VNC server process that provides PC connection via SSH connection.

3.1.2. Infrastructure

The processing infrastructure of the A4MCAR is divided into two modules: Low-level module and high-level module. The low-level module uses a multi-core development kit XMOS xCore-200 eXplorerKIT, whereas high-level module uses a Raspberry Pi 3 which are both shown in Figure 3.3.

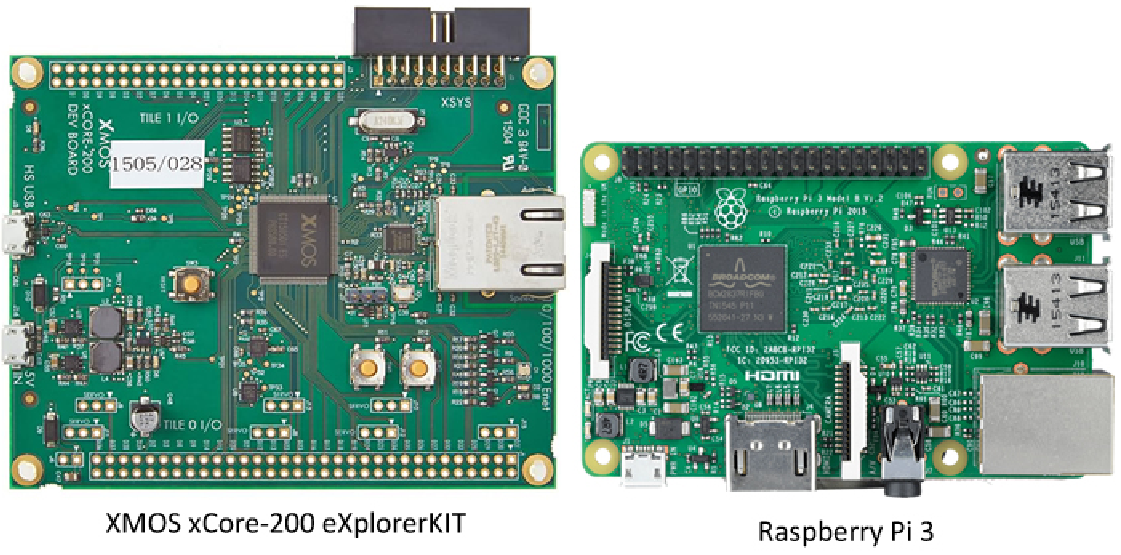


Figure 3.3.: Development boards used in A4MCAR

3.1.2.1. Low-level Infrastructure

The XMOS xCore-200 eXplorerKIT features XEF216-512, a powerful multi-core microcontroller that provides sixteen 32-bit logical cores that are divided into tiles [5], which are identical units that contain a processing unit, cache memory and a switch mechanism [35]. The XMOS xCore-200 eXplorerKIT contains two tiles with eight logical cores in each tile. It is important to add that logical cores of the eXplorerKIT provides 2000 MIPS (Million Instructions per Second) and 512 KB SRAM along with up to 500 MHz clock speed. The specified performance values are considered to be relatively powerful compared to regular microcontrollers. While the processing power and cache memory of its two tiles are identical, ports on each tile have access to different peripherals located on the board. With 53 high-performance GPIOs, the XMOS xCore-200 eXplorerKIT features a 100/1000Mbps Ethernet module, a high speed USB interface, a 3D accelerometer, a 3-axis gyroscope, and six servo interfaces which make the kit useful in a wide variety of applications that include robotics, automotive, signal processing and communication applications [5].

As the name of the development kit suggests, XEF216-512 uses the XMOS' xCore-200 architecture. An illustration of the xCore-200 architecture is given in Figure 3.4.

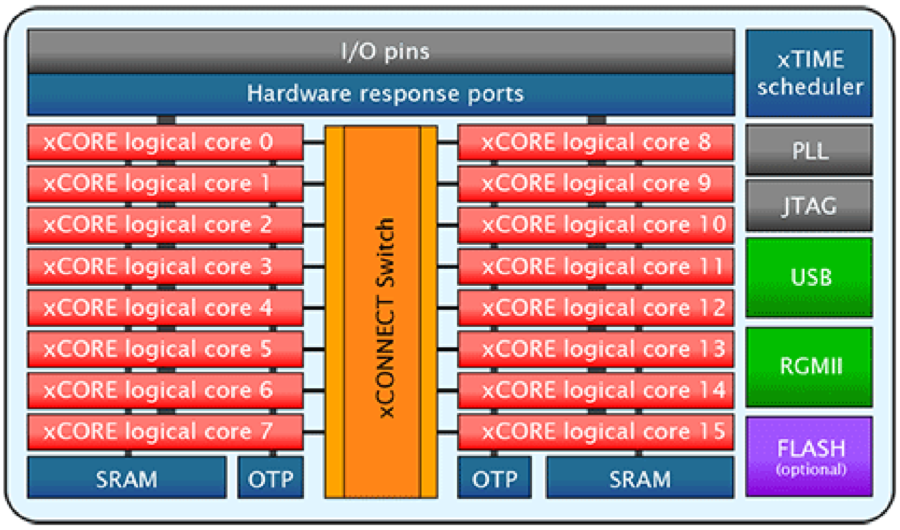


Figure 3.4.: Illustration of XMOS' xCore-200 Architecture [36]

In the xCORE-200 architecture, each core uses the memory of the tile it belongs to and logical cores communicate using a high-speed network. Thus, channels which achieve task communication are linked to other cores via the **xCONNECT Switch**. While this is the case for tasks that are distributed to seperate cores, for tasks that are placed in the same core, the **xTIME Scheduler** automatically schedules tasks by synchronizing events. The xTIME Scheduler works similar to the RTOS schedulers in traditional microcontrollers and uses the **Round-robin scheduling method** [37] [38] which is a simple and starvation-free scheduling technique that gives each task equal time slices and disregards priorities in order to schedule processes or tasks. Round-robin scheduling is widely used in operating systems [38].

In the xCORE architecture, the synchronization of task communication is handled by events rather than ISRs (Interrupt Service Routines) compared to a traditional microcontroller. Each xCORE tile is connected to hardware ports and thereby pins which can be driven high and low in order to drive electrical peripherals. xCORE tiles are also connected to an OTP (One Time Programmable Memory) and a SRAM (Static Random Access Memory). While OTP is used for code locking features, SRAM serves as a memory where the instructions and variables are located [37].

Since the xCORE features multiple cores unlike a traditional microcontroller, it should be clearly understood that the task interruption is not present in xCORE. This is illustrated delicately in the Figure 3.5 [36]. If not stated otherwise in an xCORE application, all the tasks are placed to different logical cores. This means that all the tasks are executed completely parallel in hardware. When the tasks are shared in a core, then the multi-tasking features of the XMOS are invoked and parallelized just like in an RTOS from traditional microcontrollers [36].

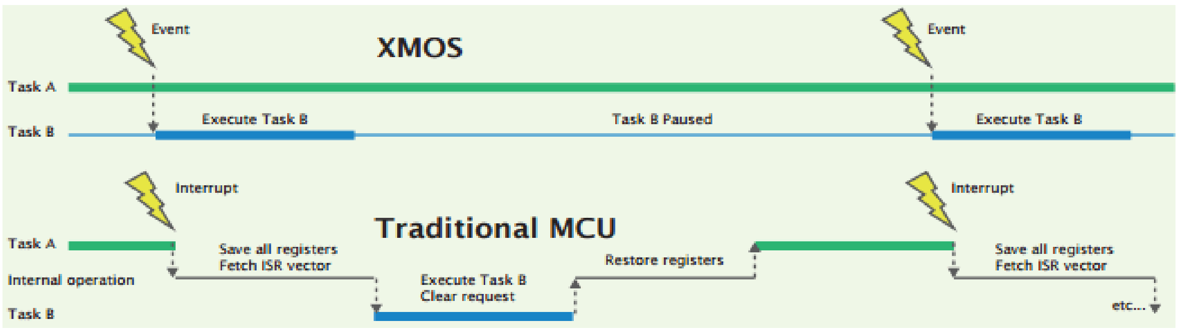


Figure 3.5.: XMOS vs Traditional Microcontroller [36]

Most of the traditional microcontrollers including xCORE microcontrollers nowadays feature pipelining mechanism. The **instruction pipeline** is a set of data processing elements connected in series, where the output of one element is the input of the next one [39]. Via instruction pipelining, processors make use of the stages in order to use the clock to its full performance in order to reduce the time taken to execute instructions. This mechanism is also present in most of the XMOS processors with five stages. How instruction pipelining mechanism achieves faster instruction execution is illustrated in the Figure 3.6 [36].

Traditionally, XMOS based microcontrollers are programmed via the xTimeComposer, which is an Eclipse-based software development platform for XMOS based multi-core microcontrollers with integrated features such as simulation, symbolic debugging, tracing, runtime instrumentation, and timing analysis with a static code timing analyzer called **XTA**[36]. As A4MCAR needs to make use of timing and performance values, some tracing tools and XTA has been used during this thesis' development. The xTIMEComposer development environment windows are shown and illustrated in Figure 3.7.
In the Figure 3.7 it is shown that the main development environment consists of the following windows:

- **Project Tree:** This window is used for managing projects and source, include, binary and configuration files within projects.

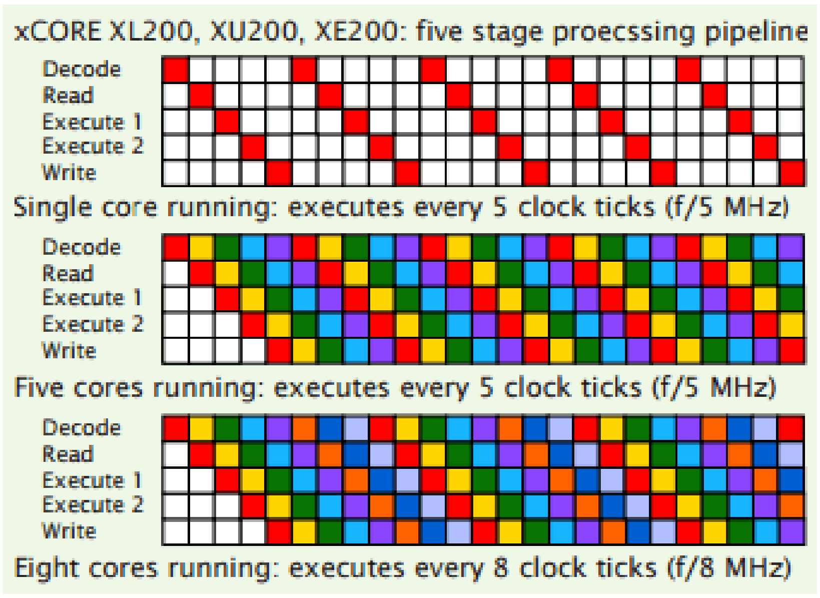
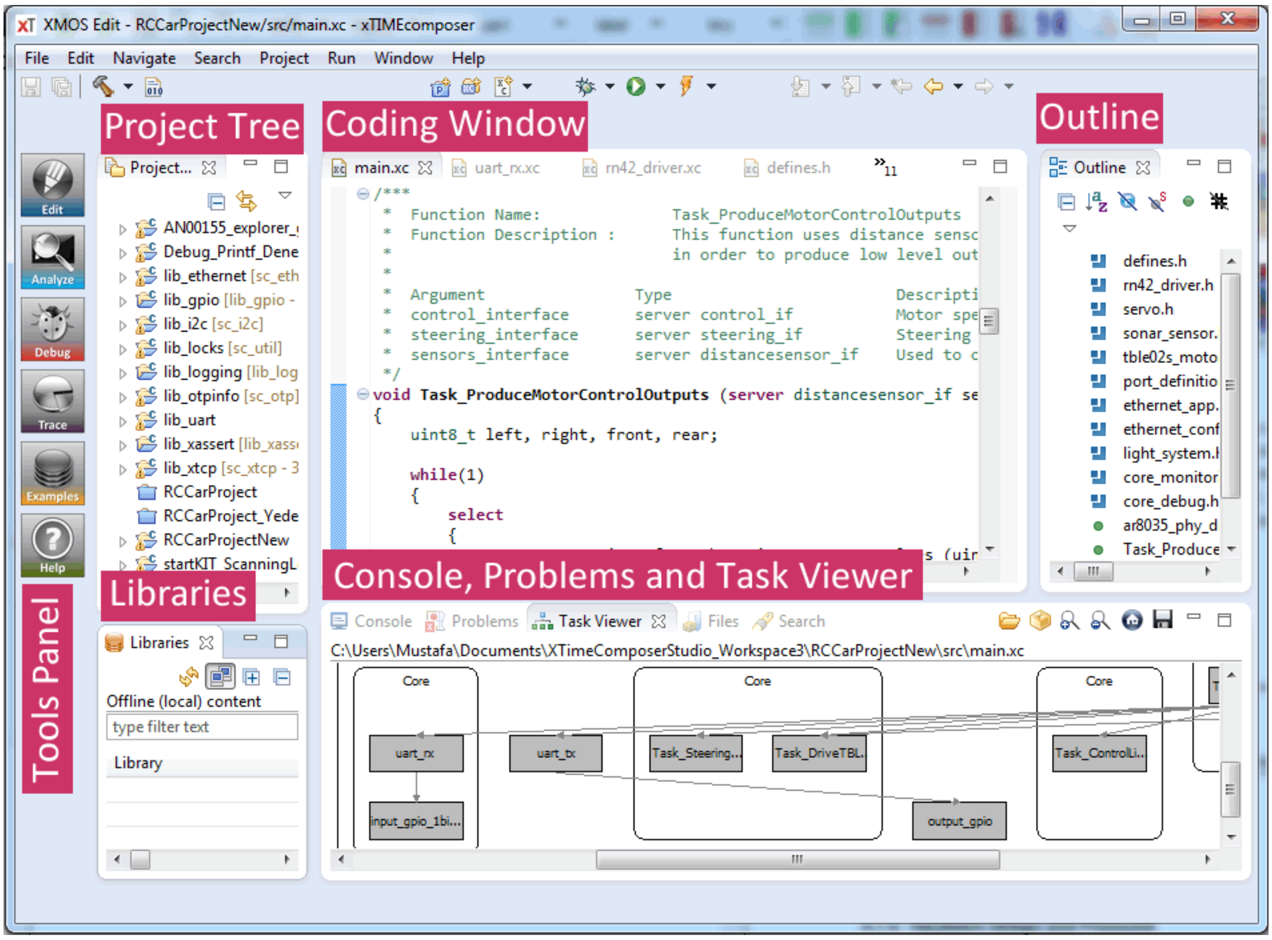

xCORE XL200, XU200, XE200: five stage proecssing pipeline
Decode
Read
Execute 1
Execute 2
Write
Single core running: executes every 5 clock ticks (f/5 MHz)
Decode
Read
Execute 1
Execute 2
Write
Five cores running: executes every 5 clock ticks (f/5 MHz)
Decode
Read
Execute 1
Execute 2
Write
Eight cores running: executes every 8 clock ticks (f/8 MHz)

Figure 3.6.: Pipelining Explained on XMOS [36]



```
/***
 *  Function Name:          Task_ProduceMotorControlOutputs
 *  Function Description :  This function uses distance sensc
 *                          in order to produce low level out
 *
 *
 *  Argument              Type                         Descripti
 *  control_interface     server control_if            Motor spe
 *  steering_interface    server steering_if           Steering
 *  sensors_interface     server distancesensor_if     Used to c
 */
void Task_ProduceMotorControlOutputs (server distancesensor_if se
{
    uint8_t left, right, front, rear;

    while(1)
    {
        select
        {
```

Figure 3.7.: xTIMEComposer 14.2.3 Development Environment Windows

- **Coding Window:** Coding window is used for writing code and placing breakpoints. One can switch between several files by clicking on the tabs located on the top of this window.
- **Console:** The console is used for viewing the building process, verbose and debugging information.
- **Problems:** The problems window is used for seeing warnings and errors that result from the code.
- **Task Viewer:** The task viewer is a special feature that is unique in xTIMEComposer and used to visualize tasks and at which core and tile they are located. The channel and interface connections between tasks are also visualized using this window.
- **Tools Panel:** This window is used in order to switch between several tools that xTIMEComposer provide. *Analyze* and *Debug* tools are widely used in development. Analyze tool opens xTIMEComposer Timing Analyzer (XTA) tool whereas Debug tool is used for traditional debugging using breakpoints.
- **Outline:** The outline window lays out the main elements of a file such as its includes, tasks, objects and so on.
- **Libraries:** The Libraries window can be used in order to search offline and online libraries.

Programming languages which are used for xCORE processors can be listed as C, C++ and xC (C with multicore extensions) [37]. The aforementioned xC language features three main keywords in order to represent task communication. To represent an interface that sends data to another task, the **client** keyword is used whereas if a task is retrieving data from one or many client ports, the receiving interface is named **server**. It is important to mention that server interface receives data by throwing events. Additionally, xC also allows to define function attributes which are **combinable** and **distributable**. The XMOS Programming Guide [40] suggests that combinable tasks are the ones that continuously react to events and they can be combined to have several tasks running on the same logical core. It is added in the XMOS Programming Guide [40] that distributable tasks are not dedicated to only one logical core but they run when required by the tasks connected to them. Furthermore, xC features **timers**, **events**, **guards**, **event priority ordering** in order to help to develop event-based software. These features of xC make multi-core programming easy and robust on xCORE processors.

3.1.2.2. High-level Infrastructure

The high-level processing unit of the A4MCAR is the Raspberry Pi 3 which is a widely used single board computer in uncritical (that does not require strict deadlines) embedded applications. It has a 1.2GHz 64-bit quad-core processor with an ARMv8 architecture, 1GB of RAM, VideoCore IV 3D graphics core and several interfaces such as 40 GPIO pins, 4 USB ports, a HDMI port, ethernet port, an audio jack, a camera interface (CSI), a display interface (DSI), and a micro SD card slot [6]. The reason Raspberry Pi 3 is preferred in embedded systems applications is that it provides excellent connectivity via 802.11n Wireless LAN module, Bluetooth 4.1 module, and Bluetooth Low Energy (BLE) module. Additionally, Raspberry Pi 3 is low-cost which makes it perfect for uncritical embedded applications.

The Raspberry Pi 3 can be booted with the modern Linux-based operating system distributions such as Debian-based Raspbian OS [41] and Ubuntu-based Ubuntu MATE[42] among many others[6]. It should be noted that for the A4MCAR, the Raspbian OS has been used due to its wide software repository and driver support. The fact that the Raspberry Pi 3 functions as a Linux computer helps in developing high-level applications that require the presence of an operating system. The open-source nature of Linux and its software ecosystem provides flexible and traceable software development. For the A4MCAR, the traceability and flexibility features of Raspberry Pi 3 are highly used. Furthermore, a wide variety of programming languages such as C, C++, Java, LISP, Python, Bash, Perl etc. are supported at the Raspberry Pi. In A4MCAR, programming languages such as C, C++, Python, Bash, HTML, JavaScript has been used in order to develop the high-level module.

Linux uses a Round-robin based scheduler called Completely Fair Scheduler (CFS). The goal of CFS is to maximize the overall CPU utilization while also maximizing interactive performance and being fair to all the processes and threads [43]. In order to ensure this fairness, a tree structure of process execution times are created and sorted. Every time, the process which is executed the least time is picked from the tree and executed until *preemption* (interrupted by another software unit) [43].

A brief explanation of the architecture of Linux-based computers and as an extension the architecture of the Raspberry Pi 3 should be given in order to develop applications and understand how applications running on Linux work. With that idea in mind, the high-level overview of the structure of the **Linux kernel** and high-level layers in a Linux system are given in Figure 3.8 [44].

According to Mauerer [44], the kernel is the intermediary level between the hardware and the software that addresses the devices and the components of the system (such as CPU, memory and I/O devices) by passing application requests. While kernel processes requests from user applications, it makes its own decision where data is located and which commands to send to hardware. The kernel is also the instance in a Linux system which shares available

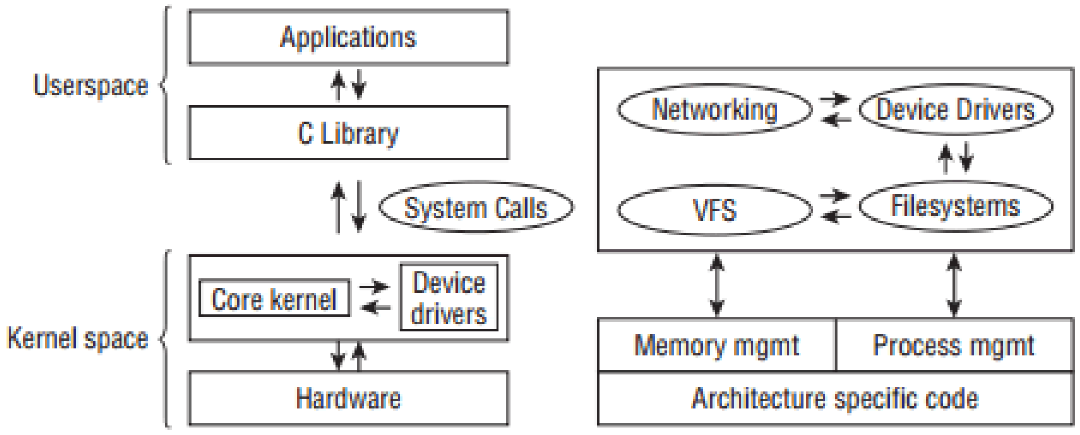


Figure 3.8.: High-level Linux system architecture [44]

resources such as CPU, memory, and network which is why it should be addressed while working with parallel applications.

In Figure 3.8, it is shown that kernel space is not only responsible for accessing device drivers, but it is also responsible for memory and process management. A program under Unix systems (such as Linux) that runs and have own virtual memory is referred to as **processes** and they are scheduled by Linux kernel. The multi-tasking of processes is handled by a mechanism that is called **task-switching** or **context-switching** which ensures that CPU performs according to the scheduled tasks. The concept of **scheduling** in a Linux system is also handled by the kernel and it is the procedure of deciding how CPU time should be shared between existing processes. Additionally, **threads** in a Linux-based computer system are also play a big role for multi-tasking which are also handled by kernel. Threads share the same data and resources but they have different execution paths throughout the program [44].

At the A4MCAR high-level module, it is mostly dealed with processes and investigations according to efficiently parallelize thread and process-based system. In this regard, it is crucial to understand the process life cycle and how kernel schedules processes. This knowledge is described delicately by Mauerer [44] and Ward [45]. Figure in 3.9 depicts an illustration of how the process life cycle basically works [44].

In Figure 3.9, a state machine for processes in a Linux system is given. The states of processes can be listed as Running, Waiting, Sleeping, and Stopped. These states can be explained using following scenerios [44]:

- If a process is being currently executed, the process is in the **Running** state.
- If a process is not being executed because it is waiting for CPU to finish executing another process, it is in **Waiting** state.

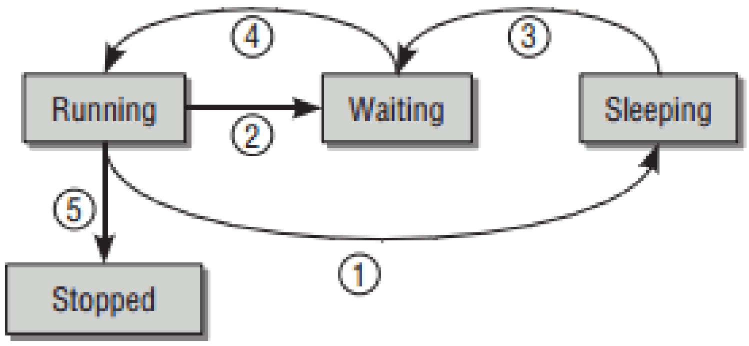
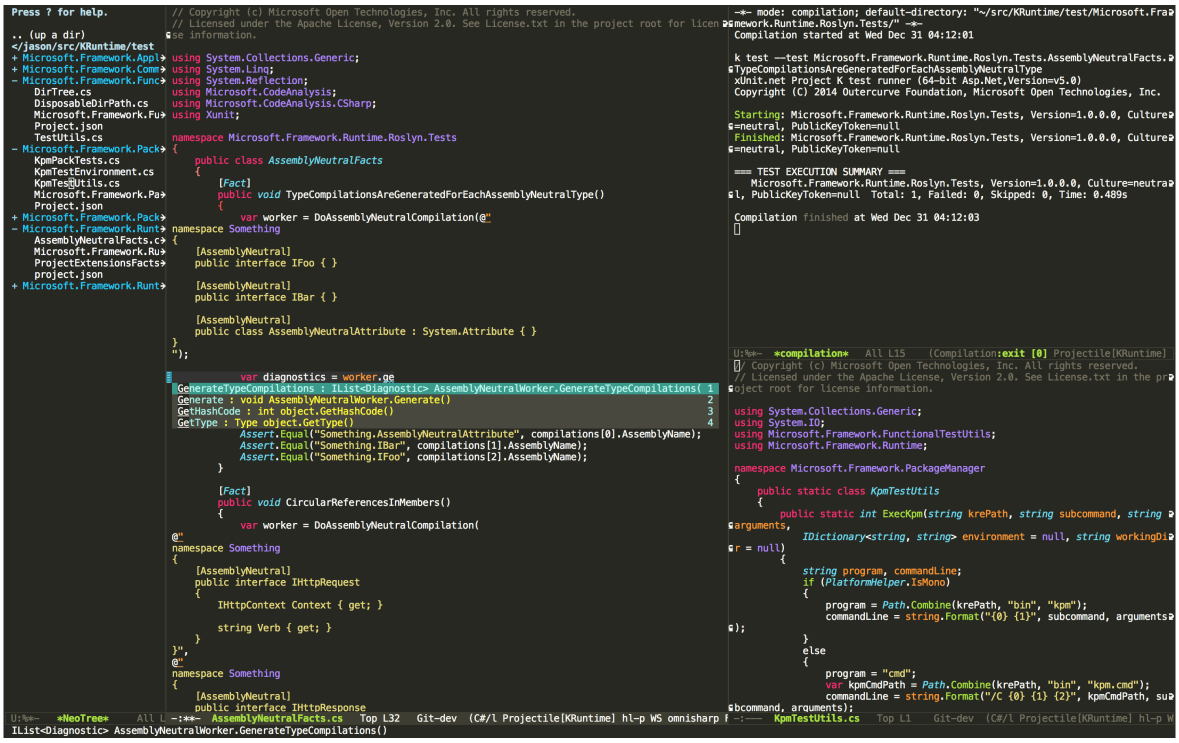


Figure 3.9.: Process Life Cycle in a Linux System [44]

- If a process is waiting for an external event such as a periodic activation or a sporadic activation, it is in the **Sleeping** state. Notice that a transition from Sleeping state to Running state is not possible. A process switches to Waiting state from Sleeping state in order to wait for current process to finish its execution.
- If the user decides to kill (terminate) the application, the process goes into **Stopped** state.
- If a process has been killed but its entries are still alive in the process table, the state of that process is called **Zombie**. Therefore, although not shown, a transition from Running state to Zombie state is also possible.

It is important to note that these states are traceable using Linux kernel access methods which will be explained in Chapter 4 of this thesis along with several other Linux kernel concepts.

The Raspberry Pi is conventionally programmed through Linux shell which is programmed and commanded with the help of the Bash scripting language. There are several editors and compilers introduced for Linux shell in order to help developers write, compile, debug, and trace their applications. Most popular editors involve Nano, Vi, and Emacs which are editors that can run without GNU Graphical User Interface. An alternative way is to use open-source platforms such as Eclipse with correct extensions and plugins. The conventional and standart C compiler for the Unix platform GCC, and the standart Python shell can be accessed using all these compilers. For the sake of demonstration, the Linux shell which is running Emacs is shown in the Figure 3.10. The editor shown not only allows code editing but also allows version control and documentation frameworks interaction. It is important to note that during the development of the A4MCAR, Nano and Emacs editors have been frequently used as Nano provides easiest way to interact with Linux shell and Emacs provides advanced features to compile and debug programs rapidly.



```
Press ? for help.

.. (up a dir)
</jason/src/KRuntime/test
+ Microsoft.Framework.Appl→
+ Microsoft.Framework.Comm→
- Microsoft.Framework.Func→
    DirTree.cs
    DisposableDirPath.cs
    Microsoft.Framework.Fu→
    Project.json
    TestUtils.cs
- Microsoft.Framework.Pack→
    KpmPackTests.cs
    KpmTestEnvironment.cs
    KpmTestUtils.cs
    Microsoft.Framework.Pa→
    Project.json
+ Microsoft.Framework.Pack→
- Microsoft.Framework.Runt→
    AssemblyNeutralFacts.c→
    Microsoft.Framework.Ru→
    ProjectExtensionsFacts→
    project.json
+ Microsoft.Framework.Runt→
```

```
// Copyright (c) Microsoft Open Technologies, Inc. All rights reserved.
// Licensed under the Apache License, Version 2.0. See License.txt in the project root for licen
se information.

using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Xunit;

namespace Microsoft.Framework.Runtime.Roslyn.Tests
{
    public class AssemblyNeutralFacts
    {
        [Fact]
        public void TypeCompilationsAreGeneratedForEachAssemblyNeutralType()
        {
            var worker = DoAssemblyNeutralCompilation(@"
namespace Something
{
    [AssemblyNeutral]
    public interface IFoo { }

    [AssemblyNeutral]
    public interface IBar { }

    [AssemblyNeutral]
    public class AssemblyNeutralAttribute : System.Attribute { }
}
");

            var diagnostics = worker.ge
 GenerateTypeCompilations : IList<Diagnostic> AssemblyNeutralWorker.GenerateTypeCompilations( 1
 Generate : void AssemblyNeutralWorker.Generate()                                             2
 GetHashCode : int object.GetHashCode()                                                       3
 GetType : Type object.GetType()                                                              4
            Assert.Equal("Something.AssemblyNeutralAttribute", compilations[0].AssemblyName);
            Assert.Equal("Something.IBar", compilations[1].AssemblyName);
            Assert.Equal("Something.IFoo", compilations[2].AssemblyName);
        }

        [Fact]
        public void CircularReferencesInMembers()
        {
            var worker = DoAssemblyNeutralCompilation(
@"
namespace Something
{
    [AssemblyNeutral]
    public interface IHttpRequest
    {
        IHttpContext Context { get; }

        string Verb { get; }
    }
}",
@"
namespace Something
{
    [AssemblyNeutral]
    public interface IHttpResponse
U:%*-   *NeoTree*     All L -:**-  AssemblyNeutralFacts.cs   Top L32   Git-dev   (C#/l Projectile[KRuntime] hl-p WS omnisharp F
IList<Diagnostic> AssemblyNeutralWorker.GenerateTypeCompilations()
```

```
-*- mode: compilation; default-directory: "~/src/KRuntime/test/Microsoft.Fra
mework.Runtime.Roslyn.Tests/" -*-
Compilation started at Wed Dec 31 04:12:01

k test --test Microsoft.Framework.Runtime.Roslyn.Tests.AssemblyNeutralFacts.
TypeCompilationsAreGeneratedForEachAssemblyNeutralType
xUnit.net Project K test runner (64-bit Asp.Net,Version=v5.0)
Copyright (C) 2014 Outercurve Foundation, Microsoft Open Technologies, Inc.

Starting: Microsoft.Framework.Runtime.Roslyn.Tests, Version=1.0.0.0, Culture
=neutral, PublicKeyToken=null
Finished: Microsoft.Framework.Runtime.Roslyn.Tests, Version=1.0.0.0, Culture
=neutral, PublicKeyToken=null

=== TEST EXECUTION SUMMARY ===
   Microsoft.Framework.Runtime.Roslyn.Tests, Version=1.0.0.0, Culture=neutra
l, PublicKeyToken=null  Total: 1, Failed: 0, Skipped: 0, Time: 0.489s

Compilation finished at Wed Dec 31 04:12:03

U:%*-  *compilation*   All L15    (Compilation:exit [0] Projectile[KRuntime]
// Copyright (c) Microsoft Open Technologies, Inc. All rights reserved.
// Licensed under the Apache License, Version 2.0. See License.txt in the pr
oject root for license information.

using System.Collections.Generic;
using System.IO;
using Microsoft.Framework.FunctionalTestUtils;
using Microsoft.Framework.Runtime;

namespace Microsoft.Framework.PackageManager
{
    public static class KpmTestUtils
    {
        public static int ExecKpm(string krePath, string subcommand, string 
arguments,
            IDictionary<string, string> environment = null, string workingDi
r = null)
        {
            string program, commandLine;
            if (PlatformHelper.IsMono)
            {
                program = Path.Combine(krePath, "bin", "kpm");
                commandLine = string.Format("{0} {1}", subcommand, arguments
);
            }
            else
            {
                program = "cmd";
                var kpmCmdPath = Path.Combine(krePath, "bin", "kpm.cmd");
                commandLine = string.Format("/C {0} {1} {2}", kpmCmdPath, su
bcommand, arguments);
-:---  KpmTestUtils.cs   Top L1   Git-dev  (C#/l Projectile[KRuntime] hl-p W
```

Figure 3.10.: Linux Shell running Emacs

3.1.3. Hardware Design, Sensors and Protocols

One should care for a robust hardware design in order to avoid having software problem and safety related issues. In A4MCAR, a number of modules have been used alongside development kits in order to provide utility to the demonstrator. On the low-level module side, a light system, an RN-42 Bluetooth module, four SRF-02 ultrasonic sensors, a servo motor, a TBLE-02S electronic speed controller have been used. The high-level module side however only is connected to a touchscreen display and a Raspicam (Raspberry Pi camera). The overview of hardware connections, used protocols and device architectures is given in Figure 3.11.

The system uses various communication protocols, shown in Figure 3.11 in order to interact with sensors, actuators and utility devices. The communication protocols and associated devices used for the A4MCAR could be listed as follows:

- **PWM:** In order to interact with the servo motor, the TBLE-02S electronic speed controller, and light system, Pulse Width Modulation (PWM) signaling has been used. PWM is a type of modulated digital signal used mostly in control applications [46]. By describing how much a signal is high and low with respect to time, the *duty cycle* is measured which is given in percentage. One could observe the signal illustration given in 3.12 to see how commonly used duty cycles look like. In a control circuitry, by

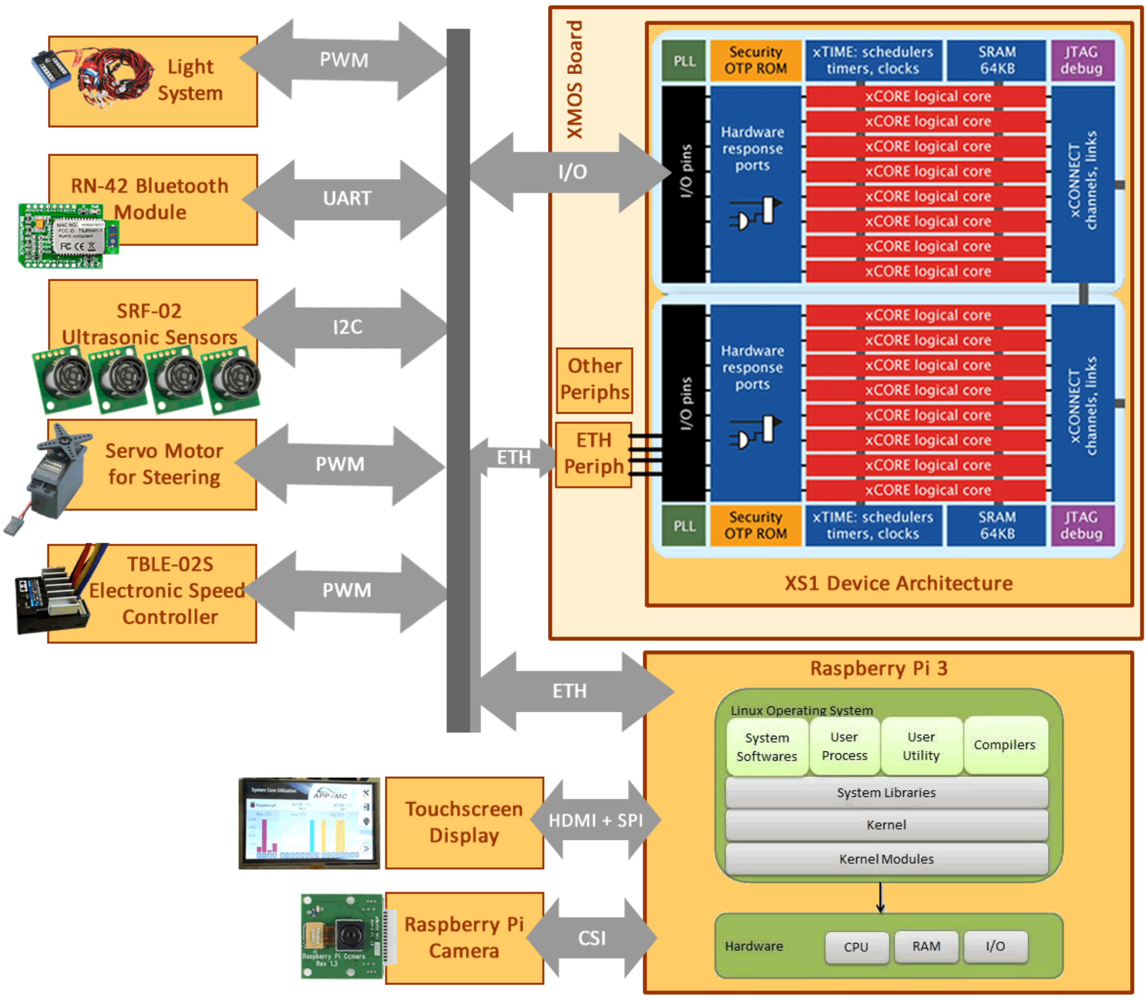


Figure 3.11.: Hardware overview of A4MCAR

achieving various duty cycles, dimming a light or controlling the direction or speed of a motor is possible [46].

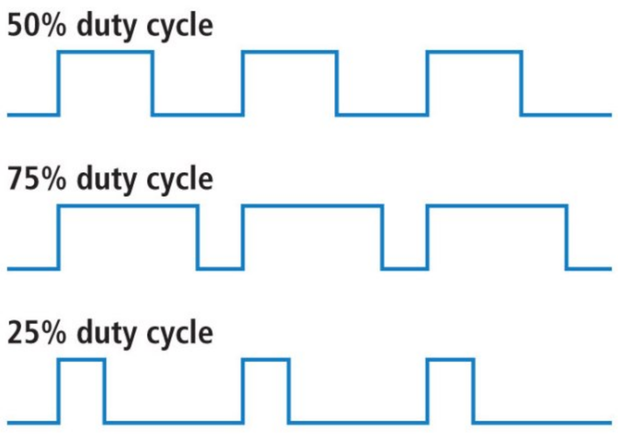


Figure 3.12.: Duty cycle example in pulse width modulation [46]

- **UART:** The RN-42 is a master-slave configurable bluetooth module (shown in Figure 3.11) that is programmed using AT commands via UART. In A4MCAR, a RN-42 bluetooth module is used in order to interact with the bluetooth of Android devices. UART (Universal Asynchronous Receiver and Transmitter) is a communication protocol that achieves simple communication of two equivalent nodes. UART is a half-duplex and asynchronous serial protocol that doesn't communicate using a clock. Half-duplex nature of UART makes it so that transmitting and receiving lines can not be used simultaneously. It became a universal format because it is being used in telephone lines and USB ports of computers for decades. Number of bits transmitted or received per second is referred to as baud rate and it is standardized to values such as 9600, 14400, 19200, 38400, 57600, and 115200. A basic UART data packet is given in Figure 3.13. The figure should depict that the basic format usually contains 6 to 8 data bits and start and stop bits to mark start and stop of the data packet. It should be noted that there are various formats with different sizes [47].

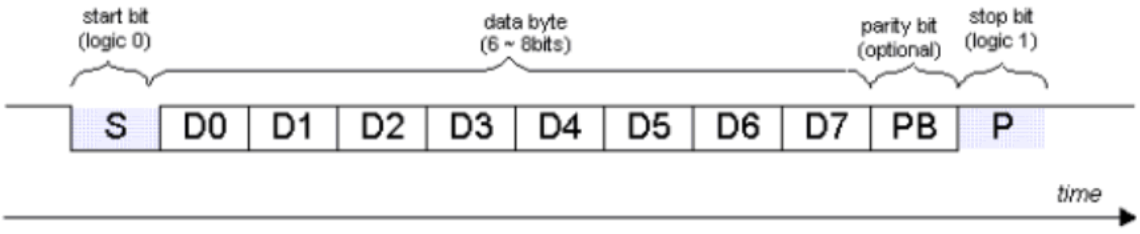


Figure 3.13.: Simple UART data packet [47]

- **I^{2}C:** The proximity sensor network of A4MCAR that consists of four SRF-02 sonar sensors uses I^{2}C communication protocol in order to address devices in the network and obtain distance information in centimeters. I^{2}C (Inter integrated circuit) is a communication protocol that is intended for short distances to handle the communication of multiple slave units with one or multiple master units. Its advantage is that it uses only two wires in order to handle communication between many devices. Compared to the very similar serial communication protocol SPI, I^{2}C can support a multi-master system with up to 1008 slave devices. I^{2}C chips consist of two signals: clock signal SCL and data signal SDA. Since signals are open drain, each signal must have a pull-up resistor. The communication is handled by sending the address of the register and the data to be sent in order to write into the registers of the I^{2}C chips. A basic frame with 7 address bits and 8 data bits is given in Figure 3.14 [48].

- **Ethernet/TCP**: Ethernet communication using TCP (Transmission Control Protocol) is a very common method of communation that is applied within the Application, Presentation, and Session layers of the well-known OSI model. It is also a protocol that is

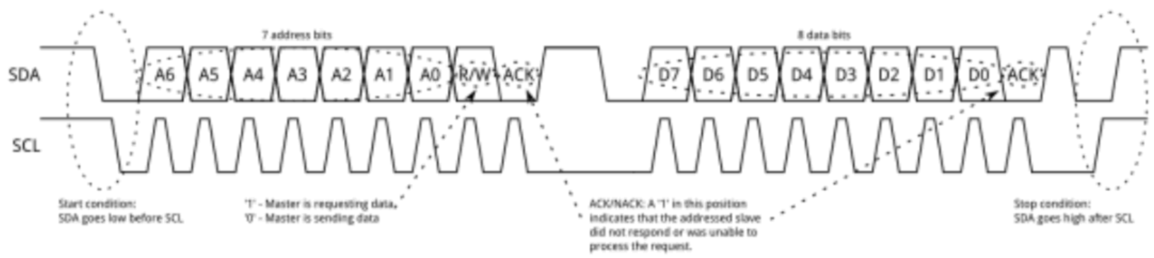

7 address bits
8 data bits
SDA
SCL
A6
A5
A4
A3
A2
A1
A0
R/W
ACK
D7
D6
D5
D4
D3
D2
D1
D0
ACK
Start condition: SDA goes low before SCL
'1' - Master is requesting data, '0' - Master is sending data
ACK/NACK: A '1' in this position indicates that the addressed slave did not respond or was unable to process the request.
Stop condition: SDA goes high after SCL

Figure 3.14.: I2C protocol frame[48]

used for high-speed data transmission to other network devices on the same network segment if used in Telnet mode. Ethernet defines two units of transmission, packet and frame. The frame includes not just the payload of data being transmitted but also the information identifying the physical *Media Access Control (MAC)* addresses of both sender and receiver, VLAN tagging, quality of service information, and error-correction information to detect problems during the transmission [49] [50].

In A4MCAR, a telnet server and client has been implemented using the TCP protocol in order to send and receive data between the high-level and low-level modules. The high-level module is configured as a client, whereas low-level module is configured as the server.

- **SPI**: Just like I^{2}C, SPI (Serial Peripheral Interface) is a communication protocol that is used to send data between processors and small devices such as sensors or displays. In SPI, MOSI, MISO, and SCK lines are available that are two data lines for each direction and a clock line. Additionally, a line of SS (Slave Select) could be used in order to select which slave device in the network is being addressd at that moment. Since SPI does only work with a clock unlike the conventional UART, SPI is a synchronous communication method [51].

  In A4MCAR, SPI is used by the touchscreen kernel drivers in order to get touchscreen controls working. HDMI interface is also used in order to transfer media from the Raspberry Pi to the Touchscreen Display.

- **CSI**: CSI (Camera Interface) is used by third-party applications and it is the interface that is used in order to get RaspiCam working.

In the Figure 3.15, the complete circuit schematics regarding the A4MCAR low-level module is given. Interfacing sensors and devices should be handled accordingly in order to program peripherals with xCORE-200 board properly.

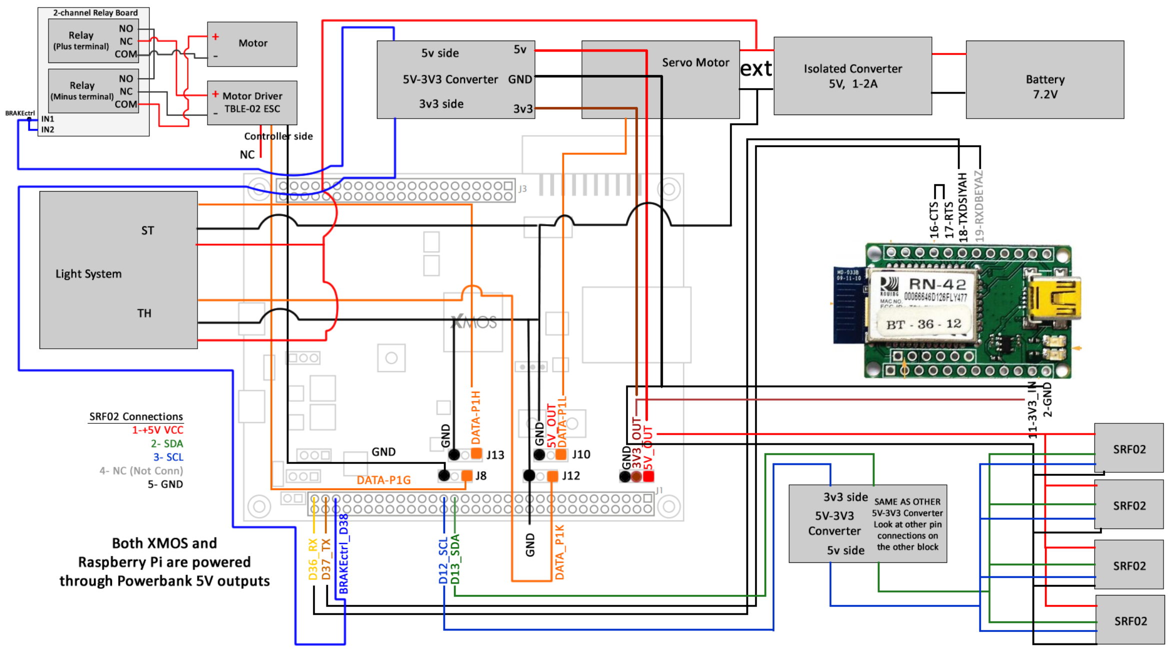


Figure 3.15.: Low-level module schematics of A4MCAR using XMOS xCore-200 eXplorerKIT

3.1.4. Safety and Power

As Figure 3.15 illustrates, there are three units that are introduced in order to get rid of the problems that are related to safety and power. To start with, since it is stated in the XS-1 architecture datasheet [37] that XMOS works typically with 3.3V signals and SRF-02 sensors use 5V signals [52], a 4-channel I^{2}C-safe bi-directional 5V-3.3V Logic Level Converter from Adafruit with model number BSS138 [53] has been used to convert the SDA and SCL lines of the constructed I^{2}C network.

The second issue to solve that occured to low-level having multiple motors connected is the noise and excessive current drain into boards due to motors. Since that could lead to damaged development boards and chips, the solution of using two seperate power lines have been introduced. That is, using a 5V 10000mAh Powerbank to power the development boards xCORE-200 eXplorerKIT and Raspberry Pi 3 using micro-USB connectors, while using an external battery for the motors. That would reduce the noise that occurs in the signal lines since the ground lines of each battery would be isolated. In order to be on the safe side, a 5V 1A rated isolated voltage converter XP Power JCA0605S05 [54] is also used in order to convert 7.2V battery voltage to power servo motor which is typically powered with 5V. Since the datasheet of the XP Power JCA0605S05 suggests that an emission circuit should be constructed, the circuit in Figure 3.16 has been constructed and printed along the isolated converter in order to meet the suggested emission level B [54].

Applying the mentioned solutions, the issues faced regarding the power voltage levels and

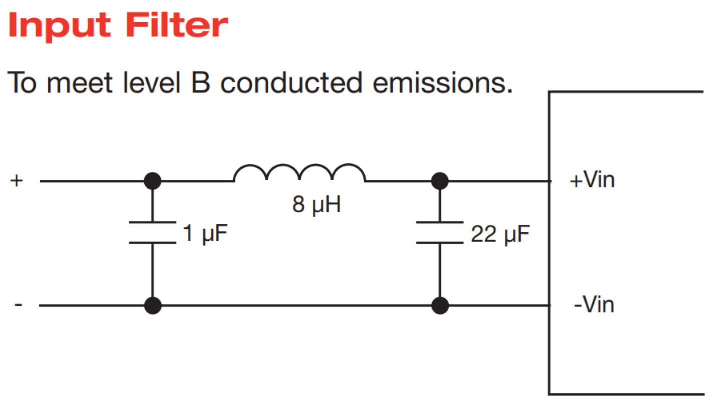


Figure 3.16.: XP Power JCA0605S05 Level B Emission Circuit [54]

excessive current drain have been dealt with and the constructed system safety is ensured.

3.1.5. Mechanical Design

To construct A4MCAR's exterior structure, the RC-Car chassis kit Tamiya TT01-E [55] has been used. The chassis kit consists of several parts and it is a kit that is used in professional RC-Car competitions. Figure 3.17 shows the constructed Tamiya TT01-E chassis kit along with several other equipment that is used to construct the A4MCAR. Since the A4MCAR does not only have basic driving elements but also many other equipments that are related to sensing, processing, and power, the space on the RC-Car was not be enough to hold the extra elements. Therefore, an extension to the existing chassis was needed. To solve this problem, an extension have been designed that is able to hold other elements used in the board with holders and screws.

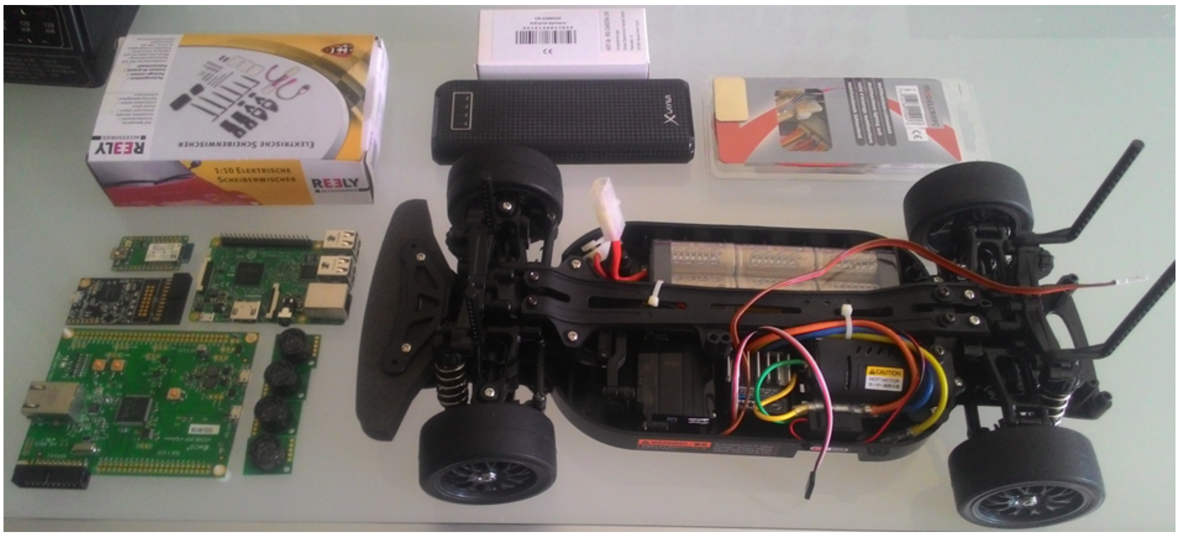


Figure 3.17.: Tamiya TT01-E Chassis and other parts used in A4MCAR

An illustration of the mechanical overview that shows the designed model is shown in 3.18. This mechanical layer has been designed using a 3D model software (Google SketchUp) and the software output with the .STL extension is used for the layer production using a 3D printer called Ultimaker. Using appropriate 1mm to 2mm diameter screws, the constructed body is installed to the main chassis as an extension layer and the other elements are installed on top of this extension layer.

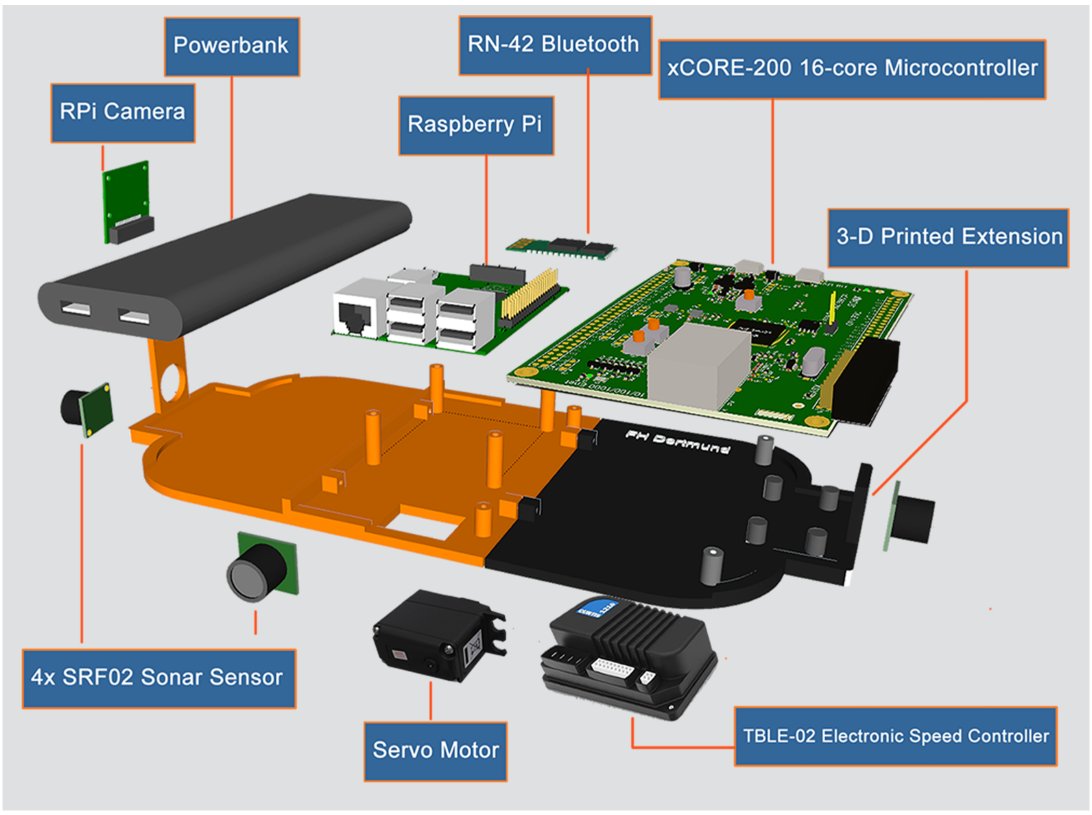


Figure 3.18.: Mechanical overview of the A4MCAR

3.2. Low-Level Module Design and Implementation

3.2.1. Overview

As mentioned in the Section 3.1.2.1, the low-level module software has been implemented on the xCORE-200 eXplorerKIT using the development platform xTIMEcomposer 14.2.3. While developing with xC on xTIMEcomposer, task communication is handled by channels and interfaces. In A4MCAR, for the sake of structured development with defined variable types, interfaces are more commonly used for user-defined tasks. An software design analogy of equating provided interfaces to client interfaces in xC could be made. Similarly, required interfaces could be thought of server interfaces in xC. Using this analogy, the designed software components could be illustrated with a SysML [56] diagram as shown in Figure 3.19.

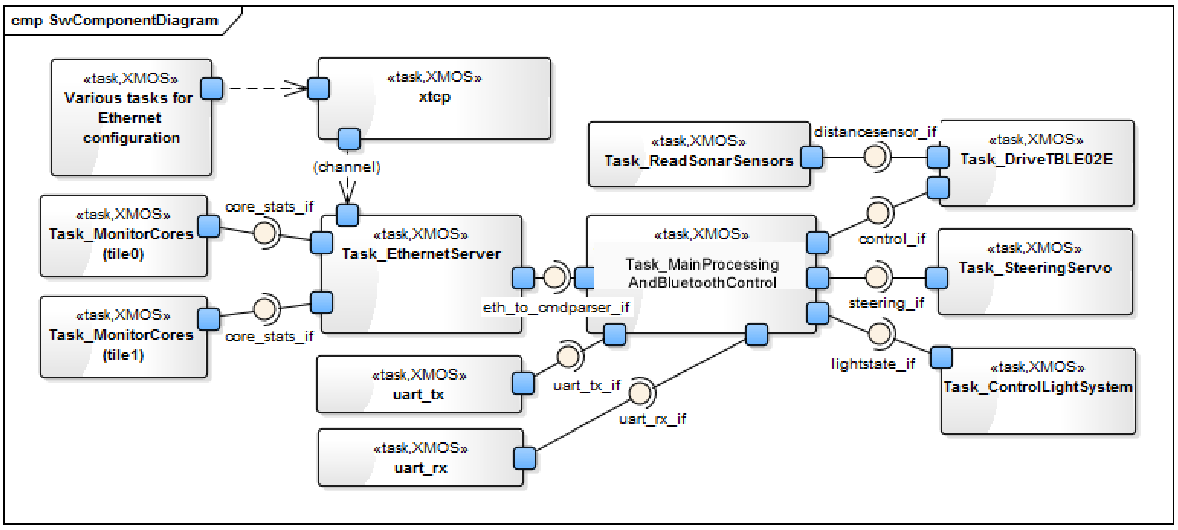

cmp SwComponentDiagram
«task,XMOS» Various tasks for Ethernet configuration
«task,XMOS» xtcp
(channel)
«task,XMOS» Task_MonitorCores (tile0)
core_stats_if
«task,XMOS» Task_EthernetServer
«task,XMOS» Task_MonitorCores (tile1)
core_stats_if
eth_to_cmdparser_if
«task,XMOS» Task_ReadSonarSensors
distancesensor_if
«task,XMOS» Task_DriveTBLE02E
«task,XMOS» Task_MainProcessing AndBluetoothControl
control_if
«task,XMOS» Task_SteeringServo
steering_if
lightstate_if
«task,XMOS» Task_ControlLightSystem
«task,XMOS» uart_tx
uart_tx_if
uart_rx_if
«task,XMOS» uart_rx

Figure 3.19.: Brief block diagram for the developed tasks and interfaces for low-level module

The complete component diagram of the developed software is shown in Figure 3.20.

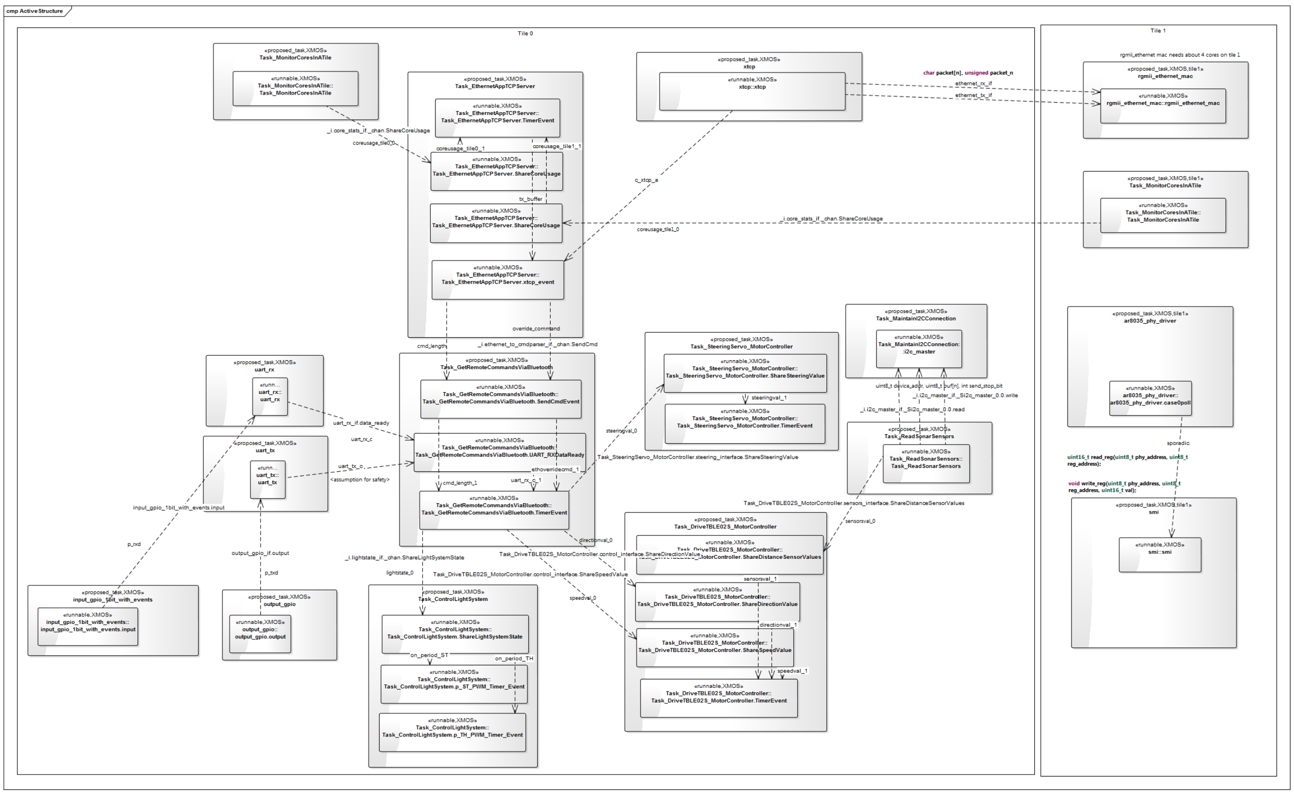


Figure 3.20.: Block diagram for the developed tasks and interfaces for low-level module

In xC, two essential concepts are worthy to explain in order to understand multi tasked development. At first, a task is created and the second is how tasks are connected. A task in xC is nothing but a function that has client and server ports with interfaces. Once all functions are connected using globally instantiated interface variables they start acting as tasks. An example of how a task function is declared and how functions are placed on cores and interconnected are shown in Listing 3.1 and Listing 3.2, respectively.

```
[[combinable]]
void Task_GetRemoteCommandsViaBluetooth(client uart_tx_if uart_tx,
                                        client uart_rx_if uart_rx,
                                        client control_if control_interface,
                                        client steering_if steering_interface,
                                        server ethernet_to_cmdparser_if
                                            cmd_from_ethernet_to_override,
                                        client lightstate_if lightstate_interface);
```

Listing 3.1: An Example Task Decleration in xC

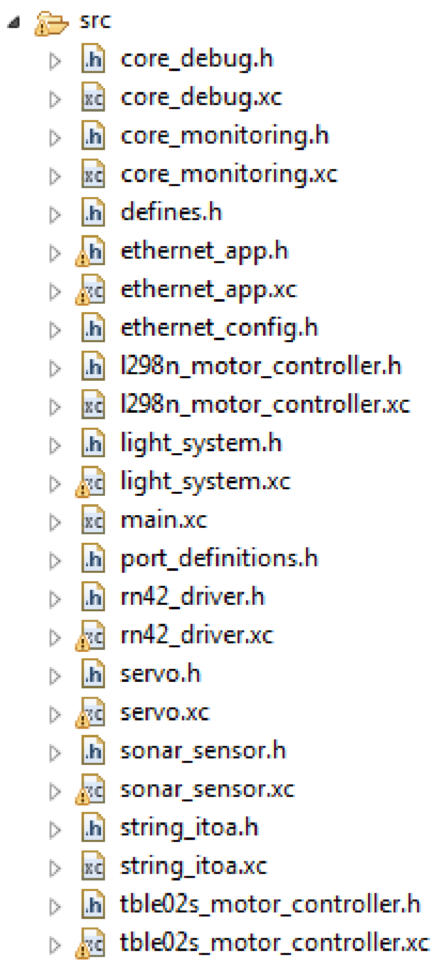
In the code given with the Listing 3.1, it is shown that the function prototype has several arguments such as client and server interfaces. Those interfaces indicate the role of the data communication using the respective interface. When a task function takes client interface as an argument, it means that the task function sends data to that interface, whereas when a task function receives a message using event handles, it is given by the server keyword.

```
1   par {
2       // I2C Task
3       on tile[0] : Task_MaintainI2CConnection(i2c_client_device_instances, 1, PortSCL, PortSDA
            , I2C_SPEED_KBITPERSEC);
4
5       // Motor Speed Controller (PWM) Tasks
6       on tile[0].core[4] :      Task_DriveTBLE02S_MotorController(PortMotorSpeedController,
            control_interface, sensors_interface);
7
8       // Steering Servo (PWM) Tasks
9       on tile[0].core[4] :      Task_SteeringServo_MotorController (PortSteeringServo,
            steering_interface);
10
11      // Core Monitoring Tasks
12      on tile[0]:               Task_MonitorCoresInATile (core_stats_interface_tile0);
13      on tile[1]:               Task_MonitorCoresInATile (core_stats_interface_tile1);
14   }
```

Listing 3.2: An Example of How Tasks are Placed and Interconnected in xC

The Listing in 3.2 shows in which tile and at which core a task is placed. Using the par keyword (given in Line 1), every line of code in that particular code block will be paralellized using the scheduler of xCORE.

All the source and header files that contain task functions and that are developed in this fashion are shown in Figure 3.21.



```
src
  core_debug.h
  core_debug.xc
  core_monitoring.h
  core_monitoring.xc
  defines.h
  ethernet_app.h
  ethernet_app.xc
  ethernet_config.h
  l298n_motor_controller.h
  l298n_motor_controller.xc
  light_system.h
  light_system.xc
  main.xc
  port_definitions.h
  rn42_driver.h
  rn42_driver.xc
  servo.h
  servo.xc
  sonar_sensor.h
  sonar_sensor.xc
  string_itoa.h
  string_itoa.xc
  tble02s_motor_controller.h
  tble02s_motor_controller.xc
```

Figure 3.21.: Full file tree for all the tasks developed for low-level module

3.2.2. Actuation

3.2.2.1. Acceleration

For acceleration and deceleration, a brushless DC motor is controlled by delivering PWM signals to the TBLE02-S Electronic Speed Controller. The task and how it is connected to other tasks can be seen in Figure 3.19. In order to deliver the desired PWM signal, a task that uses timers has been created which is given in Listing 3.3. The created template of the task in Listing 3.3 is not only used for the acceleration task, but also for the other tasks that use PWM signaling to control other devices. In those tasks, it is shown that in order to generate the desired duty cycle, the amount of time for which the output signal must be on and off are calculated (Lines 19 through 38) and the output port is toggled (Lines 39 through

48) accordingly. The port toggling is done inside a timer event (shown in Line 17) that is dynamically delayed given the calculated on and off times.

```
1  [[combinable]]
2  void Task_DriveTBLE02S_MotorController (port p, server control_if control_interface, server
       distancesensor_if sensors_interface)
3  {
4      while(1)
5      {
6          select
7          {
8              //Wait for the direction value
9              case control_interface.ShareDirectionValue (int direction):
10                 direction_val = direction;
11                 break;
12             //Wait for the speed value
13             case control_interface.ShareSpeedValue (int speed):
14                 speed_val = speed;
15                 break;
16             //Calculate PWM periods and apply period within the timer
17             case tmr when timerafter(time) :> void :
18                 tmr :> time;
19                 if (direction_val == FORWARD){
20                        if (speed_val == 0){
21                            on_period = TBLE02S_FWD_MINSPEED_PULSE_WIDTH;
22                        }else if (speed_val > 99){
23                            on_period = TBLE02S_FWD_MAXSPEED_PULSE_WIDTH;
24                        }
25                        else{
26                            on_period = (TBLE02S_FWD_MINSPEED_PULSE_WIDTH - ((
                                  TBLE02S_FWD_MINSPEED_PULSE_WIDTH -
                                  TBLE02S_FWD_MAXSPEED_PULSE_WIDTH) * (speed_val/100.0)));
27                        }
28                    off_period = overall_pwm_period - on_period;
29                 }else if (direction_val == REVERSE){ //Reverse speed 0-100 mapping to on
                       period
30                    if (speed_val == 0){
31                        on_period = TBLE02S_REV_MINSPEED_PULSE_WIDTH;
32                    }else if (speed_val > 99){
33                        on_period = TBLE02S_REV_MAXSPEED_PULSE_WIDTH;
34                    }else{
35                        on_period = (TBLE02S_REV_MINSPEED_PULSE_WIDTH + ((
                              TBLE02S_REV_MAXSPEED_PULSE_WIDTH - TBLE02S_REV_MINSPEED_PULSE_WIDTH)
                               * (speed_val/100.0)));
36                    }
37                    off_period = overall_pwm_period - on_period;
38                 }
39                 //PWM Port Toggling
40                  if(port_state == 0){
41                      p <: 1;
42                      port_state = 1;
43                      time += on_period; //Extend timer deadline
44                  }else if(port_state == 1){
45                      p <: 0;
46                      port_state = 0;
47                      time += off_period; //Extend timer deadline
48                  }
49                  break;
50          }
51      }
52  }
```

Listing 3.3: Created PWM signaling template

The desired PWM pulse widths are taken from the TBLE02-S Electronic Speed Controller manual and the overall PWM period has been set to 20ms which is the standard period for most of the controllers. Since the motor speed can be very high but is not desired to that extent in our application, the pulse widths are manipulated in order to reach lower speeds in full force. The interface `control_if` (Line 9 and Line 13) is used that delivers a number between 0-100 in order to express speeding information while also delivering a direction value which is either FORWARD (0) or REVERSE (1). With this information, the developed task is able to control the acceleration of the A4MCAR. Additionaly, the acceleration task is modified in order to control acceleration using the proximity sensor inputs for safety. Minimum safest front distance is set to 50 centimeters.

The acceleration task and how data is transferred between other tasks can be seen in the software component diagram in Figure 3.19 with the acceleration task having the function name `Task_DriveTBLE02E`.

3.2.2.2. Steering

Steering of the A4MCAR is done with the help of a Servo motor that is also controlled with PWM signaling. As mentioned in Section 3.2.2.1, the tasks that are related to PWM use the template from Listing 3.3. For steering the interface `steering_if` is used which is set to 0 for very left and 100 for very right positions. Additionally, the following two changes that are made to the acceleration task (from Listing 3.3) could be listed as follows:

- The pulse widths are altered in order to conform a servo motor's behavior which is usually 1.5ms pulse for stationary position, 1-1.5ms for left steering and 1.5-2.0ms for right steering. For the A4MCAR, these values are set to 1.3-1.5ms for left steering and 1.5-1.75ms for right steering in order to get rid of the issue that the servo motor is turning more than its holding platform could handle what may result in damaging its gears.

- On and off periods are calculated differently as compared to the Listing 3.3. The Listing in 3.4 shows the calculation of the periods using the pulse width values.

```
if  (steering == 0){
   on_period = STEERINGSERVO_PWM_MAXRIGHT_PULSE_WIDTH;}
else if (steering > 100){
   on_period = STEERINGSERVO_PWM_MAXLEFT_PULSE_WIDTH;
}else{
   on_period = (STEERINGSERVO_PWM_MAXRIGHT_PULSE_WIDTH - ((
       STEERINGSERVO_PWM_MAXRIGHT_PULSE_WIDTH - STEERINGSERVO_PWM_MAXLEFT_PULSE_WIDTH)
       * (steering/100.0)));
}
off_period = overall_pwm_period - on_period;
```

Listing 3.4: Calculation of on and off times for servo control

The steering task and how data is transferred between other tasks can be seen in the software component diagram in Figure 3.19 with the steering task having the function name `Task_SteeringServo`.

3.2.2.3. Braking

Since the motor that is used in A4MCAR is a brushed motor, it does not come with braking features. Therefore, a braking mechanism is needed. For this purpose, a 2-channel relay board from Sainsmart (shown in Figure 3.22) is applied to the A4MCAR to short circuit the terminals of the motor when braking is required. In Figure 3.23, the circuit portion that is controlling the operation of the brake is given.

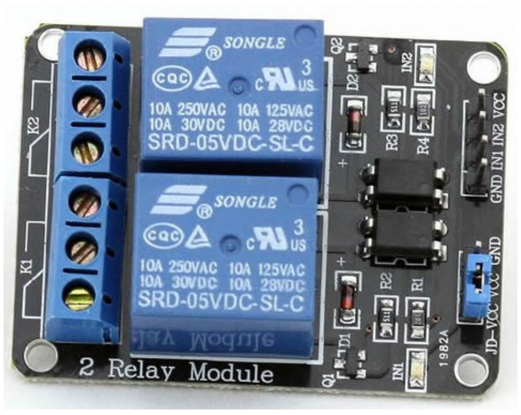


Figure 3.22.: Two-channel relay board that is used for braking

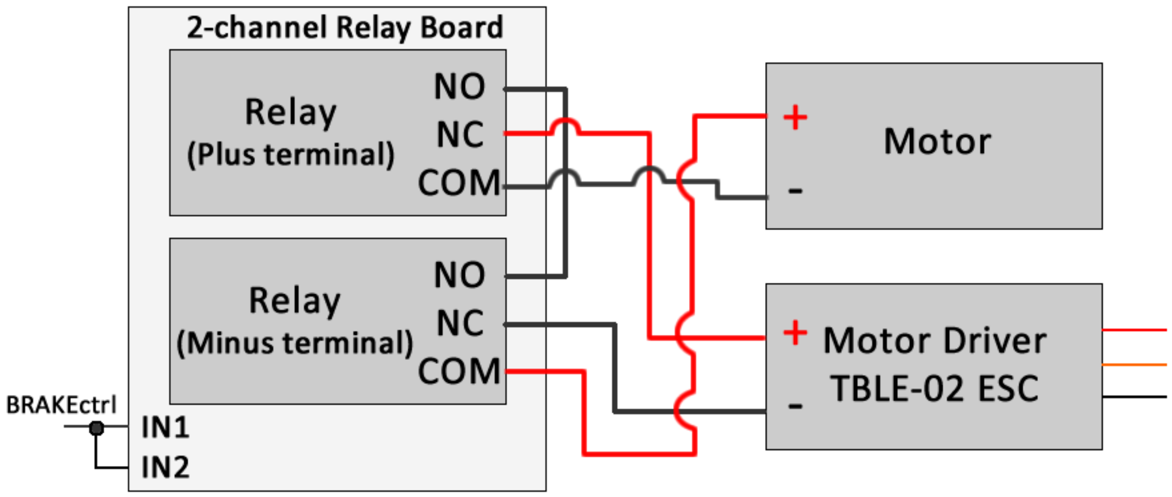


Figure 3.23.: Relay circuit to control braking

Normally, the terminals of the motor and the motor driver (Electronic Speed Controller TBLE-02) are connected together. Using the switching mechanism that is provided by the relays, motor terminals are short circuited when the relays are activated.

Relays have three essential input or output signals that are used in operation. Those signals involve IN, NO, NC, and COM. With simple switching in mind, the operation of the relay can be described as follows. When the IN signal is low, the COM is short circuited with NO whereas when the IN signal is high, the COM signal is short circuited to NC signal.

By using this mechanism and using the COM signal output, the motor is supplied by the motor driver signal normally and motor terminals are short circuited when braking is required. The control has been applied by giving the same input to IN1 and IN2 signals, IN signals for each relay, from the xCORE-200 eXplorerKIT. The integration of the braking mechanism to the system is done at the acceleration task, in which when the received speed is zero, brake is activated. Since the relay works with only 5V input signal for the IN terminal, BRAKEctrl signal, given in Figure 3.23 is connected to the xCORE-200 eXplorerKIT through a 5V to 3.3V converter, which is also shown in the schematics given in the Figure 3.15.

3.2.3. Proximity Sensing

As stated in the Section 3.2.1, the proximity sensing is handled via four SRF-02 ultrasonic sensors connected to an I^{2}C network. In the software, *lib_i2c* from XMOS is used in order to handle communication with the peripheral.

The pseudo version of the proximity sensing task is given in Listing 3.5. The proximity sensing task is a periodic task that polls individual ultrasonic sensors using their respective addresses (Line 12 and Line 13) in order to obtain what is the distance perceived by front, rear, left, and right sensors. In order to handle this in a modular manner, device addresses are placed in the header file of the proximity sensing task source files. Furthermore, the task that maintains I^{2}C communication alongside proximity sensing task is seperated and the two tasks are connected using the `i2c_master_if` interface (shown in Line 1). The sensing is achieved every 0.2 seconds via a timer event (shown in Line 7) and when the sensing of each sensor complete, the value is sent by its respective interface `distancesensor_if` to the acceleration task (Line 22).

The proximity sensing task and how data is transferred between other tasks can be seen in the software component diagram in Figure 3.19 with the proximity sensing task having the function name `Task_ReadSonarSensors`.

```
 1  void Task_ReadSonarSensors(client i2c_master_if i2c_interface, client distancesensor_if
        sensors_interface)
 2  {
 3     // .. Declaration of some variables..
 4     while (1) {
 5        select
 6        {
 7           case tmr when timerafter(time) :> void :
 8              //Initialize messaging
 9              InitializeMessaging(i2c_interface);
10              // For Left Sensor
11              // Read from high and low byte respectively
12              high_byte = i2c_interface.read_reg(getDistanceSensorAddr(
                    LEFT_DISTANCE_SENSOR_ID), 0x02, result);
13              low_byte = i2c_interface.read_reg(getDistanceSensorAddr(
                    LEFT_DISTANCE_SENSOR_ID), 0x03, result);
14              // Construct the distance information in centimeters
15              acc = (high_byte * 256) + low_byte;
16              if ((acc < 600) && (acc > 0)) // Distance should be in between 600cm and 0cm
17                 left = acc;
18              else
19                 left = 0;
20              //...repeated for other sensors...
21              // Send sensor values all together
22              sensors_interface.ShareDistanceSensorValues (left, right, front, rear);
23              // Delay
24              time += delay;
25              break;
26        }
27     }
28  }
```

Listing 3.5: Proximity sensing task

3.2.4. Lighting System

The light system of the A4MCAR is a light module from the RC-Car parts producer Modelcraft that is driven with PWM. The module uses two PWM channels: one for the light adjustment due to the steering input, one for light adjustment due to acceleration input. Similarly to all the other PWM control tasks, the PWM signal generator template that is given in 3.3 has been used in order to the generate correct pulse widths for the light system. The task that is given in 3.3 has been adjusted to have two timer events for each PWM channel as opposed to one timer event. Desired pulse widths for several modes are contained in the header file of the light system task and these modes have been selected given the steering, angle, and gear inputs from the Bluetooth communication task. The functions regarding pulse width time generation (with 1ms to 2ms pulse length) for different light system modes (acceleration, braking, turning right or left) have also been created.

The light system task and how data is transferred between other tasks can be seen in the software component diagram in Figure 3.19 with the light system task having the function name `Task_ControlLightSystem`.

3.2.5. Bluetooth Communication

In order to configure the RN42 Bluetooth module [57] as a slave to communicate with the Android phone and received data, the UART communication has been implemented using `lib_uart` from XMOS. In order for the UART communication to be handled correctly, CTS and RTS pins of the RN42 module should be short-circuited since a data flow protocol such as RS232 is not used in our application. The UART library has been configured to have a buffer size of 512 bytes and a baudrate of 115200bps which is a high speed UART data rate standard that conforms many microcontrollers. Several tasks have been implemented such as uart_rx, uart_tx, and the Bluetooth communication task in order to implement bluetooth control feature into the A4MCAR. While the uart_rx and uart_tx tasks handle port accesses and data transfer in respective directions using sporadic events, the Bluetooth communication task is responsible for configuring the Bluetooth module and receiving driving commands. For an easy communication, a string with a number of bytes is constructed and interpreted by the Bluetooth communication task. This string which is referred in this thesis as *driving command* is given and explained in the Figure 3.24. The command parsing and how the received data is accumulated to other tasks is given in the Listing 3.6.

The operation of this portion of the task is explained as follows:

- A preprocessor macro is defined which is RN42_INITIAL_CONFIG. This macro activates the configuration function to configure the RN42 module in slave mode. This

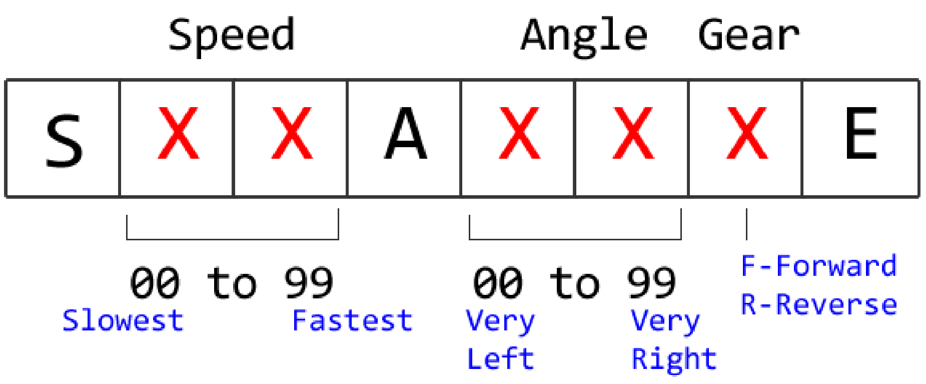


Figure 3.24.: Driving command string format generated to contain speed, angle, and gear information

configuration is shown line 2 of the Listing 3.6. It should be noted that the configuration should only be done once per bluetooth module.

- Receiving driving commands using the UART receive event (Listing 3.6, Line 7), integrity check for the command (Line 20), and command parsing (in order to obtain speed, angle, and gear values) (Line 21) are implemented.
- Since the driving command is not the only source of actuation data source for the A4MCAR and it could have overriding commands over Ethernet from the high-level module image processing task, ethernet override is handled with an integrated event in the Bluetooth communication task (shown in Line 11 in Listing 3.6).
- Reverse driving mode is not entered by the TBLE02-S Electronic Speed Controller unless the motor fully stops. To achieve this, normally the user has to select reverse mode several times from the controller. In order to get rid of this issue, once the reverse command is received, the software enters the reverse mode a few times with very short delays, thus users can select it only once and enjoy driving without having to select the mode multiple times by themselves. This implementation is shown in Line 23 in the Listing 3.6.
- The obtained control and steering values are sent to the associated task functions in order to handle the actuation (Listing 3.6, Lines 27 through 29).

```
 //... Decleration of some variables ...
#ifdef RN42_INITIAL_CONFIG
 InitializeRN42asSlave(uart_tx);
#endif
 while (1) {
     select {
     case uart_rx.data_ready(): //Read when data is available
         data = uart_rx.read();
         // ... Read byte and fill the buffer ...
         break;
     case cmd_from_ethernet_to_override.SendCmd(char* override_command, int cmd_length):
         // ... Fill the buffer with override command and raise
         //    the flag to show override occured ...
         break;
     //Process the commands received above in a timer event
     case tmr2 when timerafter(time2) :> void : // Timer event
         time2 += delay2;
         if ( command_line_ready ){
             // Check if incoming data is as expected..
             if ( CheckIfCommandFormatIsValid(command) == 1 ){
                     {speed, steering, direction} = ParseRCCommandString (command);
                     //...Send light system mode given our speed, steering, and direction...
                     if (previous_direction == FORWARD && direction == REVERSE){
                             //Commands to cheat into REVERSE mode
                             CheatIntoReverseMode();
                     }
                     steering_interface.ShareSteeringValue(steering);
                     control_interface.ShareDirectionValue(direction);
                     control_interface.ShareSpeedValue(speed);
                     command_line_ready = 0;
                     previous_direction = direction;
                     previous_lightstate = lightstate;
             }
         }
         break;
     }
 }
```

Listing 3.6: Bluetooth communication task pseudocode

The bluetooth communication task and how data is transferred between other tasks can be seen in the software component diagram in Figure 3.19 with bluetooth communication task having the function name `Task_MainProcessingAndBluetoothControl`.

3.2.6. Ethernet (TCP) Server Implementation

TCP server of the A4MCAR low-level module acts as the only source of communication that is implemented between the low-level module and the high-level module. The TCP communication that is implemented for this basic data transmission and reception is called as `Telnet` [58]. As mentioned, while *overriding driving command* (Figure 3.24) is sent from high-level module (configured as the Telnet server) to low-level module (configured as the Telnet client), for the visualization purposes the core utilization information is sent from low-level module to high-level module. On the low-level side, library that is provided from XMOS *lib_xtcp* is used with the following adjustments:

- The application notes from XMOS involves only UDP applications. This application has been manipulated in order to support the TCP protocol based Telnet server
- The TCP server has been configured with a static IP address and with a bind port.
- Data receiving and transmitting event handlers as well as the interface that sends the driving command to the Bluetooth communication task are implemented.
- Since the media-independent interface (MII) of xCORE-200 eXplorerKIT supports up to 1000Mbps high-speed connection (RGMII), the ethernet server task takes up to 3 to 4 cores in order to be executed concurrently. In order to reduce the excessive core usage, this interface has been reduced to a speed of 100Mbps by the software that takes about 2 cores in order to be parallelized efficiently. Furthermore, it is important to mention that since application in the A4MCAR does not require a gigabit ethernet connection, this decrease in the speed did not affect the performance of the communication.

The TCP server task and how data is transferred between other tasks can be seen in the software component diagram in Figure 3.19 with TCP server task having the function name `Task_EthernetServer`.

3.2.7. Core and Tile Monitoring

The core monitoring task, that is responsible for identifying core utilization percentage and sending it to the TCP server, is created for the two individual tiles of the xCORE-200 eXplorerKIT. By checking the status register `XS1_PSWITCH_T0_SR_NUM` that is mentioned in the XMOS datasheet [37], and polling this register with a maximum polling rate of 1250Hz, whether a core is busy or idle at a given time is detected. A code snippet that is responsible for this operation is given in Listing 3.7.

```
 1  default://Timer event with maximum possible polling rate
 2     //For each core in the tile
 3     for (t = 0; t <= 7; t++) {
 4        // Read the processor state
 5        int ps_value = getps(0x100*t+4);
 6
 7        // Read the status register
 8        unsigned int sr_value;
 9        read_pswitch_reg(tile_id, XS1_PSWITCH_TO_SR_NUM+t, sr_value);
10
11        const int in_use = (ps_value & 0x1);
12        const int waiting = (sr_value >> 6) & 0x1;
13        if (in_use) {
14           if (waiting) {
15              core_idle[t] += 1; //Count this cycle as idle
16           } else {
17              core_busy[t] += 1; //Count this cycle as busy
18           }
19        }
20     }
21     break;
```

Listing 3.7: Finding busy and idle cycles in XS1 architecture

In Listing 3.7, the status register `XS1_PSWITCH_TO_SR_NUM` and the processor state are read (Line 5 and Line 9, respectively) for every core in order to find which core is idle and which core is busy at that moment.

The obtained busy and idle cycles are then converted to a percentage value with the following Equation:

$$core\_usage\_percentage = \frac{busy\_cycles}{busy\_cycles + idle\_cycles} 100 \tag{3.1}$$

Further features introduced in the core monitoring task are listed as follows:

- An interface is created (`core_stats_if` in Figure 3.19) in order to periodicallysend core utilization percentage to the TCP server task.
- A preprocessor macro `FLOATING_POINT_SHOW` is also created in order to find core utilization percentage in floating point format for better precision if desired.

Observing the results showed that the user created applications for the low-level module in A4MCAR are not very time intensive. Consequently, that is why some of the tasks resulted in a core utilization value that is lower than 0 percent in utilization according to the used core monitoring approach.

The core monitoring tasks and how its data is transferred between other tasks is shown in the software component diagram in Figure 3.19 with tasks having the function names `Task_MonitorCores`.

3.3. High-Level Module Design and Implementation

3.3.1. Overview

The high-level module of the A4MCAR is composed of several processes and threads running under the Raspbian [41] distribution of Linux Operating System that is designed for the Raspberry Pi 3. In the Section 3.1.2.2, the operation of the Linux kernel is briefly introduced. During the compilation, debugging and execution of the developed processes, several development platforms such as Python 2.7 shell [59], GNU C Compiler (GCC) [60] are used. Although it should be noted that remote development using Eclipse IDE [61] is also possible, the development of the A4MCAR has been done using the aforementioned development platforms by connecting into the Raspberry Pi 3 using SSH connection. While the main processes involve C, C++, Python, and Bash [62] languages, via the capability of the integrated web server to serve web pages, several other scripting and markup languages such as HTML, CSS, JavaScript (with AJAX [63] and jQuery [64] frameworks) have also been used. The operation of the user developed processes along with third party utility processes and threads that are integrated into the system are given in the Figure 3.25. In the figure, yellow blocks represent threads, blue blocks represent mapped-memory-based inter process communication, whereas white blocks are dedicated to processes.

Since cross development platforms using languages such as C, C++ and Python have been used at the A4MCAR high-level module, the multi-tasking is handled mostly in the process layer rather than at thread or task level. This means that each process are executables of their own using different libraries and compilers. However, the touchscreen display process and dummy graph are designed with several threads.

In Figure 3.25, it is also shown that the communication between user developed processes are handled with mostly via file accesses (technically called as `mapped memory communication`). All file accesses are asynchronous and there is no event to wait for data or require data within some time as it is in the low-level module inter-process communication. This should indicate that the communication using read-write accesses does not constrain the processes as it is in a regular inter-process communication. Furthermore, it should be noticed from Figure 3.25 that although a process is able to read from many files, there is no example of two or more processes trying to write to the same file. Reading from many files is not critical, while the latter (i.e. two or more processes trying to write to the same file) should be handled by

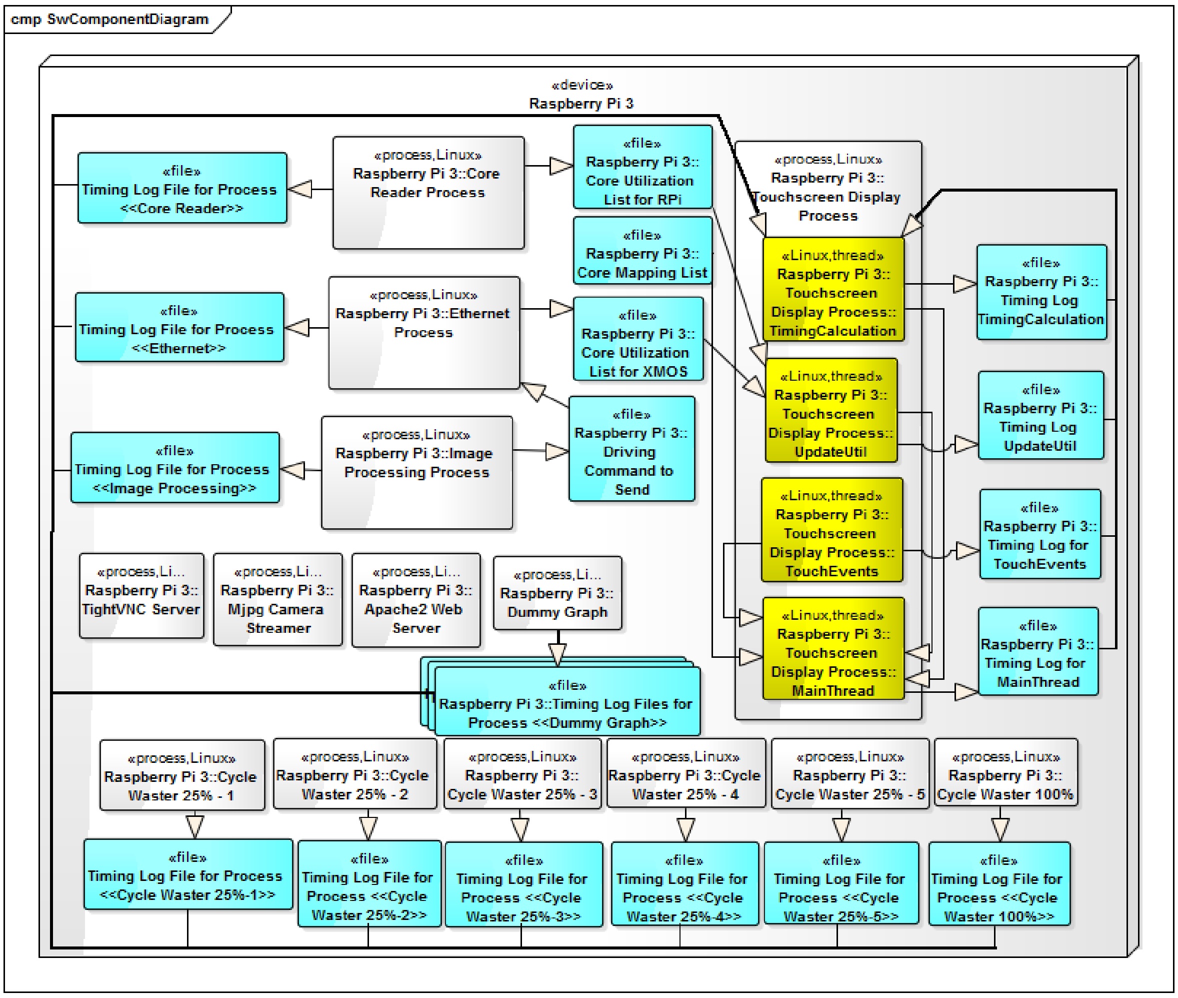

cmp SwComponentDiagram
«device» Raspberry Pi 3
«file» Timing Log File for Process <<Core Reader>>
«process,Linux» Raspberry Pi 3::Core Reader Process
«file» Raspberry Pi 3:: Core Utilization List for RPi
«file» Raspberry Pi 3:: Core Mapping List
«process,Linux» Raspberry Pi 3:: Touchscreen Display Process
«file» Timing Log File for Process <<Ethernet>>
«process,Linux» Raspberry Pi 3::Ethernet Process
«file» Raspberry Pi 3:: Core Utilization List for XMOS
«Linux,thread» Raspberry Pi 3:: Touchscreen Display Process:: TimingCalculation
«file» Raspberry Pi 3:: Timing Log TimingCalculation
«Linux,thread» Raspberry Pi 3:: Touchscreen Display Process:: UpdateUtil
«file» Raspberry Pi 3:: Timing Log UpdateUtil
«file» Timing Log File for Process <<Image Processing>>
«process,Linux» Raspberry Pi 3::Image Processing Process
«file» Raspberry Pi 3:: Driving Command to Send
«Linux,thread» Raspberry Pi 3:: Touchscreen Display Process:: TouchEvents
«file» Raspberry Pi 3:: Timing Log for TouchEvents
«process,Li... Raspberry Pi 3:: TightVNC Server
«process,Li... Raspberry Pi 3:: Mjpg Camera Streamer
«process,Li... Raspberry Pi 3:: Apache2 Web Server
«process,Li... Raspberry Pi 3:: Dummy Graph
«Linux,thread» Raspberry Pi 3:: Touchscreen Display Process:: MainThread
«file» Raspberry Pi 3:: Timing Log for MainThread
«file» Raspberry Pi 3::Timing Log Files for Process <<Dummy Graph>>
«process,Linux» Raspberry Pi 3::Cycle Waster 25% - 1
«process,Linux» Raspberry Pi 3::Cycle Waster 25% - 2
«process,Linux» Raspberry Pi 3:: Cycle Waster 25% - 3
«process,Linux» Raspberry Pi 3::Cycle Waster 25% - 4
«process,Linux» Raspberry Pi 3:: Cycle Waster 25% - 5
«process,Linux» Raspberry Pi 3:: Cycle Waster 100%
«file» Timing Log File for Process <<Cycle Waster 25%-1>>
«file» Timing Log File for Process <<Cycle Waster 25%-2>>
«file» Timing Log File for Process <<Cycle Waster 25%-3>>
«file» Timing Log File for Process <<Cycle Waster 25%-4>>
«file» Timing Log File for Process <<Cycle Waster 25%-5>>
«file» Timing Log File for Process <<Cycle Waster 100%>>

Figure 3.25.: High-level module software component diagram including files and file accesses

cross-process mutexes or semaphores that would be able to lock and unlock the same physical memory space from cross-processes. Although rarely used in A4MCAR's touchscreen display, it must be known that by using the existing cross-process mutexes or creating a semaphore mechanism, one should be able to allow two or more processes to write to the same file [44]. However, it must be pointed out that the locking of the files are handled with the locks from OS kernel in the case of A4MCAR and there is no need to create new locks in the applications for file accesses.

The way multi-tasking is handled within this constructed software architecture (Figure 3.25) is that every process is run by an external script at boot time (or via touchscreen interface, which is the main control interface in our case) and their scheduling is handled by the Linux kernel. While the scheduling is not manipulated, the mapping or pinning of processes to different cores and evaluating them are the focus of A4MCAR in order to find the most optimal parallelization solution.

Regarding hardware, the high-level module is connected to two devices. The interfacing of these devices, a Raspberry Pi camera v2.0 and a Touchscreen display is illustrated in Figure 3.11. It is shown in the figure that interfaces such as HDMI, SPI, and CSI have been utilized. In the following sections, hardware communication and the related software is further explained.

3.3.2. Implemented Online Timing Features and Making Processes Schedulable

In order to seek an assessment technique to compare timing performance of different distributions and to ensure that the developed processes and threads are schedulable, online timing features are implemented in the user-developed processes of the high-level module of the A4MCAR. Thus, while applications are running, a performance evaluation is done with the help of the those features. The code skeleton is developed for both Python and C,C++ applications and the applications are integrated on top of the skeleton with timing features. Therefore, it is important to understand how each application that will be discussed in the following sections performed regarding timing.

Recall that in the Section 2.7 the timing properties in a scheduled system are explained as shown in Figure 2.4. By observing the figure, limitations of implementations and the implemented timing features could be listed as follows:

- Because the values such as IPT, CETs and RT (referred from the Figure 2.4) are out of our reach and they are hidden in the Linux kernel, in the online timing analysis features that are implemented, those values have been neglected. With the help of the offline scheduling analysis, however, CET values can be easily obtained.
- Recording the start and end times of tasks requires an accurate clock. For that purpose, in computers in general there are two types of clocks: (1)- User CPU clock and (2)- System CPU clock. While the user CPU clock is used for finding out how long it has passed since the program has started, the system CPU clock takes place in the kernel space and measures how long it has passed since 1st of January, 1970. The latter clock was found as the viable solution as compared to the user CPU clock since the user CPU clock does not the allow comparison along the entire Linux kernel. In the code, functions `time.time()` for Python and `clock_t clock()` have been used in order to record start and end times more accurately [65] [66].
- Finding the execution time (ET) of one task/process iteration using the following Equation, provided that `start_time` is the time recorded before the iteration and `end_time` is the time recorded after the iteration. The units are all denoted in seconds.

$$execution\_time = end\_time - start\_time \tag{3.2}$$

In the version that is developed for the C language, since clock_t is able to measure clock cycles rather than seconds, the Equation is changed to the following:

$$execution\_time = \frac{end\_time - start\_time}{CLOCKS\_PER\_SEC} \tag{3.3}$$

- Finding the slack time (ST) is one of the most important tasks that is within the scope of the online timing features. As a rule of thumb, we could assess the timing performance by saying that if a process has a higher slack time than before, it means that task is better utilized compared to before. This is because of the fact that the CPU is doing some other task which results in a higher slack or idle time. The slack time of a previous iteration is measured by the following equation, provided that the calculation takes place right after start time is recorded and IPT is neglected.

  $$previous\_slack\_time = start\_time - end\_time \tag{3.4}$$

  It should also be noted that the C language version could be created by dividing the clock cycles with the clock cycles per second (`CLOCKS_PER_SEC`) in the same manner as execution time.

- In order to keep a constant period while the process is scheduled, the processes are delayed dynamically between each iteration. Thus, processes which have a constant period could be modeled easier and having the constant period will make the process or thread schedulable. In order to achieve a constant period each process are delayed by the following:

  $$delay\_time = period - execution\_time \tag{3.5}$$

  However, if the execution time of a process is bigger than its period, that process is counted as a process that missed its deadline, given that its period is equal to its deadline due to practicality. In addition, the deadline miss percentage is another important criteria in order to assess parallelization quality as it is normally undesired to have any missed deadlines. In case of a missed deadline in A4MCAR, the process is not delayed.

- As seen in Figure 3.25, there are many timing log files that are created. Those timing log files are created within every user-defined process and later used in the Touchscreen Display application.

While each processes and thread are constructed in the aforementioned manner in terms of traceability, the overall online evaluation is handled within the Touchscreen Display process in the TimingCalculation thread (shown in Figure 3.25), in Chapter 4, the details about these are given.

An example of the timing skeleton for Python-running processes is given in Listing 3.8. The user-defined Python-running applications have been created using this template, and the space that is left for task content is used for the actual features of that task.

```python
#!/usr/bin/env python
import psutil
import time
import string
import numpy

#Initialization
_DEADLINE = 1.40
_START_TIME = 0
_END_TIME = 0
_EXECUTION_TIME = 0
_PREV_SLACK_TIME = 0
_PERIOD = 1.40

def CreateTimingLog(filename):
   global _START_TIME
   global _DEADLINE
   global _END_TIME
   global _EXECUTION_TIME
   global _PREV_SLACK_TIME
   global _PERIOD

   try:
      file_obj = open(str(filename), "w+r")
   except Exception as inst:
      print inst
   _END_TIME = time.time()
   _EXECUTION_TIME = _END_TIME - _START_TIME
   try:
      file_obj.write(str(_PREV_SLACK_TIME)+'␣'+str(_EXECUTION_TIME)+'␣'+str(_PERIOD)+'␣'+str(
          _DEADLINE))
      file_obj.close()
   except Exception as inst:
      print inst

while True:
    #Timing Related
   _START_TIME = time.time()
   _PREV_SLACK_TIME = _START_TIME - _END_TIME

   #####
   #TASK CONTENT GOES HERE
   #####
```

```
43
44    #Timing Related
45    CreateTimingLog("deadline_logger_burn_cycles_around25_1.inc")
46
47    #Sleep
48    if(_PERIOD>_EXECUTION_TIME):
49        time.sleep(_PERIOD - _EXECUTION_TIME)
```

Listing 3.8: Online timing features implemented in Python language

With the given code by the Listing 3.8, following remarks can be made:

- Lines 2 through 5 indicate which libraries are used.
- Between the Lines 8 and 13, global variables to hold the time values are initialized.
- A timing data logging function is created (Lines 15 through 33). In this function, timing log file is opened (Lines 23 through 26), execution time is calculated after end time is recorded (Line 27 and Line 28) and then all the timing values at that instant are written into the opened text file (Lines 29 through 33). After the write operation the file is closed (Line 33).
- In the loop section of the process `start time` and `previous slack time` are recorded (Lines 37 through 38) before the actual task content (Lines 40 through 42) is executed. After the task content is executed, timing log is created by using the timing data logging function (Line 45) and then the task is delayed according to its period by finding the `delay_time` that was given in Equation 3.5. This delay operation is also given in the Listing 3.8 at the Lines 48 through 49.

3.3.3. Core Reader

To support the utilization assessment and visualization purposes, a core reading process is developed that monitors cores periodically and writes the core usage information to a text file. For that purpose, the *psutil* module [67] from Python is used. The *psutil* module allows to find information of Linux processes and cores. The core usage information for four cores of Raspberry Pi that is logged into a text file is then used for visualization in the web interface and the touchscreen interface.

With the help of a simple function, the core usage information is easily retrieved. The function is given in the Listing 3.9. It should be noted that `d` in the listing is an array with 4 elements, each of which indicating thecore usage for an individual cores.

```
if (_PERIOD>_EXECUTION_TIME):
     d = psutil.cpu_percent(interval=(_PERIOD - _EXECUTION_TIME), percpu=True)
```

Listing 3.9: Psutil function to retrieve core utilization information

3.3.4. Ethernet (TCP) Client Implementation

In Section 3.2.6, the TCP server implementation using the xCORE-200 eXplorerKIT is discussed. In order to maintain a sound data communication between low-level module and high-level module, a TCP client process is implemented in the high-level module. Since the Python language offers very stable and easy-to-use threading support and exception handling, the TCP client implementation is achieved by using Python. The *socket* library [68] in Python is capable of delivering several functions and objects that are required for this purpose. The TCP client has been configured to have non-blocking data reception with 0.5 second period and with a timeout of 2 seconds. While the data reception is handled by an additional thread, operations such as connecting to the server, binding to the server port, and sending data periodically is handled in the main thread. The overriding driving command which is send to the low-level module is read from the file before data transmission. It should be also noted that after the data reception the content is written to the file which is responsible for holding low-level module core usage information. This communication between high-level module and low-level module is illustrated in the deployment diagram given in Figure 3.26 (to be explained in the following sections).

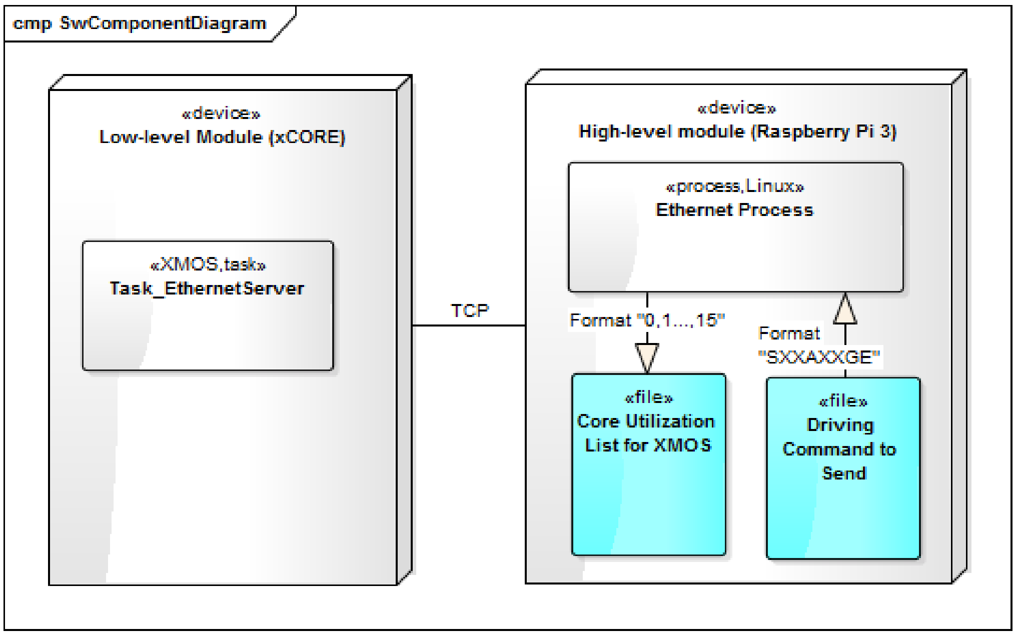


Figure 3.26.: Deployment diagram showing Ethernet communication

3.3.5. Web Server and its Applications

3.3.5.1. Web Server

In order to develop a web interface for the A4MCAR, a web server is installed and configured on the high-level module. Web servers are responsible for processing HTTP requests and delivering HTTP responses [69]. The HTTP requests and responses are usually visualized using a web browser from clients in the form of web pages [69]. At the A4MCAR high-level module, the Apache 2 web server is installed and configured as the web server since it is an open-source, robust, light-weight cross-platform that has a large user community. Additionally, another reason Apache 2 is selected is that it is capable of serving for script languages such as PHP and Python, which are used at the A4MCAR applications.

Just like a Telnet server, a web server is bind to a port in a wireless or wired network. Although for different communication channels one could use different ports, web servers usually use the port number 80. Another difference of a web server is that unlike a telnet server, the data that is sent and interpreted is in the HTTP (HyperText Transfer Protocol) [70] format unlike the TCP (Transmission Control Protocol) format. How web servers and web browsers work in order to help visualize web pages is illustrated with the Figure 3.27 [70].

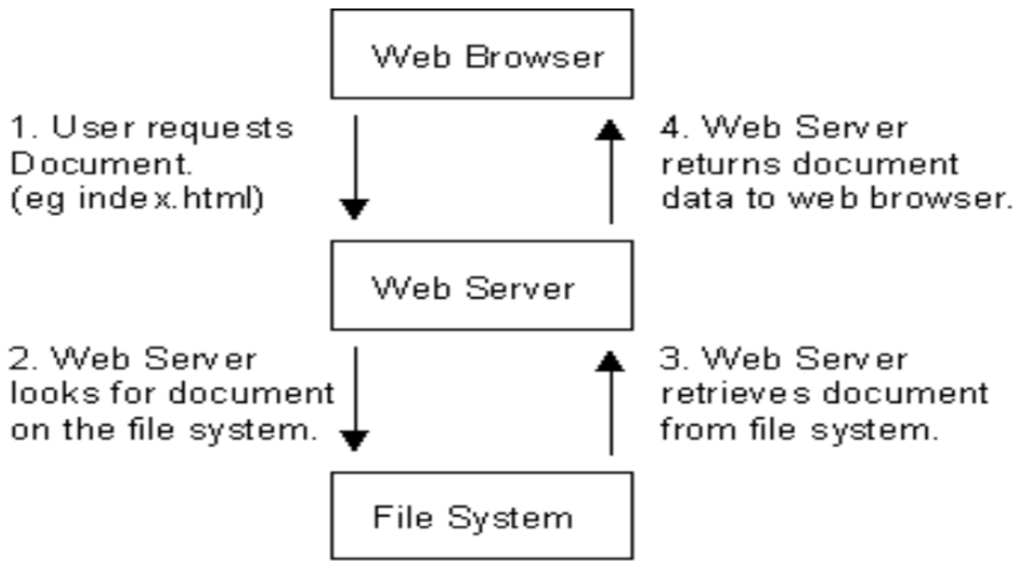

Web Browser
1. User requests Document. (eg index.html)
4. Web Server returns document data to web browser.
Web Server
2. Web Server looks for document on the file system.
3. Web Server retrieves document from file system.
File System

Figure 3.27.: How web servers and web browsers work illustrated [70]

The following technologies have been used in order to create dynamical webpage:

- **HTML:** This markup language is used for defining how body elements are located in a web page and including scripts.

- **CSS:** CSS is used for defining the style of body elements. Borders, background properties, colors, button styles, positioning of elements are defined with CSS language.
- **JavaScript:** The JavaScript language is used for defining animations, as well as how events would behave.
- **jQuery:** jQuery [64] is a JavaScript framework that is written using JavaScript which helps to use define scripts easier than it is with the JavaScript. It is open-source and widely used in almost every web page.
- **AJAX:** AJAX [63] is another framework for JavaScript which is used for handling dynamical HTTP requests without having to refresh the page. With the objects it delivers, events such as *key press*, *mouse events*, or *conditional events* could be sent to server and processed. The returned data could be processed using JavaScript in order to dynamically update the web page content [63]. This mentioned working principle is illustrated in Figure 3.28 [63].

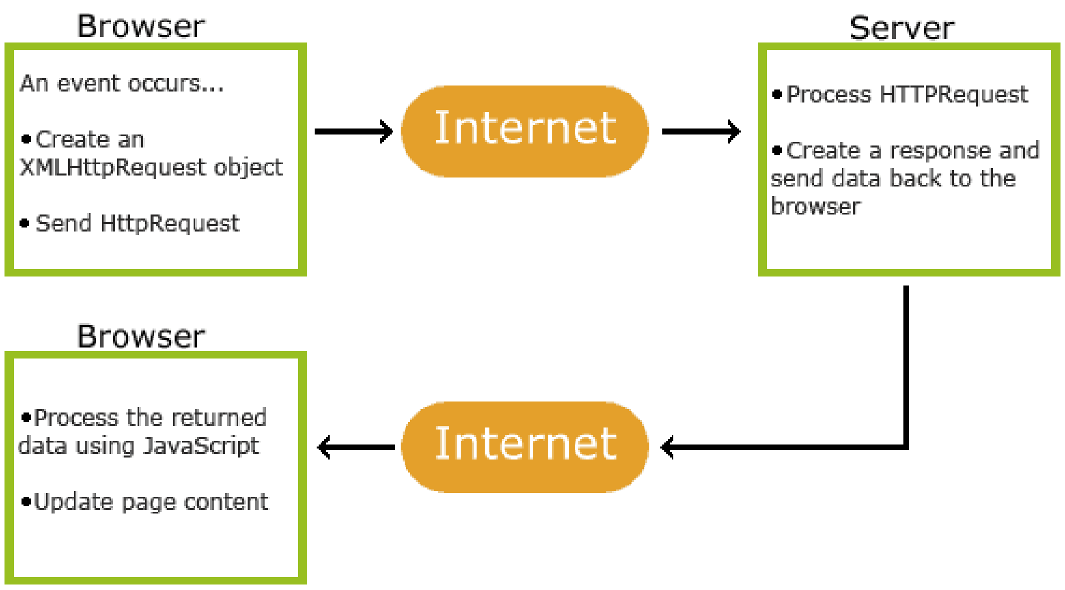

Browser
An event occurs...
•Create an XMLHttpRequest object
•Send HttpRequest
Internet
Server
•Process HTTPRequest
•Create a response and send data back to the browser
Browser
•Process the returned data using JavaScript
•Update page content
Internet

Figure 3.28.: How AJAX works [63]

3.3.5.2. Web Page Design and Implementation

The web page that has been designed is shown in Figure 3.29. At the web page, a camera stream, control buttons and sliders, and an information graph that shows core utilization in both high-level and low-level module are embedded. For the static design of the web page HTML and CSS are used, while the dynamical behavior of the web page is supported with jQuery, AJAX, and Python. The dynamical behavior of the individual parts of the web page will be explained in the following sections.

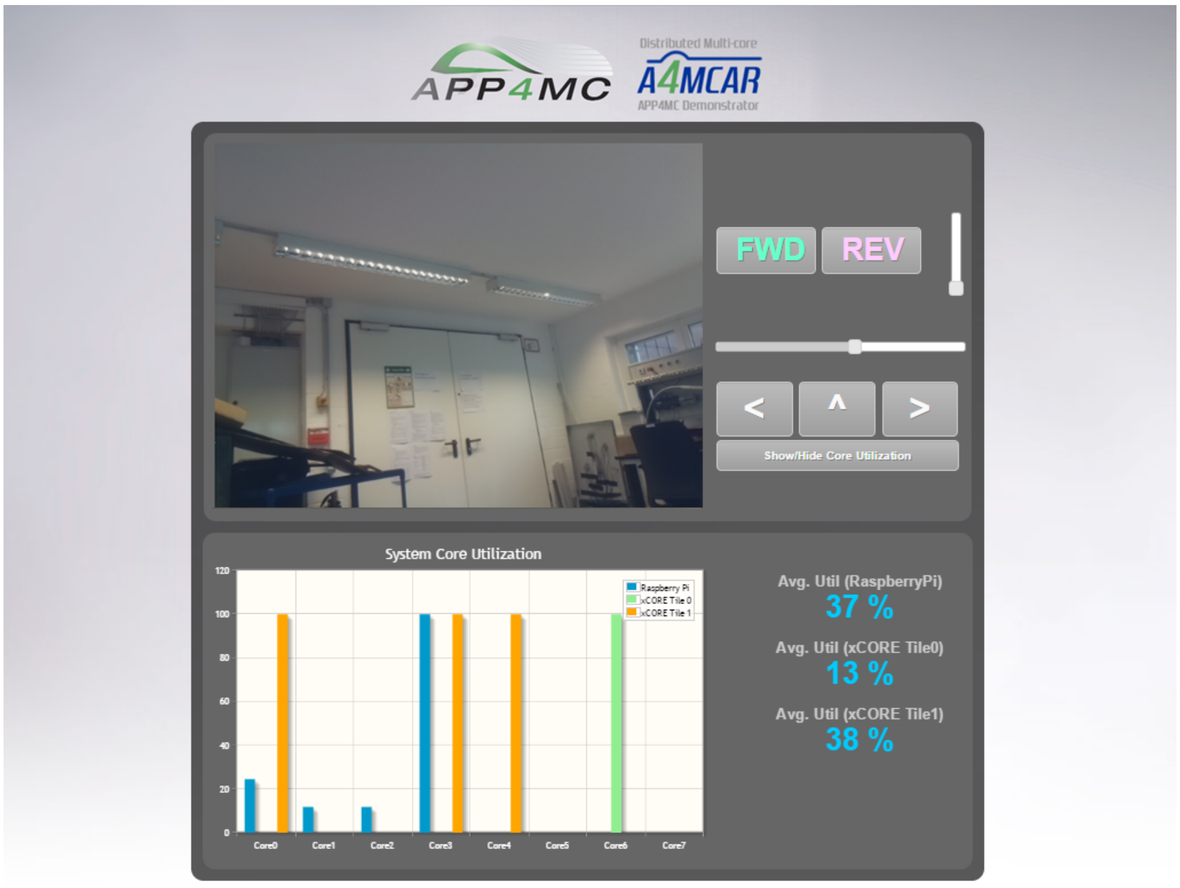


Figure 3.29.: Web interface of the A4MCAR

An alternative to this interface has also been created which is used for smoother driving by using arrow buttons. However, the interface that uses buttons can use only constant speeds for actuation which are set to 80 percent of the full speed. The alternative interface can be seen in Figure 3.30

The overall dynamical behavior of the web page is illustrated in a component diagram in Figure 3.31. In this diagram, one must notice that the server page `jqueryControl.php` is the main web interface. It has some server pages and files embedded to it in order to function as a whole to deliver the features of controlling the A4MCAR, core utilization display, and camera streaming. In the following subsections, each of these tasks is explained using the component diagram shown in Figure 3.31.

3.3.5.3. Controlling A4MCAR via Web Page

At the top right of the Figure 3.29, the controls to drive the A4MCAR over web interface are shown. It is shown that there are gear selection buttons such as *Forward (FWD)* and *Reverse (REV)*, along with two sliders. The sliders are created using the third party script

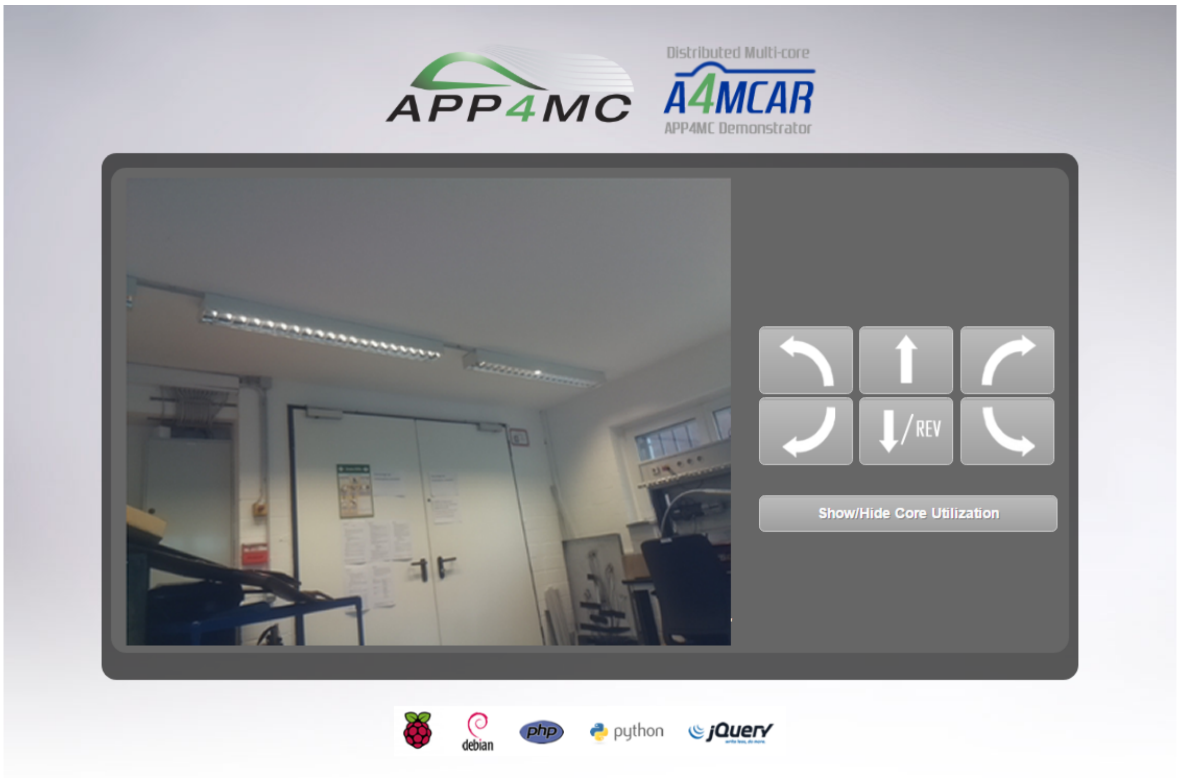


Figure 3.30.: Alternative web interface of the A4MCAR

library called *jquery_ui*. While the vertical slider is for speed adjustment, the horizontal slider is used for angle adjustment. Additionally, to select the very left, straight, and very right angles the arrow buttons can be used.

With the help of jQuery and AJAX' ability to create event handlers within the server pages, on a button press or when slider position is changed, an event handler is run that collects the position and gear information and sends it using HTTP GET request dynamically to another server page called `pythonControl.php` (shown in Figure 3.29). With the idea of demonstrating a basic AJAX request, an example is given in the Listing 3.10.

```
$.ajax({
     url: "pythonControl.php?process=S0"+speed+"A0"+direction+gear+"E",
     method: "GET",
     data: {spd: speed, dr: direction, gr: gear} //This is not necessary in every request
}).done(function( msg ) {
     alert( "Data␣Saved:␣" + msg );
});
```

Listing 3.10: Sending dynamic HTTP GET requests using jQuery

As it is seen from the Listing 3.10 url field, the information that is sent is nothing other than the format defined for the bluetooth communication in the low-level module, which is

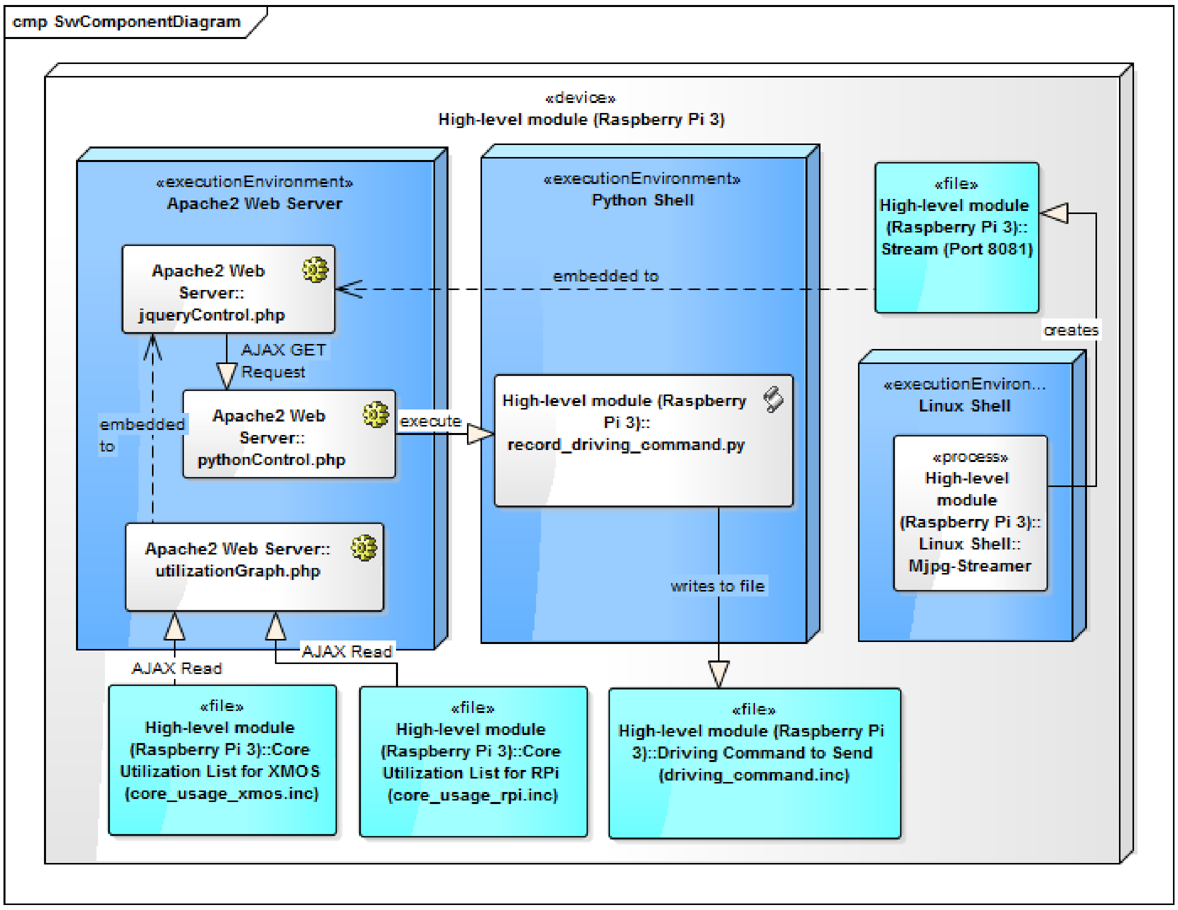

cmp SwComponentDiagram
«device»
High-level module (Raspberry Pi 3)
«executionEnvironment»
Apache2 Web Server
Apache2 Web Server:: jqueryControl.php
AJAX GET Request
embedded to
Apache2 Web Server:: pythonControl.php
execute
Apache2 Web Server:: utilizationGraph.php
AJAX Read
AJAX Read
«executionEnvironment»
Python Shell
embedded to
High-level module (Raspberry Pi 3):: record_driving_command.py
writes to file
«file»
High-level module (Raspberry Pi 3):: Stream (Port 8081)
creates
«executionEnviron...
Linux Shell
«process»
High-level module (Raspberry Pi 3):: Linux Shell:: Mjpg-Streamer
«file»
High-level module (Raspberry Pi 3)::Core Utilization List for XMOS (core_usage_xmos.inc)
«file»
High-level module (Raspberry Pi 3)::Core Utilization List for RPi (core_usage_rpi.inc)
«file»
High-level module (Raspberry Pi 3)::Driving Command to Send (driving_command.inc)

Figure 3.31.: Component diagram showing how communication inside the created web-interface works

given in Figure 3.24. In the Figure 3.29, it is seen that after the information is received by pythonControl.php, using the ability of PHP to run a shell script, a Python script is run using the Python shell automatically. The python script, whenever executed, writes the received driving command information into the text file that holds the driving command. This operation is done asynchronously. How the driving command is sent afterwards is explained in Section 3.2.6.

3.3.5.4. Camera Streaming

For the camera streaming, the third party module `mjpg-streamer` [71] has been used. This module is able to communicate over the CSI interface in order to generate a stream on a network port, which then can be embedded to web pages. It is shown at the Figure 3.31 how this module works along with the Apache2 Web Server.

For the stream, Raspberry Pi camera version 2.0 with the CSI interface is used which is shown in Figure 3.32.

Figure 3.32.: Raspberry Pi camera v2.0

Based on the documentation of the `mjpg-streamer`, a script using Bash language is created which is used for generating a web stream with the correct parameters. These parameters involve resolution, frames per second, quality value, and port on which the stream will be generated. For the case of the A4MCAR, the experimental version of `mjpg-streamer` has been used which is able to stream using the Raspberry Pi camera besides a webcam. By using the experimental version library, the following setup is found to give robust performance with respect to Raspberry Pi's graphical computing power:

- Resolution: 640x480
- Frames per second: 30
- Quality: default
- Port: 8081

3.3.5.5. Core Utilization Display

Core utilization display is shown at the bottom of Figure 3.29. It is responsible for gathering all the core usage information from the files, displaying a graph showing percentages, and calculating average core utilizations. These operations are handled within the server page `utilizationGraph.php` as shown in Figure 3.31. This server page is embedded into the main web interface which is `jqueryControl.php`.

For the efforts regarding creating a graph, a third party script library called *jqPlot* [72] which runs within the jQuery framework is used. The *jqPlot* offers various functions in order to create various plots such as bar graphs, line graphs, pie charts and 3D plots. In order to embed the plots into the web page, AJAX has been used.

3.3.6. Dummy Loads and Dummy Graph

In order to fully utilize the developed parallel software on the high-level module under full load, several processes have been created which do dummy operations to consume a certain percentage of the cores. Although the initial distribution does not involve dummy load processes due to increased responsiveness, the dummy loads are used for stressing the software. To create dummy loads, two ideas are investigated:

- **Basic load with very short periods:** While using this methodology achieves certain core percentage loads, having a very short delay is considered to be non model-safe and since periods of the processes are very short, timing logs resulted in deadline misses for further analysis. Therefore, a new method of using bigger loads with bigger periods is analyzed.

- **Bigger load with bigger periods:** While using a bigger load with bigger period is also viable in achieving certain core percentages, the fluctuation in the core utilization values is higher than the previous option. This fluctuation is trivial in our application as most of the processes behave in this way.

```python
a=numpy.random.random([1000,1000])
b=numpy.random.random([1000,1000])
c=numpy.mean(a*b)
```

Listing 3.11: Dummy load created with Python

In order to create the dummy loads in the code, matrix multiplication of random 1000 by 1000 matrices is processed. Python offers libraries to create those matrices as well as to multiply them. The basic load that is written in Python is given with the Listing 3.11.

While various loads with the same matrix operation are created, it should be noted that they differ in their iteration periods which helps to achieve different core utilization percentages. The Table 3.1 shows a list of all dummy processes that are created in order to help with the utilization research:

Although the aforementioned processes have been created to stress the Raspberry Pi to find out the parallelization behavior under full utilization, several other purposes have also been considered. One very important distinction that seperates APP4MC's industrial use from the A4MCAR is that the industry has sophisticated tracing and distribution tools and

Process Name	File Name	Period	Core Utilization
CycleWaster25_1	dummy_load25_1.py	1.4 second	25 percent
CycleWaster25_2	dummy_load25_2.py	1.4 second	25 percent
CycleWaster25_3	dummy_load25_3.py	1.4 second	25 percent
CycleWaster25_4	dummy_load25_4.py	1.4 second	25 percent
CycleWaster25_5	dummy_load25_5.py	1.4 second	25 percent
CycleWaster100	dummy_load100.py	0.50 second	100 percent

Table 3.1.: Dummy load processes running in high-level module

standards such as AUTOSAR. Thus, in industry, fine-grained runnables can be created and distributed easily. However for the A4MCAR, experiments showed that distributing and tracing such runnables are not so easy with sophisticated real-world applications. This means that the granularity of processes and threads can have a huge size difference when the functionality of the system is considered. Therefore, to demonstrate the partitioning feature of the APP4MC as in an industrial application, a software graph that is called *Dummy Graph* has been created using Python threads. The created software graph is illustrated in Figure 3.33.

The created graph depicts software runnables (in our case threads that are distributed) and their global communication via shared variables. Each arrow represents a label access and chronological execution order of runnables. For example, arrow pointing from B to F indicates that B writes to a shared variable, and after it is done, F can read and start its calculations. Inside the threads, dummy matrix multiplication with adjustable matrix size (default: 190x190) is introduced. By looping this multiplication and writing a byte to a shared variable after this is done, the dummy graph is constructed. As the legend of the image suggests, each thread is activated using 0.5 second periodic activations and threads have instruction sizes varying from approximately 9 million to 81 million, according to the dynamic profiling results. Further discussions and results of this distribution is discussed in Chapter 5.

3.3.7. Image Processing with OpenCV

Developing cyber-physical systems with multiple sensors require a good knowledge of computer vision. Automotive applications especially when developing Advanced Driving Asistance Systems (ADAS) make use of this knowledge. Huge computational power need involved in such applications makes such processes a challenge. Therefore, a demonstration of an image processing application is crucial for the A4MCAR.

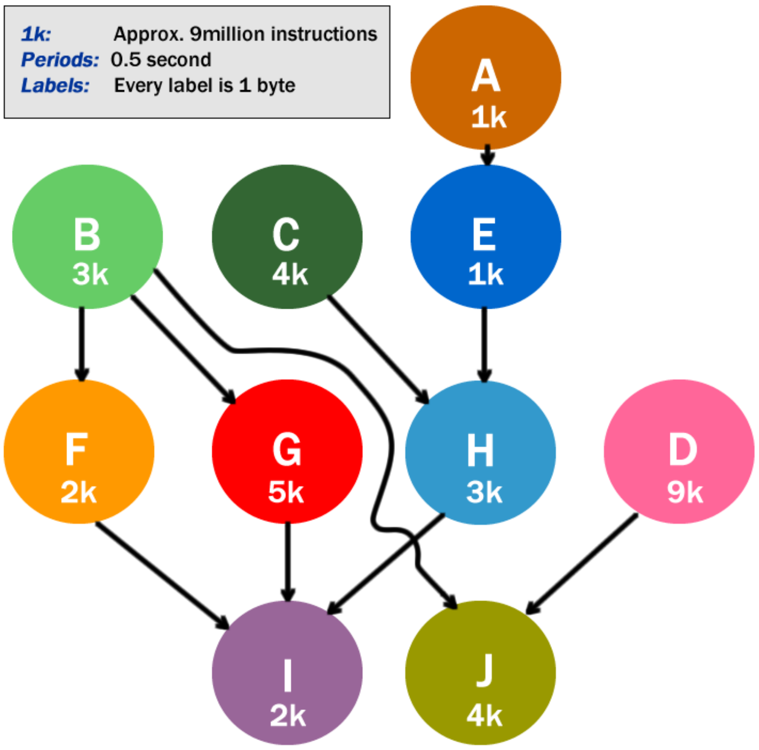

1k: Approx. 9million instructions
Periods: 0.5 second
Labels: Every label is 1 byte
A 1k
B 3k
C 4k
E 1k
F 2k
G 5k
H 3k
D 9k
I 2k
J 4k

Figure 3.33.: Dummy Graph that is created and its details

For the demonstration of parallelization of an image processing process along with several other processes and threads, an application that can roughly detect a traffic cone has been developed using the C++-based and well established computer vision library OpenCV [73]. The developed application makes use of the Raspberry Pi camera (using *raspicam* library) to retrieve images and performs several transforms to the image to detect traffic cones. The developed application outputs an `"OBJECT FOUND"` message to demonstrate its operation. Example detections are given in Figure 3.34.

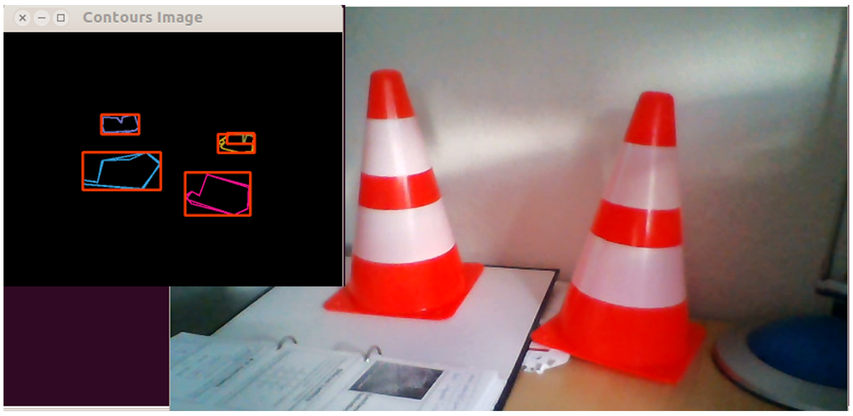
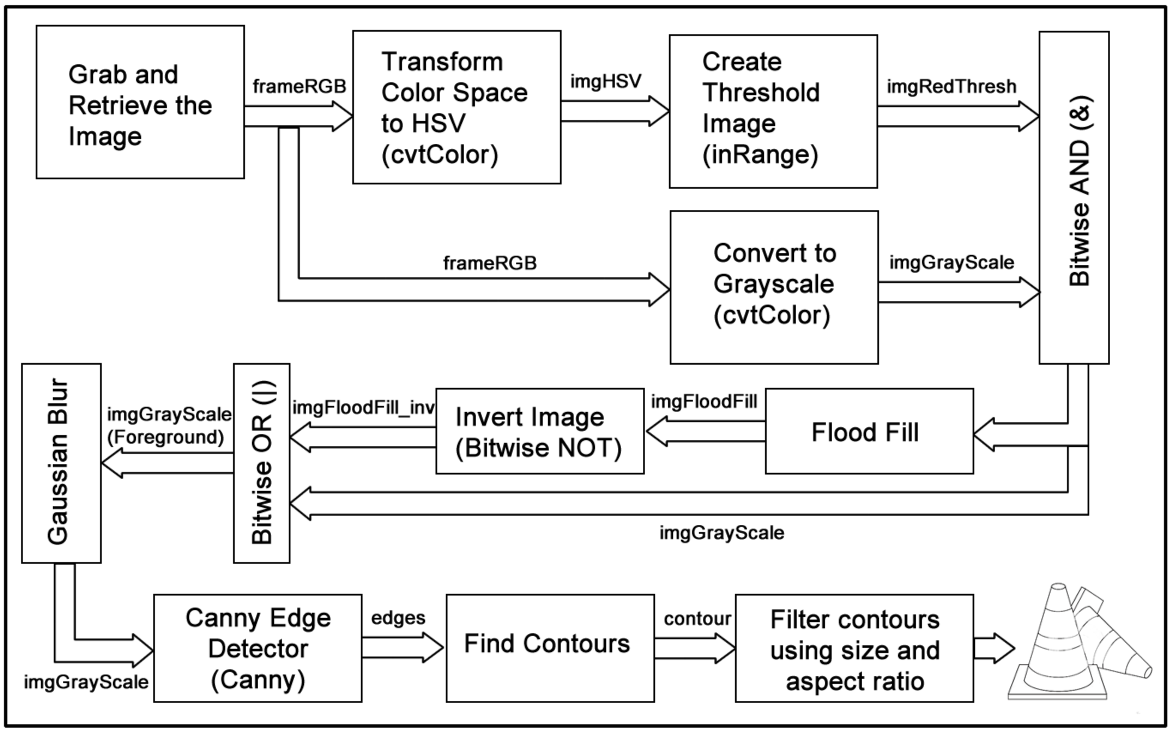


Figure 3.34.: Developed Image Processing Application

The applied transformations and how the traffic cone is detected is illustrated with Figure

3.35. The idea that is depicted is to retrieve contours of the possible objects and then determine whether it is a traffic cone or not by filtering those contours by their sizes and aspect ratios. In Figure 3.35, it is shown that several transformations are applied for this purpose. Via creating the threshold image and then subtracting the background (steps: Background AND, Flood Fill, Image Inversion, Bitwise OR), the image is prepared for the canny edge detection step. The *Canny Edge Detector*, i.e. `Canny` function is able to find edges of desired objects. By using edges, contours can be found which represent the outlines of objects. By judging the contours regarding their sizes and aspect ratios the desired objects are detected.



Grab and Retrieve the Image
frameRGB
Transform Color Space to HSV (cvtColor)
imgHSV
Create Threshold Image (inRange)
imgRedThresh
Bitwise AND (&)
frameRGB
Convert to Grayscale (cvtColor)
imgGrayScale
Gaussian Blur
imgGrayScale (Foreground)
Bitwise OR (|)
imgFloodFill_inv
Invert Image (Bitwise NOT)
imgFloodFill
Flood Fill
imgGrayScale
imgGrayScale
Canny Edge Detector (Canny)
edges
Find Contours
contour
Filter contours using size and aspect ratio

Figure 3.35.: Applied Functions in OpenCV to Detect a Traffic Cone

3.3.8. Touchscreen Display

3.3.8.1. Touchscreen Display Features

The touchscreen display that is embedded to the Raspberry Pi 3 features several functions. It can not only show core the utilization graph, average utilization percentages, timing performance, but it can also be used to manage and allocate the processes of the high-level module. It also features connectivity settings in order to connect to an access point. Main interface and buttons are shown in the Figure 3.36. Using the buttons in Figure 3.36 users can switch between display modes, as well as go to the Settings menu, exit the touchscreen application, and shutdown the Raspberry Pi.

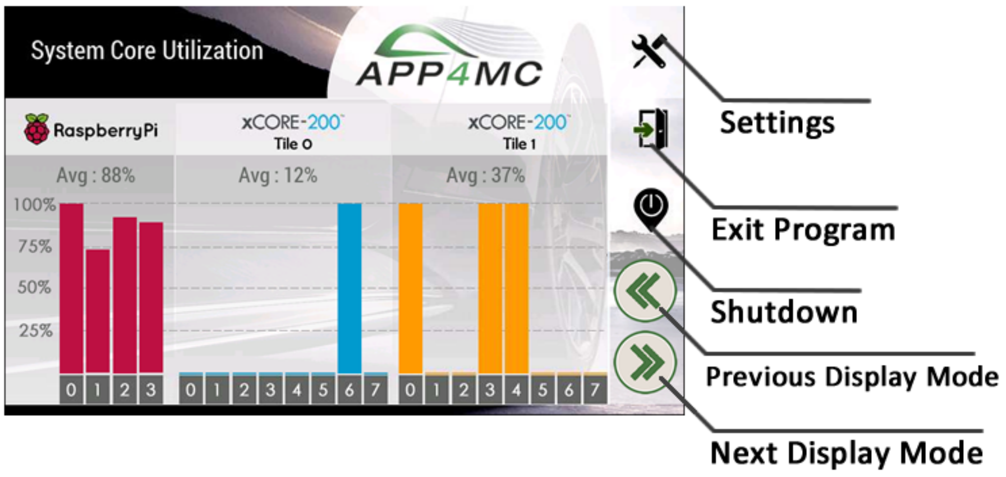


Figure 3.36.: Button functions of A4MCAR Touchscreen Display

3.3.8.2. Touchscreen Display Implementation

Regarding the hardware, a 5 inch HDMI touchscreen module from Waveshare has been used. This module can act as a primary monitor for the Raspberry Pi. Additionally, the module features touchscreen controls using the SPI pins of the Raspberry Pi GPIO. Regarding the software interface to the Linux system, module driver is installed and calibrated on the Raspberry Pi in order to use the module as a primary monitor.

The touchscreen display process uses a third-party library from Python that is called Pygame [74]. This library is exclusively developed for creating Python language based games but it is also useful for creating graphical interfaces. After the images for the interface have been designed, the main interface has been created using the functions from Pygame. As an example, one page is designed to show the system core utilization. In that page, visualization is handled by creating rectangles that scale from 0 percent to 100 percent to show the core utilization in each core that are obtained by reading the files that are responsible for holding the core usage information for both low and high level modules.

Pages that are developed for the touchscreen module are shown in the Figure 3.39. In order to understand the behavioral operation of the touchscreen process, the Figure in 3.38 can be observed in parallel with Figure 3.39.

The introduction page that is shown in Figure 3.39 (a) is entered as the process is started. After displaying the logo for 3 seconds, other pages are entered. The pages (b) through (g) (shown in Figure 3.38) are navigated with the help of the *Next* and *Previous* button in this state. Users can browse the settings page, shutdown or exit the touchscreen display process by clicking to the respective buttons.

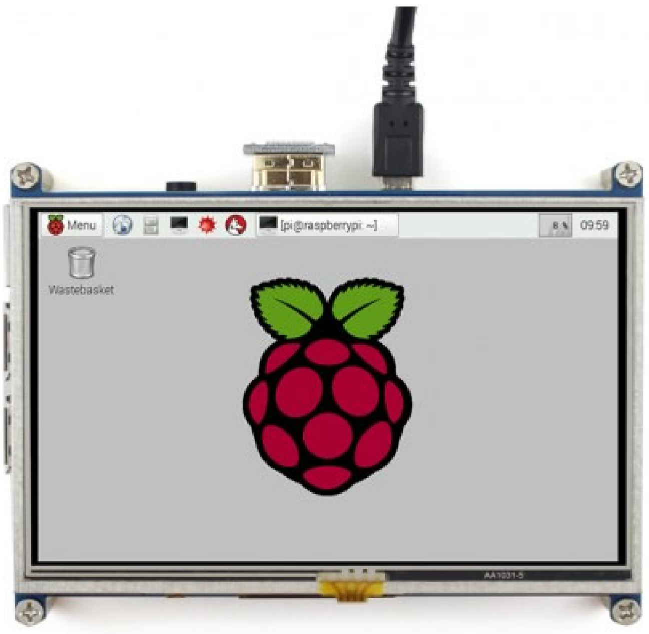


Figure 3.37.: 5 inch Touchscreen module from Waveshare

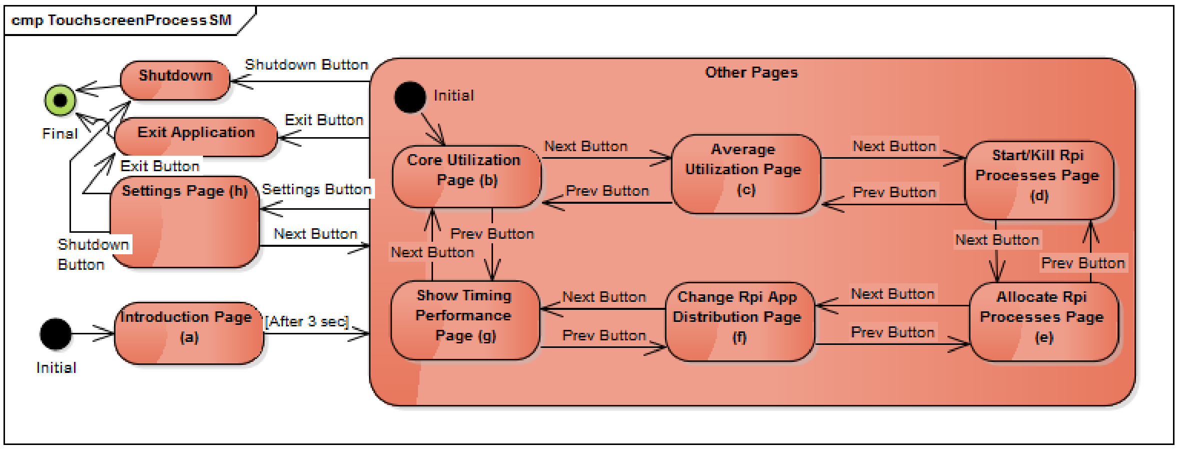
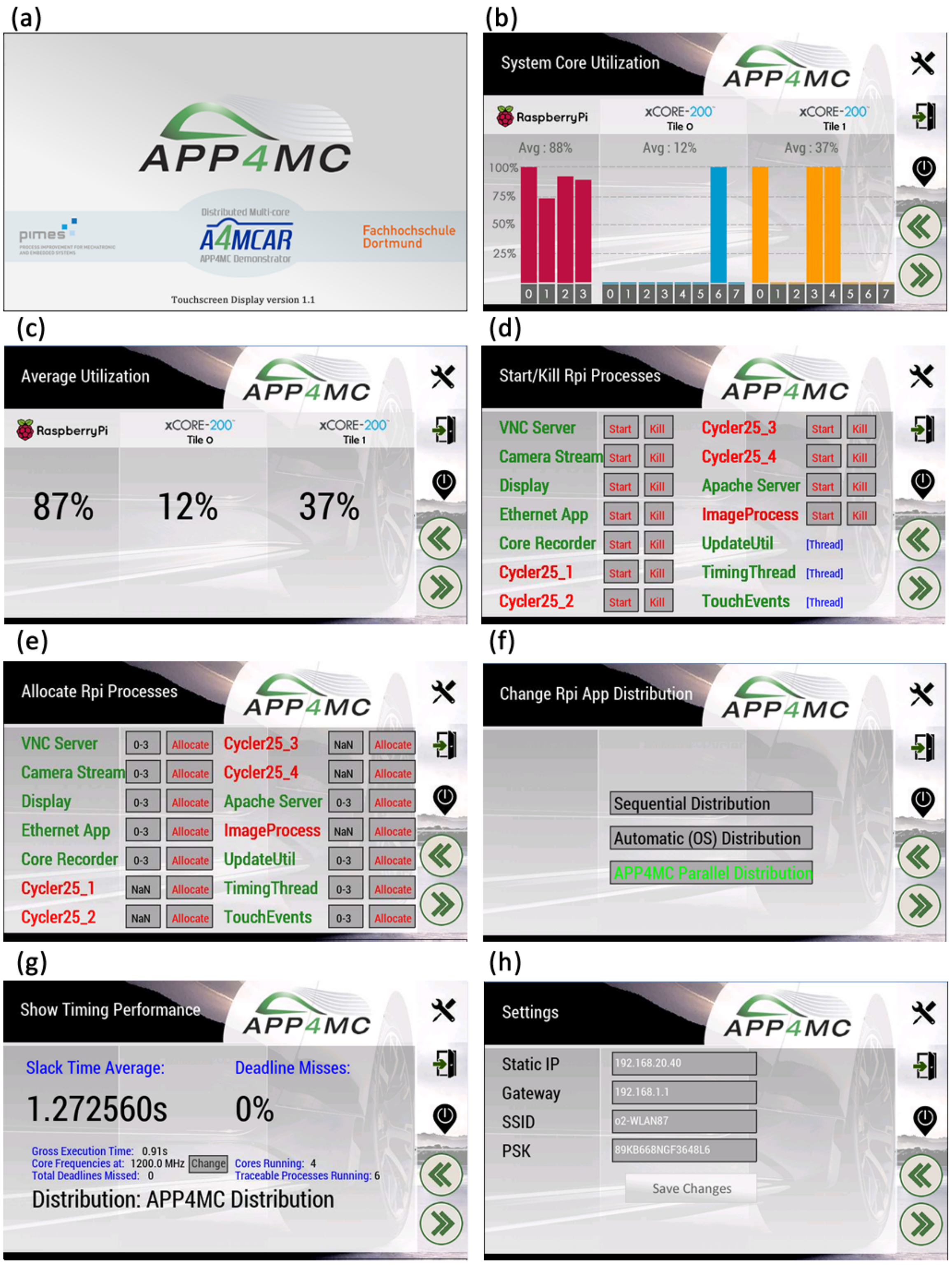


Figure 3.38.: State machine of the touchscreen process for pages as modes

In order to make the the touchscreen display software modular, a class that is called *aprocess* has been created. By instantiating *aprocess* objects and appending them to the object list `aprocess_list`, one can define the processes or threads that are displayed and traced. The touchscreen display software is designed in a modular fashion so that it automatically generates all the pages using the aprocess object list. *aprocess* class has the following attributes and functions to make the process handling easier for Python-based software:

- **Attributes:** Process Name (`apname`), Process ID (`apid`), Running flag (`aprunning`), Core Affinity (`aaffinity`), Command to start the process (`apstartcommand`), Traceable flag (`traceable`), Online tracing log file (`aplogfilepath`), Displayed flag (`displayed`), Display name (`display_name`), Flag to tell if it is a process or a thread (`is_thread`)



VNC Server	0-3		Cycler25_3	NaN
Camera Stream	0-3		Cycler25_4	NaN
Display	0-3		Apache Server	0-3
Ethernet App	0-3		ImageProcess	NaN
Core Recorder	0-3		UpdateUtil	0-3
Cycler25_1	NaN		TimingThread	0-3
Cycler25_2	NaN		TouchEvents	0-3

Slack Time Average: 1.272560s
Deadline Misses: 0%
Gross Execution Time: 0.91s
Core Frequencies at: 1200.0 MHz Change
Total Deadlines Missed: 0
Cores Running: 4
Traceable Processes Running: 6
Distribution: APP4MC Distribution

Static IP	192.168.20.40
Gateway	192.168.1.1
SSID	o2-WLAN87
PSK	89KB668NGF3648L6

Figure 3.39.: Display modes from A4MCAR Touchscreen Display

- **Functions:** `UpdateProcessIDAndRunning()` function is used to update the object process ID and whether the process is running. `UpdateProcessCoreAffinity()` function is used to retrieve the process core affinity while the `SetCoreAffinity(core_affinity)` function is used for setting the core affinity. There are also thread-specific methods such as `SetCoreAffinityOfThread(core_affinity)` and `UpdateThreadIDAndRunning()`.

The touchscreen display process has a multi-threaded design. In this design Python's threading library [18] is used. With Python's threading library, one can create threads and handle the shared memory communication between threads. The aforementioned process and thread list, `aprocess_list` is protected by using the mutex implementation `Lock()` [18] of Python's threading library. The Listing 3.12 shows how locking works with Python's threading library:

```python
lock.acquire()
# Access to the shared variable
lock.release()
```

Listing 3.12: Using locks in Python

In the implementation of the touchscreen application, the display resulted in problems due to non-locked access to `aprocess_list` variable. The concurrent write and reads to this shared variable therefore is prevented using `Lock()`.

The initial implementation of the touchscreen display did not involve any threads. However, that resulted in several problems. The most crucial problem that occured by not having a multi-threaded design was that in order to make the processes schedulable (explained in the Section 3.3.2), the responsiveness of the touchscreen would drop significantly since each display mode has different periods due to embedded calculations. By isolating the calculations, touchscreen events, and utilization updates, this problem could be resolved. The four threads that the touchscreen display component contains can be listed as follows:

- **Main Thread:** Responsible of solely displaying the information using shared variables
- **TimingCalculation Thread:** This thread is responsible of reading from all the timing log files that are registered to the application using `aprocess` class, and calculating the values such as gross execution time, slack time average, deadline misses, traceable processes running.
- **TouchscreenEvents Thread:** Pygame library provides all event handling within a signle loop. Therefore, it is unwise to handle it without a thread, in case there are many events to be checked. For that purpose, TouchscreenEvents thread is created. The thread is able to emit events in case there is a mouse click or a key press.
- **UpdateUtil Thread:** This thread is able to read from core utilization log files and parse the information to update shared variables so that the Main Thread can show the results.

For the sake of informational completeness, how the schedulable Python threads are created should be depicted. Therefore, the Listing 3.13 is given and explained as follows.

```
 1  def Thread_Name():
 2     global aprocess_list
 3     global aprocess_list_len
 4     global SharedVariable2
 5
 6     #Initialize thread and append it to the global process list
 7     this_thread = aprocess.aprocess("Thread_Name", 1, "file.inc", 1, "Name", "None", 1)
 8     this_thread.UpdateThreadIDAndRunning()
 9     this_thread.SetCoreAffinityOfThread("0-3")
10     lock_aprocess_list.acquire()
11     aprocess_list.append(this_thread)
12     lock_aprocess_list.release()
13     aprocess_list_len = len(aprocess_list)
14
15     while True:
16        _thr_START_TIME = time.time()
17        _thr_PREV_SLACK_TIME = _thr_START_TIME - _thr_END_TIME
18        #TASK CONTENT starts here
19        # ...
20        #TASK CONTENT ends here
21        CreateTimingLog()
22        #Delay
23        if (_thr_PERIOD > _thr_EXECUTION_TIME):
24           time.sleep(_thr_PERIOD - _thr_EXECUTION_TIME)
```

Listing 3.13: Thread skeleton in Python

In Listing 3.13, a schedulable dummy thread skeleton is shown. Between lines 2 through 4, the shared variables are defined. Line 7 shows instantiating an `aprocess` object by entering the thread name, traceability, log file, displayability, display name, starting command, and whether or not if it is a thread, respectively. In the line 8 and line 9, the thread ID is updated and the core affinity of the thread is set. Lines 10 through 12 shows globally updating the `aprocess_list` with the created thread by making use of mutexes. Between lines 16 and 24, the schedulability and traceability features are implemented which was discussed in Section 3.3.2.

Since the touchscreen display process is responsible for displaying many information, libraries to gather up such information are used. Furthermore, the data that is gathered from the timing logs and the core usage logs have also been used in this application. Information that is gathered involve core usage percentages of both low-level and high-level modules, slack times of high-level processes, core frequency of high-level module, active cores count for the high-level module and a core mapping list from the high-level module. There is also ability to change the core frequency (Figure 3.39 (g)), and display the gross execution time. The detailed information on how the timing information is extracted is explained in Section 4.

3.3.9. VNC Server

The **Virtual Network Computing (VNC)** is a system that allows creating and managing virtual computers as well as connecting to them remotely [75]. While dealing with programming single board computers such as the Raspberry Pi, VNC is used for viewing the single board computer desktop remotely. During the development of A4MCAR, a third-party application called *XtightVNC* is installed to both the Raspberry Pi and the development computer in order to connect to each other without having to use external hardware. The VNC server that is installed at the Raspberry Pi, *XtightVNC*, is run at boot time and scheduled like any other process on the Raspberry Pi. While the server has not been manipulated during the development, in order to investigate the parallelism efficiency, this third party application should also be considered in order to get more accurate results. Further information regarding parallelism findings will be given in Chapter 4 and Chapter 5.

3.4. Android Application Implementation

To control the A4MCAR remotely via communicating with the RN42 bluetooth module that is connected to the low-level module, the A4MCAR control application is developed using Android [76] environment. In the Figure 3.2, one can see how the developed A4MCAR control application interacts with the entire software A4MCAR software.

As an integrated Android development environment, Android Studio [77] is used. Using Android Studio, developers can not only design XML-based user interfaces for their applications, but also describe the behavior of their programs using the Java programming language. Additionally, Android Studio can emulate many of the available Android devices to help the developers debug their software flexibly.

The developed Android application interface is given in Figure 3.40. In the figure, it is shown that the interface consists of a joystick and gear buttons that help in constructing the driving commands that are given by the Figure in 3.24. Furthermore, using a bluetooth device list, the A4MCAR can be paired with any compatible RN-42 bluetooth device in order to start data communication.

For the joystick controls, a third-party Android library that is called *virtual-joystick* [78] is used. Using this library, one can import the joystick mechanism into their applications. In order to handle the data created from the joystick, an on move event handler has been implemented which is a callback function to react on every joystick movement. Using this callback function, the angle and strength information that results from the joystick are transformed to conform the driving command (Figure 3.24). With the help of Figure 3.41, this data transformation can be explained easily.

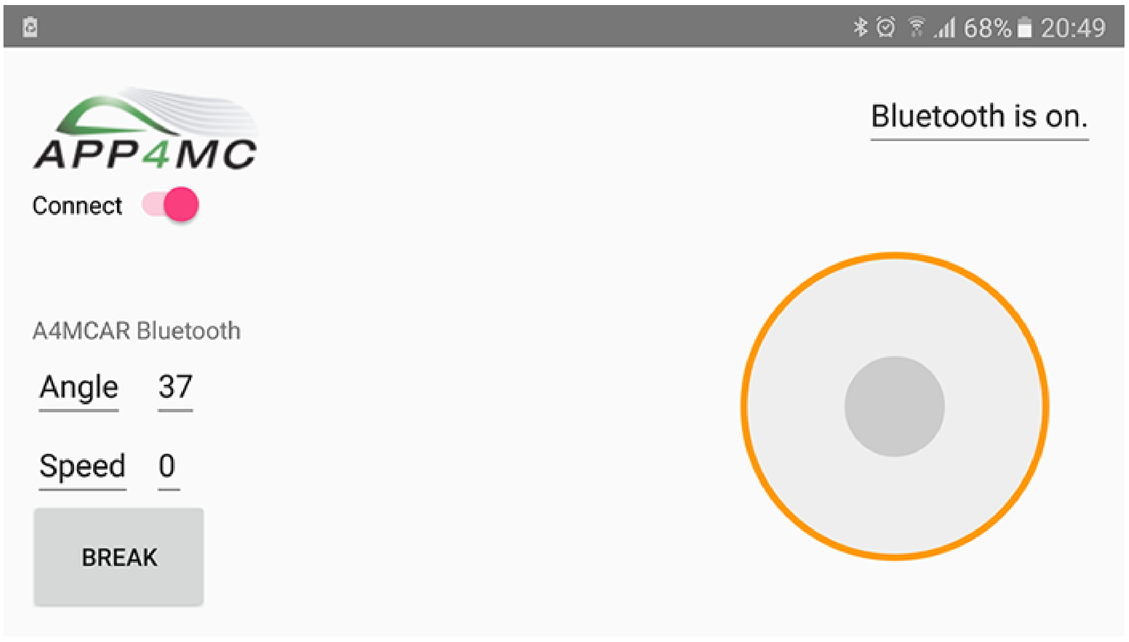


Figure 3.40.: Android Application Developed for Driving A4MCAR Remotely

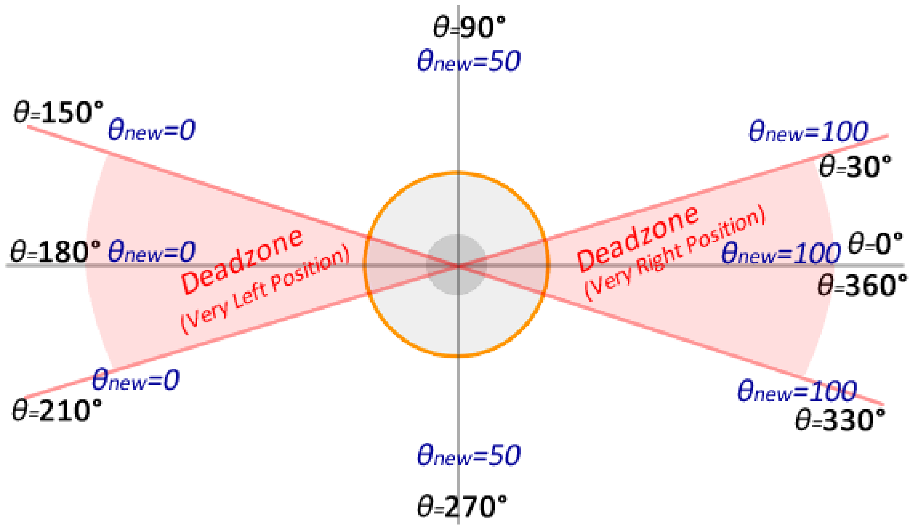


Figure 3.41.: Joystick angle transformation to construct driving command

Using the joystick illustration given in a Cartesian coordinate system, the angles that are generated by the joystick library itself θ (0 to 360 degrees) have been converted to the angles for the driving command format θ_{new} (0 to 100). Although the transformations are made to the values from 0 to 100, later on this is reduced to the values from 0 to 99 in order to get rid of the 3-digit format which is not accepted by the command parser written at the low-level module side.

As it is clearly shown in the Figure 3.41, there are certain dead zone regions on the joystick which are not taken into account in the transformation for the sake of the comfort of the users. While the transformed angle θ_{new} is constant in the dead-zone regions, a few equations have been used in order to handle the mapping of angles in the remaining regions:

- If the angle is between 30 degrees and 150 degrees, the transformed new angle is calculated using the following:

$$\theta_{\text{new}} = \left(1 - \frac{\theta - 30}{120}\right) 100 \tag{3.6}$$

- If the angle is between 210 degrees and 330 degrees, the transformed new angle is calculated using the following:

$$\theta_{\text{new}} = \left(\frac{\theta - 210}{120}\right) 100 \tag{3.7}$$

After the calculations, with the current gear setup the data is sent to the low-level module using the bluetooth functionality of the Android application.

4. Information Tracing and System Management

4.1. Introduction

After the distributed multi-core system is developed and defined, the evaluations regarding different software distributions are performed in order to parallelize the system efficiently. For that purpose, one should know how to manage the multi-core system. While managing the system is quite important to maintain and properly optimize the system to its full capabilities, one requires information regarding the system itself in order to achieve this optimization. As an example, well-known methodologies of extracting useful information from a software is to analyze an online and offline trace of the system.

In order to obtain information regarding a software such as number of instructions, how tasks are scheduled, timing details regarding tasks, core frequencies, energy consumption rates, the following techniques are most commonly used:

- **Static Binary Analysis:** Static binary analysis is a reverse engineering methodology that helps in finding errors in code such as non-determinism [79]. It is essential to analyze the binaries that are created from C and C++ programs to have another approach to traditional error finding methodologies such as testing and code inspection [79]. It is important to keep in mind that in the static binary analysis, the program is not executed [80]. Therefore, the information regarding execution and timing are not provided while the instruction information could be extracted [80]. However, it should be noted that some processor or platform specific tools can estimate the timing based on the number of instructions and the processor information. With the static binary analysis, the disassembly information, which is the list of all the instructions, can be found. With the help of the binary analyzer tools, detailed information on number of function calls, nesting, and cyclic complexity could be investigated [81].
- **Profiling (Dynamic Analysis):** Dynamic analysis, also called profiling, is the methodology of analyzing the program by considering its execution in contrast to the static

binary analysis [80]. Dynamic analysis is often done by using tools and it is done in order to get information of how a program is executed on a real or virtual processor [81]. While dynamic analysis is quite useful for identifying vulnerabilities in a runtime environment and obtaining information such as timing of execution, it can not guarantee the full test coverage of the source code [81]. Using dynamic binary instrumentation (DBI) tools for e.g. the Linux platform this way, one can obtain information such as CPU time, execution times, memory and I/O of a program [80].

- **Tracing:** Tracing is a methodology which is often mixed with profiling. According to IPM [82], a trace records the chronological information of the execution of a program or a system via logging the the execution with timestamps, whereas a profile is the collection of performance events and timings for a program's execution as a whole. Therefore, it can be said that the scheduling of an Operating system could be analyzed with the help of tracing.

  The A4MCAR involves tracing features that are not only supplied by Linux tools but also developed within the project. It can be generalized that online tracing is a type of tracing that is done while the program is being executed using buffered logs while offline tracing is done after the program has executed using the entire logs. Regarding this information, it can be commented that the developed tracing features are created for online tracing in A4MCAR while the existing tooling is used for offline tracing. The following sections consist of the information regarding tracing developments as well as the tooling support regarding tracing a Linux system.

- **System Monitoring:** Unlike static binary analysis and profiling, system monitoring [83] is done within the entire system and it is used for obtaining useful information regarding the system performance as a whole. Operating systems (especially the Linux platform) usually have system logs which could be observed via system monitoring in order to extract useful information concerning the system performance. Concerning the A4MCAR, the performance values such as core frequency and core utilization information are extracted using system monitoring.

The low-level and high-level modules of the A4MCAR requires several information related to APP4MC modeling. To start with, in order to model the software system of both modules, the number of instructions, task periods, and if exists event occurrance types are needed. This could be achieved by using static binary analysis method in the low-level module due to the fact that the XTA tool can estimate instructions and timingfrom the static binary analysis. The information of number of instructions and periods are obtained from the high-level module Linux platform by using tools that perform profiling. Although using a static binary analysis tool is possible, using a profiler tool was selected as the easier option. Secondly, the system monitoring is needed in both modules. While the system monitoring is done using registers in the memory for the low-level module, it is achieved by using Linux kernel tools for the high-level module. System monitoring for the A4MCAR is essentially needed in order to obtain core utilization percentages, real-time CPU clock information and active core count. Finally, the profiling and tracing of the programs of the high-level module is needed in order to evaluate parallelization performance and visualize how processes are scheduled. The data that is obtained from profiling and tracing involve slack time, execution times, start and end times.

In following sections of this chapter, the aforementioned techniques for system analysis are discussed with the emphasis of their applicability on a real distributed multi-core system that involves elements from a low-level multi-core micro-controller and a high-level single board computer that is running on x86/Linux platform in order to elaborate modeling, managing, profiling, tracing of systems and the evaluation of various software distributions.

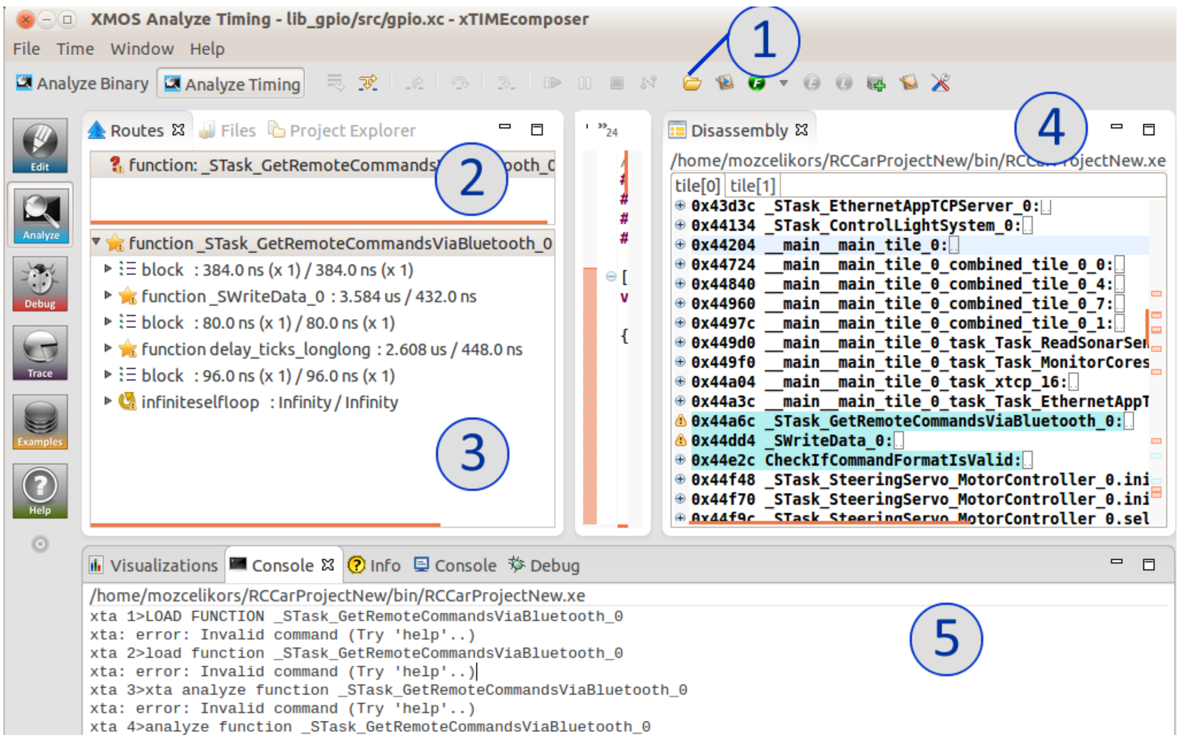
4.2. Low-Level Module Information Tracing and System Management

4.2.1. Static Binary Analysis via XTA

The XMOS Timing Analyzer (XTA) [84] is an Eclipse-based tool that comes with the xTIME-composer platform which is used for analyzing the timing and the execution details of the multi-tasked software that is developed using the XMOS boards and processors [84]. The tool is able to measure shortest and longest time required to execute a section of code by analyzing the binary file. Thus, the code is not executed in order to be analyzed. Furthermore, it is also able to check the minimum and maximum number of instructions required to execute a section of the code. A screenshot from the XTA tool is given in the Figure 4.1.

Once the code is written and built in the xTIMEcomposer platform, a binary file is generated with the extension `.xe`. By loading this binary file using XTA, the timing analysis can be performed. In order to load the binary to XTA, the *Load Binary* button (shown as 1 in the Figure 4.1) must be pressed. Once the binary is loaded, the *Disassembly window* (shown as 4) appears which shows all the runnables that are automatically detected from the binary. Using the disassembly window, the instructions that are used for each runnable can be denoted.

The XTA tool is also able to work with specific commands by using the XTA console (shown as 5). Using the specific commands taken from the XTA manual [84], timing analysis could be performed easily. In order to start timing analysis, an execution path should be defined. The execution path can be analyzed using the following possible ways [84]:



```
function: _STask_GetRemoteCommands        ooth_0
function _STask_GetRemoteCommandsViaBluetooth_0
  block  : 384.0 ns (x 1) / 384.0 ns (x 1)
  function _SWriteData_0 : 3.584 us / 432.0 ns
  block  : 80.0 ns (x 1) / 80.0 ns (x 1)
  function delay_ticks_longlong : 2.608 us / 448.0 ns
  block  : 96.0 ns (x 1) / 96.0 ns (x 1)
  infiniteselfloop  : Infinity / Infinity
```

```
/home/mozcelikors/RCCarProjectNew/bin/RCC        ojectNew.xe
0x43d3c _STask_EthernetAppTCPServer_0:
0x44134 _STask_ControlLightSystem_0:
0x44204 __main__main_tile_0:
0x44724 __main__main_tile_0_combined_tile_0_0:
0x44840 __main__main_tile_0_combined_tile_0_4:
0x44960 __main__main_tile_0_combined_tile_0_7:
0x4497c __main__main_tile_0_combined_tile_0_1:
0x449d0 __main__main_tile_0_task_Task_ReadSonarSen
0x449f0 __main__main_tile_0_task_Task_MonitorCores
0x44a04 __main__main_tile_0_task_xtcp_16:
0x44a3c __main__main_tile_0_task_Task_EthernetAppT
0x44a6c _STask_GetRemoteCommandsViaBluetooth_0:
0x44dd4 _SWriteData_0:
0x44e2c CheckIfCommandFormatIsValid:
0x44f48 _STask_SteeringServo_MotorController_0.ini
0x44f70 _STask_SteeringServo_MotorController_0.ini
0x44f9c _STask_SteeringServo_MotorController_0.sel
```

```
/home/mozcelikors/RCCarProjectNew/bin/RCCarProjectNew.xe
xta 1>LOAD FUNCTION _STask_GetRemoteCommandsViaBluetooth_0
xta: error: Invalid command (Try 'help'..)
xta 2>load function _STask_GetRemoteCommandsViaBluetooth_0
xta: error: Invalid command (Try 'help'..)
xta 3>xta analyze function _STask_GetRemoteCommandsViaBluetooth_0
xta: error: Invalid command (Try 'help'..)
xta 4>analyze function _STask_GetRemoteCommandsViaBluetooth_0
```

Figure 4.1.: XMOS Timing Analyzer (XTA) screenshot

- Via placing end points into the code using compiler directive #pragma as shown in Listing 4.1.

```
#pragma xta endpoint "start_endpoint1"
   data = DoSomeCalculations();
#pragma xta endpoint "stop_endpoint1"
```

Listing 4.1: Placing end points in xC code to define an execution path

The timing analysis between two endpoints can be started by entering the command in Listing 4.2 to the XTA console:

```
   analyze endpoints start_endpoint1 stop_endpoint1
```

Listing 4.2: Placing endpoints in XTA

- A function or a runnable with the name `Function_Name` can be analyzed from its starting point to its return point by using the command in Listing 4.3:

```
   analyze function Function_Name
```

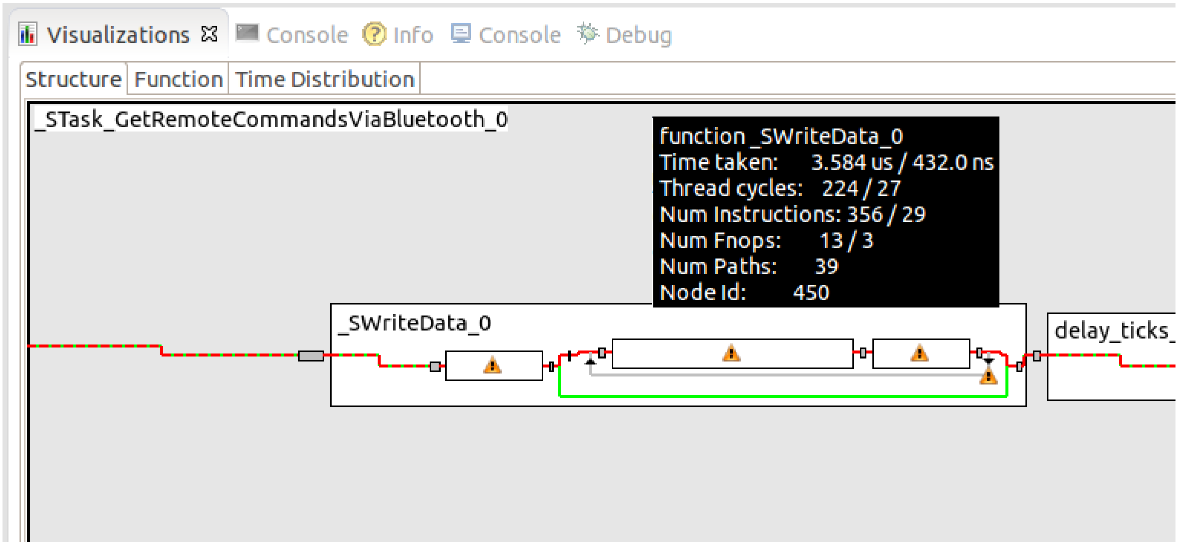
Listing 4.3: Analyzing a function in XTA

- Finally, loops can also be investigated using XTA. The way a loop is defined is either setting a loop point from the editor or defining an endpoint inside the loop. A loop point having an end point `looppoint` can be analyzed using the command in Listing 4.4 in the XTA console:

```
analyze loop looppoint
```

Listing 4.4: Analyzing a loop in XTA

The entire code can be modified in the aforementioned fashion in order to do a timing analysis to all of the runnables of the software system. One can set up timing constraints to make sure every dead line is not violated [84]. Once the timing analysis starts, the selected route, i.e. a function, a loop, or a route between two endpoints is shown in the *Routes window* (shown as (2) in the Figure 4.1). The selected route is shown in blocks in another window which is shown as (3) in Figure 4.1). It is shown that the best case execution time and the worst case execution time of each block is estimated. As an example, for the `WriteData` function, the worst case execution time is estimated to be 3.584 microseconds, whereas the best case execution time is 432 nanoseconds. Further information that is provided by XTA regarding timing analysis is given in the Figure 4.2. As it is shown in figure, what kind of timing paths could have been taken for the function can be visualized using the *Visualizations* window. Furthermore, information such as thread cycles, number of instructions, number of Fnops (NOP Instructions on the floating point unit), and number of paths are also shown.



```
function _SWriteData_0
Time taken:      3.584 us / 432.0 ns
Thread cycles:   224 / 27
Num Instructions: 356 / 29
Num Fnops:       13 / 3
Num Paths:       39
Node Id:         450
```

Figure 4.2.: XTA Visualizations window with further information

For modeling purposes in APP4MC the number of instructions are obtained for almost every runnable and event by using aforementioned techniques. For the events, the technique of defining endpoints is used, whereas for the runnables the function analysis is used. However, due to the non-determinisim in the some of the branch instructions and code sections that are related to hardware communication, some sections in many runnables were not able to be analyzed properly and returned the `Unresolved` error. In order to get rid of this error, the following techniques are used:

- The `Unresolved` error occurs usually when XTA can not resolve a branch instruction whose branch target is unknown [84]. In order to get rid of this issue, the trace of the system should be printed and branch instruction target (branch position) should be pointed manually. The details of how this is done is given in the XTA manual [84]. By using this technique, some of the `Unresolved` errors were resolved and number of instructions of those runnables were found.
- When the technique above failed to work because of memory instructions such as `memset` and infinite loops, the number of instructions are gathered by counting the instructions from the *Disassembly window*. Although this technique is time consuming and error-prone, it is assumed that the gathered information is close to the real number of instructions. Additionally, APP4MC's partitioning and mapping accuracy would not change significantly if the gathered information was not exact.

In Section 5, the gathered information along with its implications are further discussed.

4.2.2. Distribution of Tasks to Cores

In a properly utilized parallel system, gathered information is used in order to find which software distribution is most efficient. Here, the software distribution refers to the mapping stage, which is the distribution of the tasks that result from APP4MC's partitioning process.

Within the xTIMEcomposer platform, the task mapping is done easily with the capabilities of the xC programming language. Since xCORE provides a multi-core platform, using the cores for different tasks can be achieved by using simple statements. Although this is introduced in Section 3, some of the information is summarized for completeness purposes. In XMOS, placement of a function into a core is done by using the `par` statement. In the main block of a software, the `par` statement can be used in order to create several tasks in parallel. Additionally, using global interface variables, one can handle the inter-task communication between two parellel tasks.

As opposed in Listing 3.2, a simple example in Listing 4.5 help one to get a better impression of how this works using xC:

```
1 int main(void)
2 {
3    interface my_interface i1;
4    par
5    {
6       on tile[0].core[2] : Task1 (i1);
7       on tile[1].core[3] : Task2 (i2);
8    }
9 }
```

Listing 4.5: Interfacing in XMOS

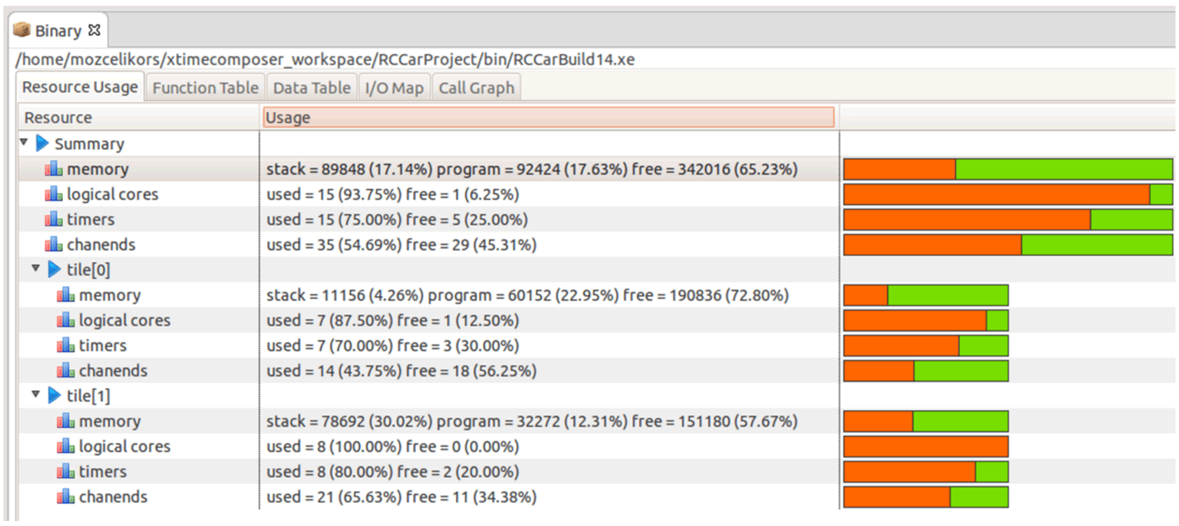
The given code is a basic example of using xC functionality to do a task mapping. The parallel block in the code is given as the Lines 4 through 8. It is shown that the *Task1* is pinned to the core 2 of tile 0, whereas the *Task2* is pinned to the core 3 of the tile 1. Furthermore, a global interface of type `my_interface` having the name i1 is declared in Line 3. This interface handles the communication between the tasks *Task1* and *Task2*. As mentioned, the hardware realizes the interface by using the xCONNECT switches to construct a bridge between the tiles and cores. Additionally, it should be noted that unlike the Raspberry Pi, the tasks are distributed to cores during the reconfiguration in a xC program, i.e where a Task is located can not be changed during run-time due to the nature of xCORE and due to the fact that hardware availability differs from tile to tile [40].

By using the aforementioned technique, all the low-level module tasks are distributed to cores at compile-time (or build-time).

4.2.3. System Monitoring in xCORE

How cores of the system is monitored in xCORE are discussed in the Chapter 3. For the sake of completeness, this subsection is dedicated to summarizing what is discussed in the Section 3.2.7. System monitoring is done by polling a system register and finding out if that core is busy or idle at that moment. Referring back to the Listing 3.7, one can understand this process better.

Besides the implemented core monitoring task, XMOS features several more means to monitor the system. First, the system can be monitored at build-time using the `Resource Usage tab` at the XTA tool *Binary window*. In Figure 4.3, it is shown that the information regarding stack memory, program memory, free memory, cores, timers, and channels could be observed in one window using XTA Binary. Another way to monitor the system is provided by a function called `debug_printf`. As the name of the function suggests, it is essentially a `printf` function to observe variables. However, `debug_printf` is a function that does not interrupt inter-process communication and that does not block cores while printing so that developers can monitor without having to worry about much overhead [40].


Binary

/home/mozcelikors/xtimecomposer_workspace/RCCarProject/bin/RCCarBuild14.xe

Resource Usage | Function Table | Data Table | I/O Map | Call Graph

Resource	Usage
Summary	
memory	stack = 89848 (17.14%) program = 92424 (17.63%) free = 342016 (65.23%)
logical cores	used = 15 (93.75%) free = 1 (6.25%)
timers	used = 15 (75.00%) free = 5 (25.00%)
chanends	used = 35 (54.69%) free = 29 (45.31%)
tile[0]	
memory	stack = 11156 (4.26%) program = 60152 (22.95%) free = 190836 (72.80%)
logical cores	used = 7 (87.50%) free = 1 (12.50%)
timers	used = 7 (70.00%) free = 3 (30.00%)
chanends	used = 14 (43.75%) free = 18 (56.25%)
tile[1]	
memory	stack = 78692 (30.02%) program = 32272 (12.31%) free = 151180 (57.67%)
logical cores	used = 8 (100.00%) free = 0 (0.00%)
timers	used = 8 (80.00%) free = 2 (20.00%)
chanends	used = 21 (65.63%) free = 11 (34.38%)

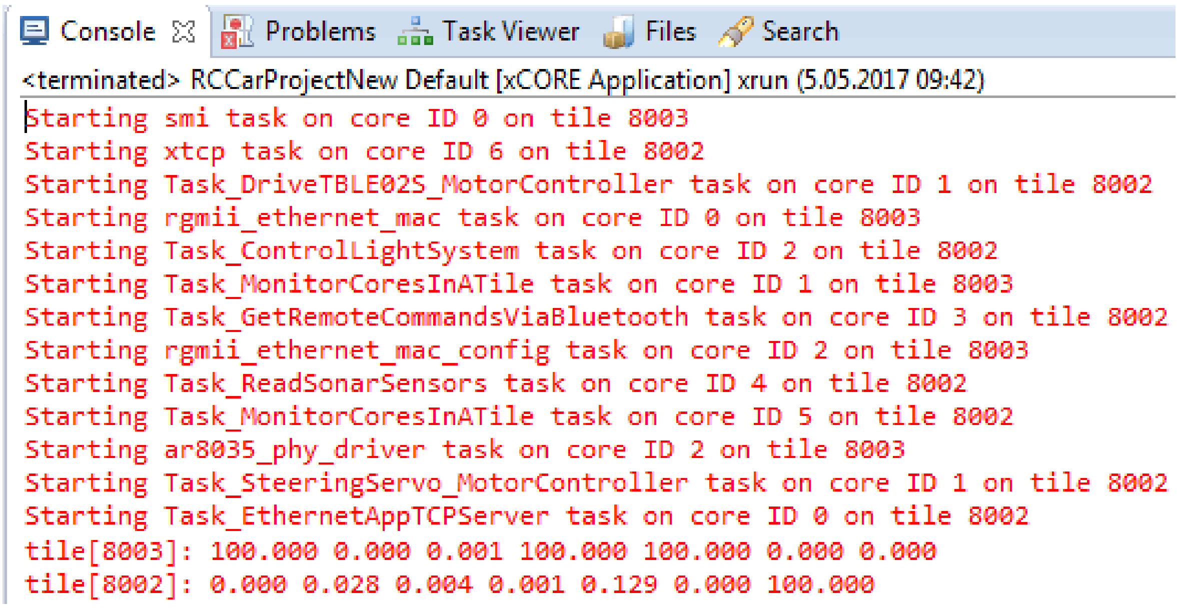
Figure 4.3.: XTA Binary Resource Usage

Using the `debug_printf` function and xC-specific tools, a function is created that helps to find out which function refers to which core. The reason this information is important is because the core IDs that are referred with the `par` statements are not the same IDs of the core usage implemention. The same issue holds true for the tile IDs as well. By using the code in Listing 4.6, this information can be monitored at run-time:

```
int PrintCoreAndTileInformation(char * Function_Name)
{
   debug_printf("Starting␣%s␣task␣on␣core␣ID␣%d␣on␣tile␣%x\n", Function_Name,
       get_logical_core_id(), get_local_tile_id());
   return 1;
}
```

Listing 4.6: Printing core and tile information

Here, the `get_logical_core_id()` is the function to get the real core ID from system registers, whereas the `get_local_tile_id()` function returns the real tile ID. This way, it is made sure which task uses how many percentage of the core. Once every task is manipulated so that they use this function while the core monitoring is running, system monitoring can be done easily by observing the *Console*. An example output of the *Console* from the final version of the low-level module software is given in Figure 4.4. It is shown that the core IDs for every task is given and then the core utilization percentages are listed for cores 0 through 7 for each tile.



```
<terminated> RCCarProjectNew Default [xCORE Application] xrun (5.05.2017 09:42)
Starting smi task on core ID 0 on tile 8003
Starting xtcp task on core ID 6 on tile 8002
Starting Task_DriveTBLE02S_MotorController task on core ID 1 on tile 8002
Starting rgmii_ethernet_mac task on core ID 0 on tile 8003
Starting Task_ControlLightSystem task on core ID 2 on tile 8002
Starting Task_MonitorCoresInATile task on core ID 1 on tile 8003
Starting Task_GetRemoteCommandsViaBluetooth task on core ID 3 on tile 8002
Starting rgmii_ethernet_mac_config task on core ID 2 on tile 8003
Starting Task_ReadSonarSensors task on core ID 4 on tile 8002
Starting Task_MonitorCoresInATile task on core ID 5 on tile 8002
Starting ar8035_phy_driver task on core ID 2 on tile 8003
Starting Task_SteeringServo_MotorController task on core ID 1 on tile 8002
Starting Task_EthernetAppTCPServer task on core ID 0 on tile 8002
tile[8003]: 100.000 0.000 0.001 100.000 100.000 0.000 0.000
tile[8002]: 0.000 0.028 0.004 0.001 0.129 0.000 100.000
```

Figure 4.4.: System monitoring implemented on xCORE

4.2.4. Discovering Energy Consumption Features

One of the most important optimization goals include reducing the energy consumption. To reduce the energy consumption one should have a properly utilized software which is achieved through balancing the CPU load through all the cores of the system. In order to understand this, Power Consumption Application Manual for XS1-L devices [85] can be studied further.

There are two types of power consumption described regarding a processing unit. Static power consumption describes a chip's power consumption that is caused by the leakage current as the chip is heating [9]. Therefore, since the leakage current could not be controlled by a user directly, dynamic power consumption, which is the power consumption that is resulted from actual computations, is the concern of this thesis' research. The dynamic power consumption of a chip is described by the following equation [86]:

$$P_{\text{dynamic}} = \alpha C_{\text{L}} {V_{\text{DD}}}^2 f_{\text{clk}} \tag{4.1}$$

We can see from the equation that the power depends on a switching probability α, a chip voltage V_{DD}, a clock frequency f_{clk} and a collective switching capacitance C_{L}. It is stated in the [9] that since V_{DD} linearly depends on the clock frequency f_{clk}, a cubic relationship between power consumption and clock frequency can be observed. The linear relationship between the current I_{DD} and the frequency is also depicted in Figure 4.5, showing the performance values for a XS1-L device of XMOS xCORE-200 eXplorerKIT [85].

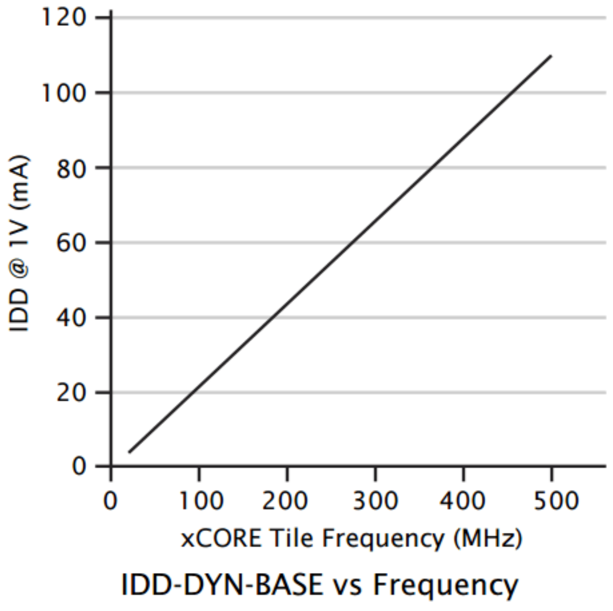


Figure 4.5.: XS-1 Power Graph Related to Base Current for an xCORE Core

In Figure 4.5, the base power consumption of one xCORE running an instruction sequence is given with respect to the clock frequency. One can see that the internal base current that is related to the operation of xCORE is directly proportional to the clock frequency of the xCORE core. Since the current is directly proportional to the power, as seen by the equation i.e. $P = I_{\text{DD}}V_{\text{DD}}$, it can be concluded that load balancing is one of the most important things to take care of to get a lower frequency thus current in the core, thus reducing the power consumption. It is important to add at this point that power consumption and energy consumption are directly proportional if the system is not fully loaded. In other words, decreasing the clock frequency alone is useful to achieve reduced energy consumption in a system in which the load is balanced and no deadlines are missed [86]. Using the provided internal registers, the xCORE core and tile frequencies can be reduced according to the manual [85].

It is important to keep in mind that reducing the clock speed is not the only way to reduce the power consumption. If the dynamic switching power is considered rather than the base power consumption, the factors such as operating frequency, amount of communication, and the data itself are all big causes of power consumption [85]. Therefore, two techniques with which power and energy is reduced can be defined. The first technique encapsulates the aforementioned techniques, i.e decreasing the frequency of a chip dynamically by using DFS (Dynamic Frequency Scaling) and decreasing the frequency and operating voltage together dynamically (Dynamic Voltage and Frequency Scaling). The second technique involves shutting down the chip sections which are not used dynamically.

An important energy related feature that comes with XS-1 devices is the AEC (Active Energy

Consumption) mode [85] which can be an example of the mixture of both of the aforementioned techniques. When this mode is turned on and AEC mode clock frequency is set to a desired value, xCORE device lowers the clock frequency of the AEC-enabled cores to the desired AEC mode clock frequency when the core is paused or waiting for an input [85]. This could be slightly related to DVFS (Dynamic Voltage and Frequency Scaling) [87] which is a processors ability to dynamically scale its frequency and operating voltage depending on the load. DVFS is common for operating systems such as Linux OS and it will be discussed in the Section 4.3.

4.3. High-Level Module Information Tracing and System Management

4.3.1. Obtaining Information

4.3.1.1. Binary Analysis of Instructions

In order to obtain the number of instructions of the created processes on the high-level side for modeling purposes, a couple of options are investigated. Most of the techniques are used for the A4MCAR in order to model the system using APP4MC.

As a static binary analysis solution, *objdump* [88] module of Linux provides disassembly information of C-based libraries and executables. After the compilation of a C/C++ program, the GNU C Compiler (GCC) provides an object file which contains the binary data for the program. Using the *objdump* module, C/C++ programs, functions, libraries can be analyzed easily. *Objdump* provides not only the disassembled instruction list but also information such as symbol tables and debug information. In order to record the instructions of a C/C++ application, one of the commands in Listing 4.7 are used.

```
objdump MyObject.o -D > MyObjectDump.txt
objdump MyObject.o -S > MyObjectDump.txt
```

Listing 4.7: Using objdump

These commands provide disassembled instructions from `MyObject.o` and records them into a file called `MyObjectDump.txt`. The command with the `-D` option provides the complete disassembly information for each function whereas the command with the `-S` option provides source code along with instructions.

Although using `objdump` is useful for C/C++ applications, it does not provide any means to investigate Python-based processes. To get the instruction information for Python-based processes Python library `dis` [89] provides a couple of functions. These functions must be used in the Python program itself in order to print out or record the instruction information. A simple example of using `dis` in a Linux shell is given in Listing 4.8.

```
python -m dis PythonApp.py
```

Listing 4.8: Using dis

This command will give bytecodes of each code line, hence giving instructions for every line of code. One can also disassemble functions by using the Python code in Listing 4.9.

```
import dis
dis.dis(FunctionName)
```

Listing 4.9: Using dis in Python shell

Perf [90] [91] is a well-known lightweight system performance counter and profiler tool for Linux. Using perf, one can obtain event and instruction counts, record events, run benchmarks, and analyze processes [90]. For the A4MCAR, the Perf profiler tool has been used for many purposes that include process instruction analysis, system-wide scheduling tracing, and trace data conversion. Perf can perform dynamic analysis (profiling) in order to obtain the number of instructions for processes and runnables using Linux shell command shown in the Listing 4.10.

```
sudo perf stat -e instructions:u -p <pid>
```

Listing 4.10: Using perf stat

It must be noted that the number of instructions are obtained dynamically so the process should either be exited or should have a finite number of iterations for the result to be accurate. It must also be made sure that since the command is used for counting the number of instructions of a running process, the command should be executed right after the process starts. In the listing, command counts the instructions only in user mode to avoid including any system overhead and the `<pid>` is the Process ID of the process according to the Linux kernel. How a process ID of a process is obtained and what it means is discussed in the next section. For the analysis of high-level applications in A4MCAR, the following variation of the perf stat command is used which is able to determine the instruction count of a running process or thread with a timeout. If the timeout is set to the period of the schedulable process or thread, one can obtain rough idea about the granularity of the process or thread. In Listing 4.11, `<timeout>` is the period in seconds.

```
sudo perf stat -e instructions:u -p <pid> -- sleep <timeout>
```

Listing 4.11: Using perf stat with timeout

For threads, the aforementioned method is error-prone, therefore using `-per-thread` switch is more reliable when profiling threads of a process as shown in Listing 4.12.

```
sudo perf stat --per-thread -e instructions:u -p <pid> -I <timeout>
```

Listing 4.12: Using perf stat with per thread switch

4.3.1.2. Process Management and Monitoring

Managing and monitoring processes and threads of a the Linux system are crucial preliminaries that should be discussed in order to work with A4MCAR's high-level module using Linux platform. One can list the process management and monitoring issues as follows:

- **Listing Processes and Threads:** By using the `top` command, processes and threads that are running can be listed. A couple of example commands and their outputs are explained below [92]. By using the command in Listing 4.13, processes of the entire system could be monitored.

```
    top
```

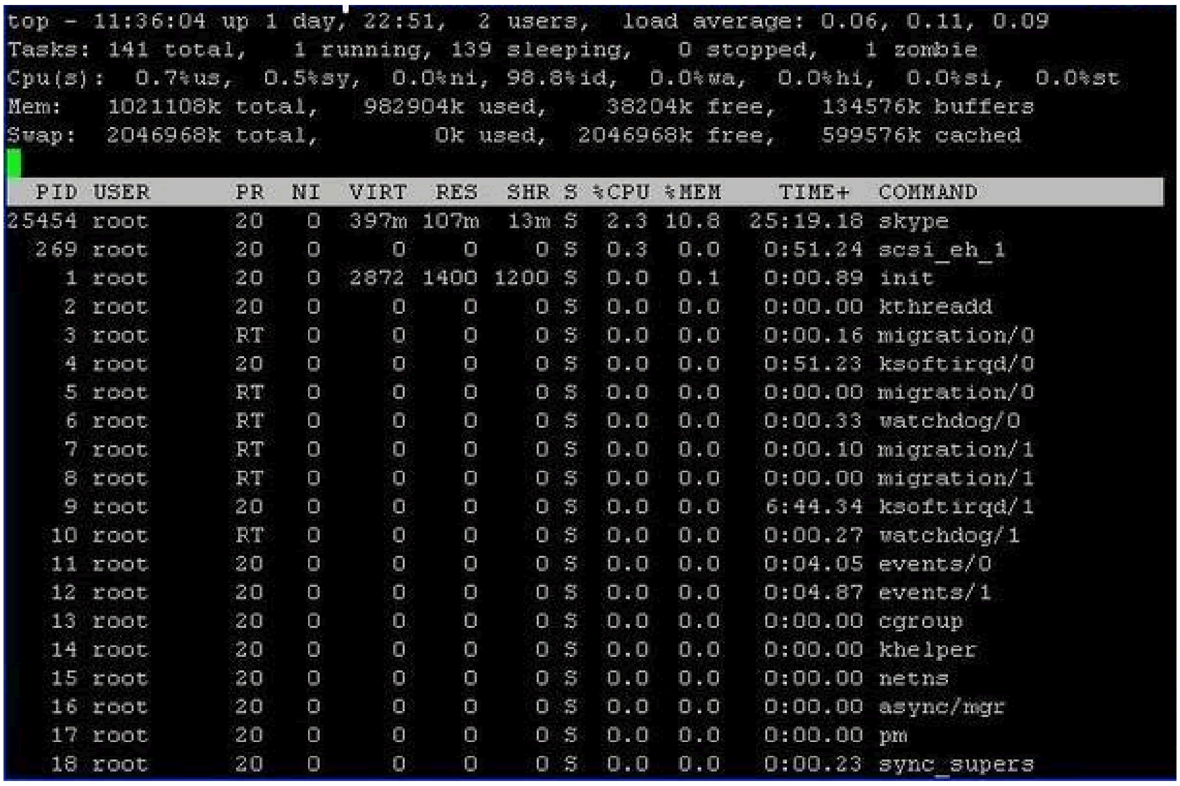
Listing 4.13: Top command in Linux shell

  In, 4.6 each process are listed in descending order according to their CPU usages. Processes could be identified by looking at their commands. Information such as *owned user (USER)*, *process ID (PID)*, how much *virtual memory are accessed (VIRT)*, *physical memory usage (RES and MEM)*, *how much virtual memory is shared (SHR)*, *cpu usage (CPU)* can be monitored in this window. By using the command in Listing 4.14, one could also see the threads of a process given its process ID.

```
  top -H -p <pid>
```

Listing 4.14: Using top to monitor threads

  Another way to manage processes is done by using `ps` command. This command not only allows to list processes or threads but also is used to kill processes. Similarly to `top` command, `ps` command could be used like the Listing 4.15 in order to list processes and threads.



```
top - 11:36:04 up 1 day, 22:51,  2 users,  load average: 0.06, 0.11, 0.09
Tasks: 141 total,   1 running, 139 sleeping,   0 stopped,   1 zombie
Cpu(s):  0.7%us,  0.5%sy,  0.0%ni, 98.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1021108k total,   982904k used,    38204k free,   134576k buffers
Swap:  2046968k total,        0k used,  2046968k free,   599576k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
25454 root      20   0  397m 107m  13m S  2.3 10.8  25:19.18 skype
  269 root      20   0     0    0    0 S  0.3  0.0   0:51.24 scsi_eh_1
    1 root      20   0  2872 1400 1200 S  0.0  0.1   0:00.89 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.16 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:51.23 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    6 root      RT   0     0    0    0 S  0.0  0.0   0:00.33 watchdog/0
    7 root      RT   0     0    0    0 S  0.0  0.0   0:00.10 migration/1
    8 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/1
    9 root      20   0     0    0    0 S  0.0  0.0   6:44.34 ksoftirqd/1
   10 root      RT   0     0    0    0 S  0.0  0.0   0:00.27 watchdog/1
   11 root      20   0     0    0    0 S  0.0  0.0   0:04.05 events/0
   12 root      20   0     0    0    0 S  0.0  0.0   0:04.87 events/1
   13 root      20   0     0    0    0 S  0.0  0.0   0:00.00 cgroup
   14 root      20   0     0    0    0 S  0.0  0.0   0:00.00 khelper
   15 root      20   0     0    0    0 S  0.0  0.0   0:00.00 netns
   16 root      20   0     0    0    0 S  0.0  0.0   0:00.00 async/mgr
   17 root      20   0     0    0    0 S  0.0  0.0   0:00.00 pm
   18 root      20   0     0    0    0 S  0.0  0.0   0:00.23 sync_supers
```

Figure 4.6.: Top command output

```
ps -aux            #All processes
ps -T -p <pid>     #Threads of a process
ps H -p <pid> -o 'pid tid cmd comm' #Threads of a process including their names
```

Listing 4.15: Using ps in Linux shell

- **Obtaining Process ID of a Process:** Identifying the process ID or a process is crucial in order to work with processes in Linux. By using the command in Listing 4.16, the PID of a process can be obtained by the process name.

```
   pgrep -f <process_name> -n
```

Listing 4.16: Using pgrep

Using this knowledge, a Linux bash script has been created that monitors a process by finding the process ID from the process name and then using perf profiler. The script is given in Listing 4.17.

```
1 #!/bin/bash
2 args=("$@")
3 process_name=${args[0]}
4 pid=$(pgrep -f $process_name -n ) #Newest result
5 sudo perf stat -p $pid
```

Listing 4.17: Created Bash script to dynamically profile applications (AppMonitor.sh)

By calling `bash AppMonitor.sh <process_name>` from the Linux shell, this script is used to monitor applications using perf. In the script, thecommand argument is retrieved (Line 2 and Line 3), process ID is obtained (Line 4) and then perf stat is used (Line 5).

```
1 def CheckIfProcessRunning(process_name):
2     # Returns process id, or 0 if process not running
3     try:
4         x = subprocess.check_output(['pgrep','-f',process_name,'-n'])
5     except Exception as inst:
6         x = 0
7     return x
```

Listing 4.18: Function to obtain process ID from Python environment

Since the touchscreen display process is responsible of doing all the online timing calculations, a function has been implemented which returns the PID of a process if the process is running (Listing 4.18). In the following, it is seen that using *subprocess* module of Python, one can check the output of a Linux shell command from Python environment (Line 4 of Listing 4.18).

- **Killing a Process:** Killing a process is handled through a simple Linux shell command that is given in Listing 4.19.

```
1     sudo kill -9 <pid>
```

Listing 4.19: Killing processes from Linux shell

Using the same manner that is explained by the Listing 4.17, a Linux shell script that is called `KillProcess.sh` has been created which is able to kill running process by their names.

- **Monitoring Process Details using The inux kernel folders:** Linux kernel provides a virtual filesystem that is located under `/proc` directory that contain runtime system information for system, device, connectivity, and process monitoring [93]. Regarding process monitoring, using the process ID as the folder name and simply viewing the files that are located under /proc/<pid>/, much information such as process status (observed in Figure 3.9), memory maps, libraries and executables, executed cpu, scheduling information can be monitored. A few examples are given in the Listing 4.20 with their explanations [93].

```
cat /proc/<pid>/cmdline #Command line arguments.
cat /proc/<pid>/cpu   #Current and last cpu in which it was executed.
cat /proc/<pid>/cwd   #Link to the current working directory.
cat /proc/<pid>/exe   #Link to the executable of this process.
cat /proc/<pid>/maps  #Memory maps to executables and library files.
cat /proc/<pid>/mem   #Memory held by this process.
cat /proc/<pid>/root  #Link to the root directory of this process.
cat /proc/<pid>/statm #Process memory status information.
cat /proc/<pid>/status #Process status in human readable form.
```

Listing 4.20: Kernel virtual filesystem /proc information retrieval examples [93]

4.3.1.3. System Monitoring for Linux Platform

As mentioned, for system monitoring investigating the /proc folder is also widely used. The information that is stored in the virtual filesystem involve the following [93]:

- Advanced power management info
- Information about the processor, such as its type, make, model, and performance
- List of device drivers configured into the currently running kernel
- Filesystems configured/supported into/by the kernel
- Which interrupts are in use, and how many of each there have been
- Memory map
- Masks for irq (interrupt request line) to cpu affinity
- Kernel locks
- Information about memory usage, both physical and swap
- Mounted filesystems
- Status information about network protocols

Although regarding the Linux system administration those information are useful, a more easy way to obtain certain information can be done through using third party modules. For the A4MCAR, *psutil* Python module has been used in order to monitor system information such as number of active cores running, core frequencies and CPU utilization of each core. Using the function given in Listing 4.21, the core frequencies are extracted in MHz.

```
str(psutil.cpu_freq()).split(',')[0].split('=')[1]
```

Listing 4.21: Using psutil to get CPU frequencies

Similarly, the `psutil.cpu_count()` function can be used to extract the number of active cores and the `psutil.cpu_percent()` function can be used to extract the core utilization percentages in a given time period [67].

4.3.1.4. Tracing the System to Obtain Scheduling Information

To evaluate performance indicators and observe load balancing, tracing the high-level module processes is crucial for the A4MCAR. By recording a system trace and using scheduling visualization tools in Linux, one can see how processes and threads are distributed amongst the existing cores. For that purpose, tracing and visualization options are investigated as follows:

- **Tracing via perf and viewing the trace:** In order to trace the system using perf profiler, perf's record command is used [91]. As an example, the system trace could be obtained for 15 seconds by entering the command in Listing 4.22 into the Linux shell.

```
    sudo perf sched record -- sleep 15
```

Listing 4.22: Using perf sched record

Once the tracing is done, the system trace is recorded to a file called `perf.data`. This trace file uses the perf tracing format which is not a common format. Therefore, it is not recognized by many of the trace visualization software. In order to visualize a basic system trace using perf.data file, the command given in Listing 4.23 could be used which saves the full trace in a text file called `fulldump.txt`.

```
    sudo perf sched script > fulldump.txt
```

Listing 4.23: Using perf sched script to get full dump of scheduling in Linux

First lines of the `fulldump.txt` should be analyzed in order to understand what information can be inferred from the trace. Referring to the code given in the Listing 4.24, each line represents a kernel event. If a task is being started to execute on a core that event is referred to as a `sched_switch` event, whereas if a task that was in the sleeping state is being executed again this is referred to as the `sched_wakeup` event. Another information regarding trace events involve *process name (comm)*, *process ID (pid)*, *target core (target_cpu)* and *time at which the event occurred*.

```
perf 16984 [005] 991962.879960: sched:sched_stat_runtime: comm=perf pid=16984 runtime
    =3901506 [ns] vruntime=165...
perf 16984 [005] 991962.879966:   sched:sched_wakeup: comm=perf pid=16999 prio=120
    target_cpu=005
perf 16984 [005] 991962.879971:   sched:sched_switch: prev_comm=perf prev_pid=16984
    prev_prio=120 prev_stat...
perf 16999 [005] 991962.880058: sched:sched_stat_runtime: comm=perf pid=16999 runtime
    =98309 [ns] vruntime=16405...
....
```

Listing 4.24: `Perf sched script` command output [91]

```
                *A0          993552.887633 secs A0 => perf:26596
*.               A0          993552.887781 secs . => swapper:0
.               *B0          993552.887843 secs B0 => migration/5:39
.               *.           993552.887858 secs
.                .  *A0      993552.887861 secs
.               *C0  A0      993552.887903 secs C0 => bash:26622
.               *.   A0      993552.888020 secs
.        *D0     .   A0      993552.888074 secs D0 => rcu_sched:7
.        *.      .   A0      993552.888082 secs
....
```

Listing 4.25: `Perf sched map` command output [91]

Since the `perf sched script` output might be messy for a system trace, one could be more interested in seeing a system trace in a more abstract form. In that regards, obtaining a cpu mapping view of the trace could help. In order to obtain a cpu mapping view, the command in Listing 4.26 is used from the Linux shell.

```
     sudo perf sched map
```

Listing 4.26: Obtaining CPU mapping view using perf

The output of this command can be seen in Listing 4.25. In the code, it is shown that each column represents a core whereas vertical axis is the time axis. In other words, each time an event occurs, a new line is added and the task is placed to a column depending on which core it is running. It should be noted that star symbol near a task is used to indicate `sched_switch` events. The timing information, the process name and process ID are also given in this view.

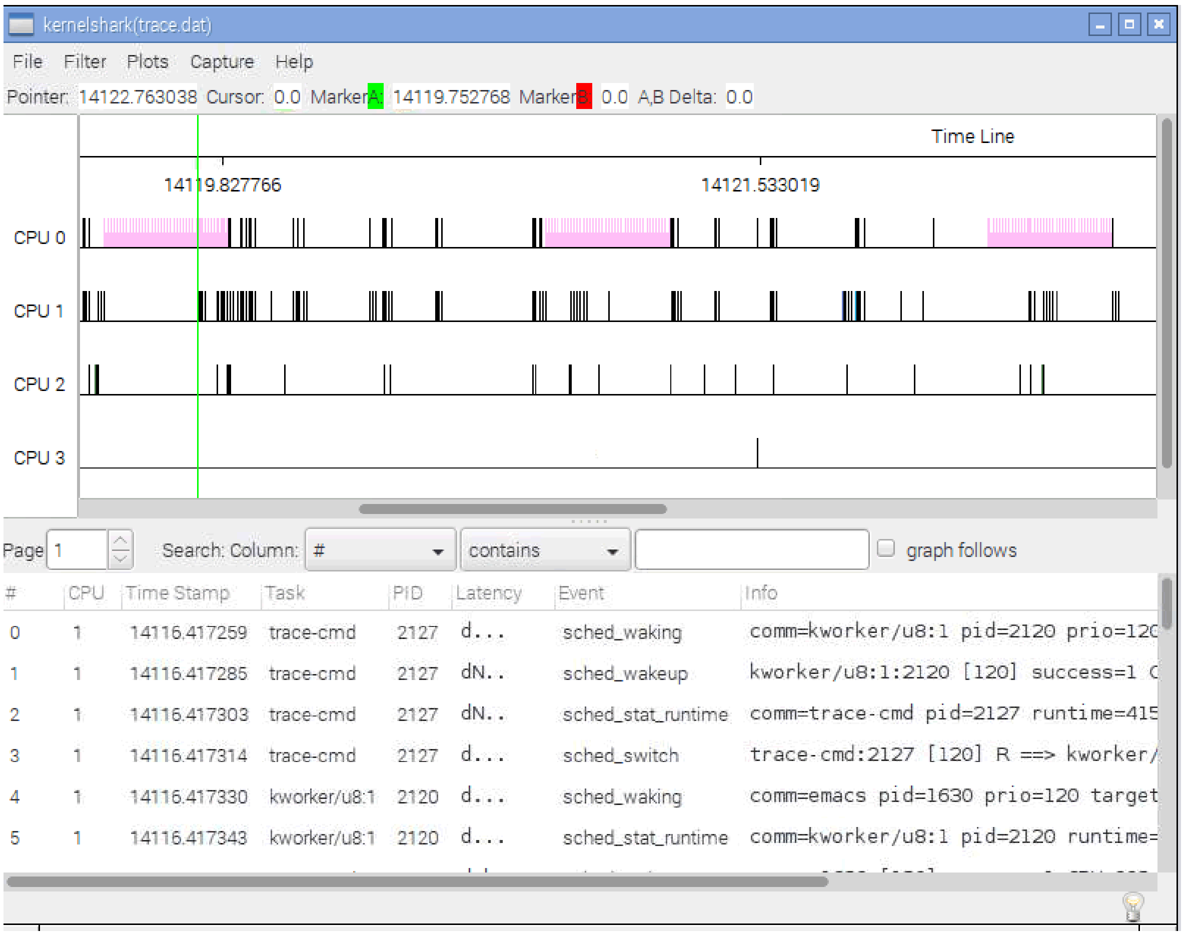
Concerning the example in Listing 4.25, if the cores had names 0 through 3 depending on their column index, it must be seen that the core 0 (first column) is not doing anything whereas most of the load is located on on cores 2 and 3. It must also seen that the process A0 (hence, perf) had switched from core 2 to core 3 at some point in time.

- **Trace-Cmd trace and visualization via kernelshark:** Trace-cmd [94] is yet another tool that records system trace. Trace-cmd trace is generated into a file `trace.dat` by using the command given in Listing 4.27.

```
    sudo trace-cmd record -e sched
```

Listing 4.27: Recording a system trace using trace-cmd

The trace that is generated from *trace-cmd tool* could be observed via a visualization tool called *kernelshark*. By simplying calling `kernelshark` from the directory that `trace.dat` is located, one can launch kernelshark to observe the system trace. An example of kernelshark is shown in the Figure 4.7. Although kernelshark along with



```
Pointer: 14122.763038  Cursor: 0.0  MarkerA: 14119.752768  MarkerB: 0.0  A,B Delta: 0.0

Page 1    Search: Column: #    contains
```

#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	1	14116.417259	trace-cmd	2127	d...	sched_waking	comm=kworker/u8:1 pid=2120 prio=12C
1	1	14116.417285	trace-cmd	2127	dN..	sched_wakeup	kworker/u8:1:2120 [120] success=1 C
2	1	14116.417303	trace-cmd	2127	dN..	sched_stat_runtime	comm=trace-cmd pid=2127 runtime=415
3	1	14116.417314	trace-cmd	2127	d...	sched_switch	trace-cmd:2127 [120] R ==> kworker/
4	1	14116.417330	kworker/u8:1	2120	d...	sched_waking	comm=emacs pid=1630 prio=120 target
5	1	14116.417343	kworker/u8:1	2120	d...	sched_stat_runtime	comm=kworker/u8:1 pid=2120 runtime=

Figure 4.7.: Kernelshark running on Linux (Raspbian) OS

trace-cmd are useful tools that show a CPU graph along with processes, the visualization does not provide information regarding CPU utilization details and communication. For that purpose, TraceCompass tool will be used which will explained in the following paragraph.

- **Perf and Babeltrace CTF tracing and visualization via TraceCompass:** Eclipse TraceCompass [95] is an Eclipse-based open-source platform that is used for providing views, graphs and metrics for many type of logs and traces [95] The graphical user interface of the Eclipse TraceCompass is shown in Figure 4.8. Since the information that is provided by Eclipse TraceCompass is more user-friendly and more detailed than the other visualization options that are discussed, for the A4MCAR the *Eclipse TraceCompass* has been used for investigating the high-level module system trace.

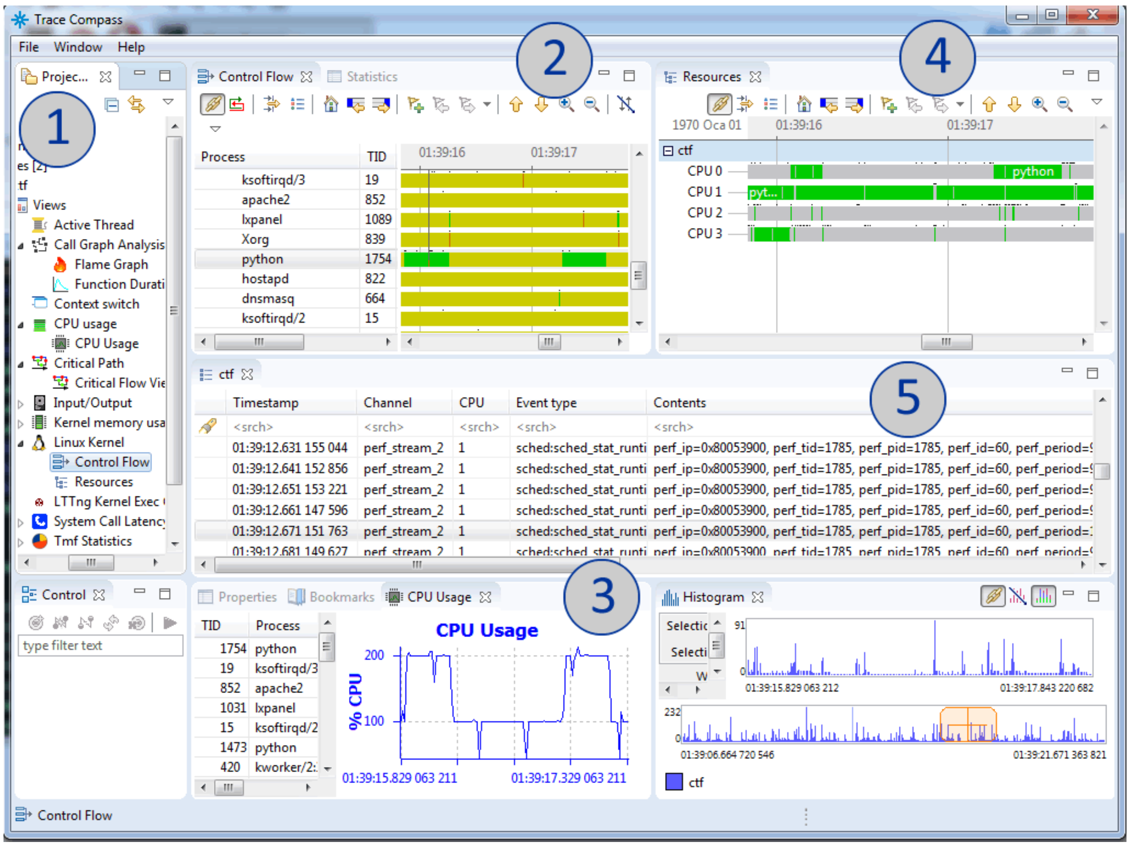
  LTTng [96] is an open-source tracing framework that is provided for the Linux platform which is quite often used with Eclipse TraceCompass due to its format compatibility, i.e. both LTTng and TraceCompass can handle the standard tracing format, *CTF (Common Trace Format)* well. However, since LTTng requires system tracepoints to be designed in the software development stage, in A4MCAR due to its ease of implementation Perf is used for tracing. Although TraceCompass accepts many trace formats, it can not read directly from perf format (`perf.data`). Therefore, the perf trace format should be converted to CTF in order to be analyzed using the Eclipse TraceCompass.

  In order to convert the perf format, one has to build a new version of perf from a Linux kernel module [97] with tracing options enabled and also with a tracing library that is called LibBabelTrace. The detailed information regarding this building process could be seen at [98]. Once the perf is built and installed with babeltrace, one can use the commands in Listing 4.28 to record a trace and then convert it to CTF data format.

```
    sudo perf sched record -e 'sched:*,raw_syscalls:*' -- sleep 15
    sudo LD_LIBRARY_PATH=/opt/libbabeltrace/lib perf data convert --to-ctf=./ctf
```

Listing 4.28: Conversion to Common Trace Format

  Once the trace which is in CTF format is generated, it can be imported into Eclipse TraceCompass. An example trace imported into Eclipse TraceCompass platform is given in the Figure 4.8. Using the figure, the main windows in the Eclipse TraceCompass can be discussed. TraceCompass enables users to look at their system using several views such as call graphs, threads, context switches, cpu usage, critical path, I/O, control flow, and resources which can be seen as (1) in the figure. *Control flow window* (2) shows each process state with respect to time including the transitions along all the processes. System-wide CPU usage and individual processes' CPU usages are shown in the *CPU Usage window* which is shown as (3) in the figure. Therefore, if the system has 4 cores, the CPU usage of up to 400 percent could be observed. *Resources window* (shown as (4)) depicts how processes are distributed amongst the existing cores with respect to time. Therefore, the load balancing could be roughly observed from this view by simply looking at each of the cores. Moreover, using the *Resources window*, one can measure and estimate the timing properties of the schedule of the system. Finally, the trace event list (shown as (5)) can be used to see exact events that occurred in a specific time by selecting a time frame from other windows.



Process	TID
ksoftirqd/3	19
apache2	852
lxpanel	1089
Xorg	839
python	1754
hostapd	822
dnsmasq	664
ksoftirqd/2	15

Timestamp	Channel	CPU	Event type	Contents
<srch>	<srch>	<srch>	<srch>	<srch>
01:39:12.631 155 044	perf_stream_2	1	sched:sched_stat_runti	perf_ip=0x80053900, perf_tid=1785, perf_pid=1785, perf_id=60, perf_period=!
01:39:12.641 152 856	perf_stream_2	1	sched:sched_stat_runti	perf_ip=0x80053900, perf_tid=1785, perf_pid=1785, perf_id=60, perf_period=!
01:39:12.651 153 221	perf_stream_2	1	sched:sched_stat_runti	perf_ip=0x80053900, perf_tid=1785, perf_pid=1785, perf_id=60, perf_period=!
01:39:12.661 147 596	perf_stream_2	1	sched:sched_stat_runti	perf_ip=0x80053900, perf_tid=1785, perf_pid=1785, perf_id=60, perf_period=!
01:39:12.671 151 763	perf_stream_2	1	sched:sched_stat_runti	perf_ip=0x80053900, perf_tid=1785, perf_pid=1785, perf_id=60, perf_period=:
01:39:12.681 149 627	perf_stream_2	1	sched:sched_stat_runti	perf_ip=0x80053900, perf_tid=1785, perf_pid=1785, perf_id=60, perf_period=!

TID	Process
1754	python
19	ksoftirqd/3
852	apache2
1031	lxpanel
15	ksoftirqd/2
1473	python
420	kworker/2:

Figure 4.8.: Eclipse TraceCompass running on Windows

To ease the process of having to trace using perf, convert the trace, and take a look at the process ID list to interpret the trace, a Linux shell (bash) script has been created in order to get necessary outputs from tracing processes and threads automatically. Listing 4.29 shows the content of the script. It is seen in the script that command line arguments are taken (Lines 1 to 4) to make the process more modular. The Lines 11 to 20 are dedicated to executing the commands that are discussed above by making use of the command line arguments. By stating the trace name, tracing period, and perf module installation location, one can use this script to generate traces easier.

```
args=("$@")
trace_name=${args[0]}
seconds=${args[1]}
perf_directory=${args[2]}

if [ "$#" -ne 3 ]; then
   echo "Entered␣arguments␣seem␣to␣be␣incorrect"
   echo "Right␣usage:␣sudo␣TraceLinuxProcesses.sh␣<trace_name>␣<period>␣<path_to_perf>"
   echo "e.g.␣sudo␣TraceLinuxProcesses.sh␣APP4MC_Trace␣15␣/home/pi/linux/tools/perf"
else
   echo "###␣Creating␣directory.."
   sudo mkdir out_$trace_name/
   echo "###␣Writing␣out␣process␣names.."
   ps -aux >> out_$trace_name/Processes_List.txt
   echo "###␣Tracing␣with␣perf␣for␣$seconds␣seconds.."
   sudo $perf_directory/./perf sched record -o out_$trace_name/perf.data -- sleep $seconds
   echo "###␣Converting␣to␣data␣to␣CTF␣(Common␣Tracing␣Format).."
   sudo LD_LIBRARY_PATH=/opt/libbabeltrace/lib $perf_directory/./perf data convert -i out_$trace_name/perf.data --to-ctf=./ctf
   sudo tar -czvf out_$trace_name/trace.tar.gz ctf/
   sudo rm -rf ctf/

   echo "###␣Process␣IDs␣are␣written␣to␣out_$trace_name/Processes_List.txt"
   echo "###␣Trace␣in␣Perf␣format␣is␣written␣to␣out_$trace_name/perf.data"
   echo "###␣Trace␣in␣CTF␣format␣is␣written␣to␣out_$trace_name/trace.tar.gz"
   echo "###␣Exiting.."
fi
```

Listing 4.29: Script to generate traces automatically

At this point, it is really important for a developer to let the Linux Kernel and by extension the TraceCompass identify the processes and threads. The way this is achieved is by manipulating the name of the processes and threads that are visible to Linux kernel, which is known as `command`. In this work, this is researched for both C++ (POSIX threads) and Python (threading) processes and threads.

For POSIX-thread based C/C++ programs, `command` is the executable name for a process when executed as `./cppprogram`. However to set the `command` for the threads of a POSIX-thread based program, `pthread_setname_np` function should be used.

For Python's threading-based programs, both process and thread names should be made visible by specifying `command`, or either the TraceCompass will recognize the processes and threads as just `python`. To set the `command` for a Python executable the first line of the Python program should be set to `#!/usr/bin/python` as opposed to `#!/usr/bin/env python` to make the script executable. After that, if the process is executed using its name `./pyprogram`, the TraceCompass will recognize the process name. Moreover for threads inside a Python program, *prctl* module can be used to set the `command`. The function `prctl.set_name` is useful in this regard.

With all processes and threads named and traced, TraceCompass will visualize the scheduling as shown in Figure 4.9.

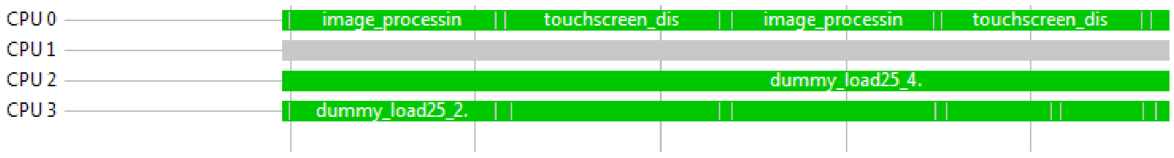


Figure 4.9.: TraceCompass visualization of processes and threads

4.3.2. Process and Thread Mapping

After software evaluation, processes and threads should be pinned to cores properly. To place the processes and threads to cores, the *taskset* module of the Linux platform is used. Taskset module [99] is used to set or retrieve the CPU affinity of a running process given its process ID. CPU affinity is a scheduler property that pins a process to a given set of CPUs on the system. Therefore, the process will not run on any other CPUs after an affinity is set to that process [99]. As described in [99], the Linux scheduler also supports natural CPU affinity: the scheduler attempts to migrate processes along different CPUs as long as it is practical for performance reasons [99]. Therefore, in A4MCAR, by forcing a specific CPU affinity, we investigate if a better distribution could be accomplished than the scheduling automatically done by Linux. To place a process to a core given its process ID, the command given in Listing 4.30 is used.

```
#Place the process on a specific core.
sudo taskset -pc <coreaffinity> <pid>
```

Listing 4.30: Using taskset

As an example, for a 4-core system such as Raspberry Pi 3, core affinity can be values such as 0, 1, 2, 3, 0-1, 0-2, 0-3, 2-3. This example shows that core affinity not necessarily has to be selected as only one core and it can be selected as a range of cores for a process to be distributed. The Listing 4.31 depicts how a process is pinned to a core using its name.

```bash
#!/bin/bash
args=("$@")
process_name=${args[0]}
core=${args[1]} #Affinity, 0-3 for raspberry pi, could be a range too.
pid=$(pgrep -f $process_name -n ) #Newest result
sudo taskset -pc $core $pid && #Place the task on a specific core.
echo "Process␣$process_name␣with␣PID=$pid␣has␣been␣placed␣on␣core␣$core"
```

Listing 4.31: CorePlacer.sh script to pin a process to a core using its name

To manage the distribution process for every process might be time consuming. In order to overcome this issue, a file format has been designed which is then read by the main processing task (Touchscreen display process). The touchscreen process, when the distribution is selected, reads from this file format `coredef_list.a4p` and makes core placements accordingly. An example `coredef_list.a4p` is shown in Listing 4.32. It is shown in the listing that task names and cores are listed by each line.

```
[COREDEF_LIST_APP4MC]
Assign Task Xtightvnc To Core 0
Assign Task mjpg_streamer To Core 0
Assign Task touchscreen_display To Core 0
Assign Task ethernet_client To Core 0
Assign Task core_recorder To Core 0
Assign Task dummy_load25_1 To Core 1
Assign Task dummy_load25_2 To Core 2
Assign Task dummy_load25_3 To Core 1
Assign Task dummy_load25_4 To Core 2
Assign Task dummy_load25_5 To Core 2
Assign Task dummy_load100 To Core 3
Assign Task apache2 To Core 1
Assign Thread Thread_UpdateCoreUsageInfo To Core 2
Assign Thread Thread_TimingCalculation To Core 3
Assign Thread Thread_TouchscreenEvents To Core 0
```

Listing 4.32: File format that contains overall process pinning information

Reading the file and making the changes required is achieved by using the function given with Listing 4.33. In this listing, the file is opened (Line 7), each line is parsed (Lines 8 through 14), then the allocation is done by searching for the item in the global `aprocess_list` (Line 15 through 24) and executing a taskset command with arguments as name of the process and core if the item is found in the process and thread list `aprocess_list`. Considering Figure 3.25, one can better understand how this procedure is interfaced amongst other processes in the high-level module. One should also notice that in the line 25, core affinities of processes are updated.

```
def APP4MCDistributionActions():
   global aprocess_list
   global aprocess_list_len
   process_names = []
   process_affinities = []
   try:
      with open('../../logs/core_mapping/coredef_list.a4p','rb') as coredef_list:
         for line in coredef_list:
            words = line.strip('\n').split('␣')
            if (len(words)>3):
               process_names.append(words[2].strip('\n'))
               process_affinities.append(words[5].strip('\n'))
   except Exception as inst:
      print inst
   lock_aprocess_list.acquire()
   for i in range(0, aprocess_list_len):
      for k in range(0, len(process_names)):
         if (aprocess_list[i].apname == process_names[k] and aprocess_list[i].aprunning == 1):
            if (aprocess_list[i].apid != "NaN" and aprocess_list[i].apid != 0):
               try:
                  os.system("sudo␣taskset␣-pc␣"+str(process_affinities[k])+"␣"+str(
                      aprocess_list[i].apid))
               except Exception as inst:
                  print inst
   lock_aprocess_list.release()
   UpdateCoreAffinityOfProcesses()
```

Listing 4.33: Reading `coredef_list.a4p` and pinning tasks with Python

4.3.3. Investigating Energy Consumption Features

One of the biggest advantages of achieving a better core utilization is to invoke energy saving features of processors by running CPU at lower clock speeds and lower voltages. As mentioned before, computers that run on Linux platform provide a feature that is called DVFS (Dynamic Voltage and Frequency Scaling) [87] which is a processor's ability to dynamically scale its frequency and operating voltage depending on the load. DVFS can affect hardware peripheral chips apart from the processor. Figure 4.10 [86] depicts how DVFS can improve the energy consumption in a system.

For equation 4.1, considering a fixed chip operating voltage V_{DD}, the power is directly proportional to the CPU operation frequency. For example, a processor that is running on 500MHz will draw less current than the same running on 200MHz. However, using a lesser frequency is not the only way to get a lesser power consumption. One can also reduce the operating voltage V_{DD} of a chip to reduce power, provided that the V_{DD} is greater or equal

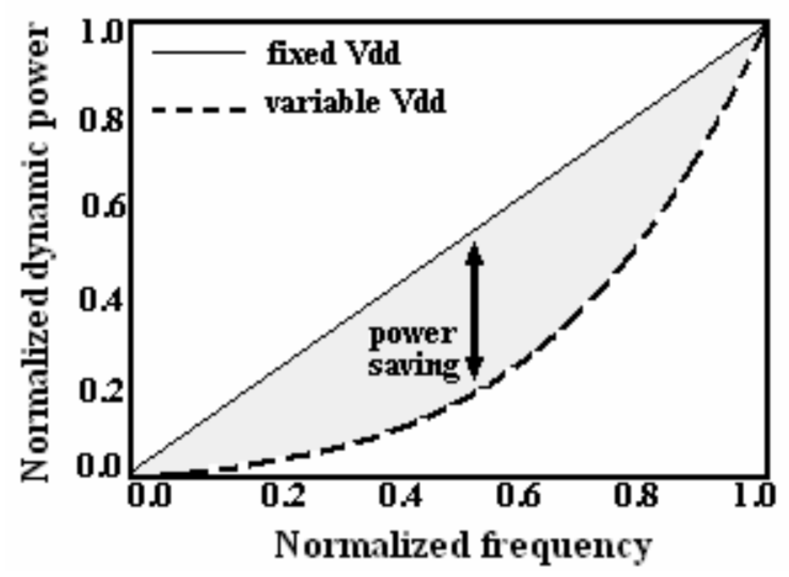


Figure 4.10.: How DVFS reduces energy consumption explained [86]

to the minimum working voltage of that particular chip [86]. It should be noted that that clock frequency should also be reduced in a system with a reduced operating voltage V_{DD} for the system to function correctly [100]. The Figure 4.10 depicts that by decreasing frequency and operating voltage at the same time (DVFS), power consumption is reduced a lot more than it is in the frequency scaling.

In Linux, *CPUFreq* [101] is provided as a module that is responsible for handling dynamic frequency scaling. CPUFreq governors which are responsible for deciding, what frequency should be used in a system, manipulate the CPUFreq driver to switch the policy of the CPU depending on the system load [101]. In A4MCAR, CPUFreq governor of the high-level module is changed in order to achieve less power consumption. Raspberry Pi 3 supports two frequencies that can be used within CPUFreq governors: 600MHz and 1.2GHz. The available governors and their functions are listed below [102]:

- **performance** - sets the frequency statically to the highest available CPU frequency (in Raspberry Pi 3, this is 1.2GHz)
- **powersave** - sets the frequency statically to the lowest available CPU frequency (in Raspberry Pi 3, this is 600MHz)
- **userspace** - set the frequency from a userspace program. A userspace program can determine customized policies and frequencies to be used. For detailed information on userspace governor, [102] can be read.
- **ondemand** - adjust based on utilization

- **conservative** - adjust based on utilization but be a bit more conservative by adjusting gradually

The CPUFreq governor of a Linux system can be changed at any given moment by using the `cpufreq-utils` command. The Linux shell commands in Listing 4.34 show installation of `cpufreq-utils` (Line 1), listing information (Line 2), and current governor selection (Line 3), respectively.

```
1  sudo apt-get install cpufrequtils
2  cpufreq-info
3  cpufreq-set -g <governor> #<governor> could be either of the governors that are listed.
```

Listing 4.34: Changing CPU governer from Linux shell

4.3.4. Online Timing Analysis Features for the A4MCAR

As explained in the Section 3.3.2 in detail, every created application in A4MCAR's high-level module are built on a template that is able to log timing information. The timing logs are read in the Touchscreen display process, which is referred as the main processing task in the high level module. The role of main processing task in the online timing analysis is depicted in the Figure 4.11. The main processing task is responsible to calculate performance indicators such as average slack time ST_{avg} and overall deadline misses percentage DLM using the timing values from other processes in seconds such as execution time ET, slack time ST, deadline DL and period PER (Recall from Section 2.7). The ain processing task can also read from core usage logs and inform users about the low-level module core utilization percentages $LU_{\text{0-15}}$ and high-level module core utilization percentages $HU_{\text{0-3}}$. Furthermore, the number of active cores N_{cores}, number of active and traceable processes N_{process}, number of missed deadlines N_{missed} and clock frequency f_{CPU} are also shown by reading the respective logs.

Subsection 3.3.2 explains the calculation of ET, ST, and determination of DL and PER. Users are able to observe ST_{avg} and DLM on the touchscreen display as seen in Figure 3.39 (g). The calculation of the information that is presented to the user ST_{avg} and DLM are given as follows:

- ST_{avg} in seconds is calculated simply by using the following equation:

$$ST_{\text{avg}} = \frac{1}{N_{\text{process}}} \sum_{n=0}^{N_{\text{process}}} ST_{\text{n}} \tag{4.2}$$

- To find DLM, first, ET and DL of each process are compared. If ET is larger than DL, that process is said to have missed its deadline. A deadline flag DF is defined

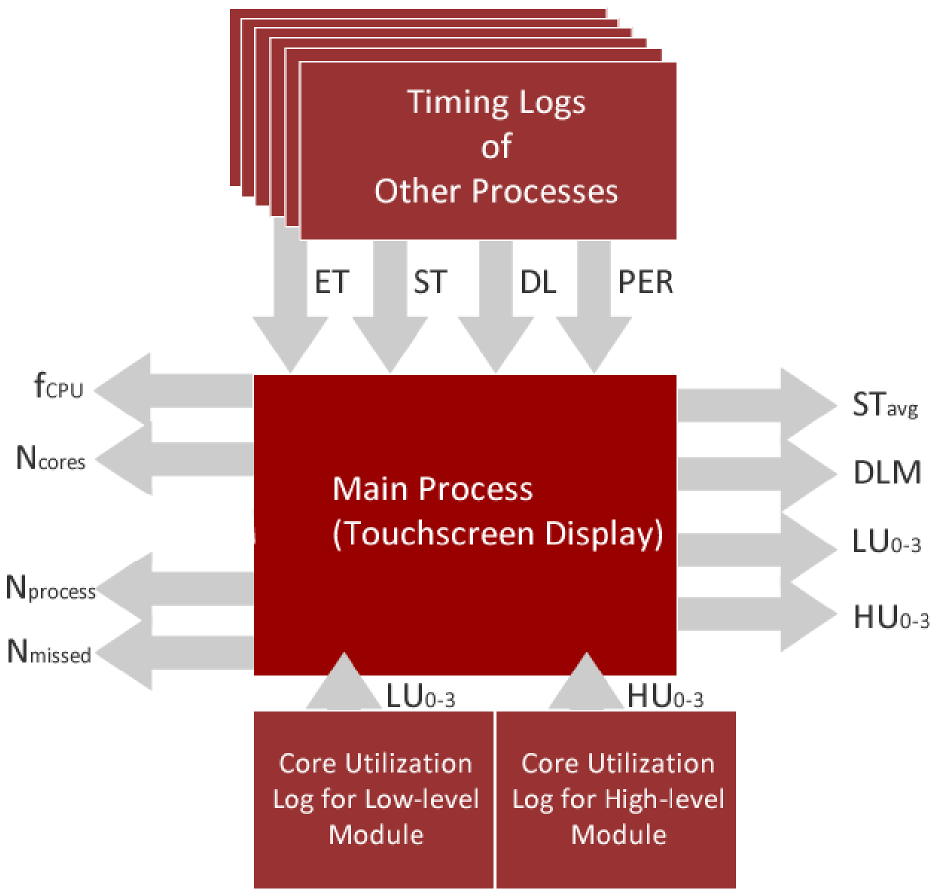

Timing Logs of Other Processes
ET
ST
DL
PER
fCPU
Ncores
Nprocess
Nmissed
Main Process (Touchscreen Display)
STavg
DLM
LU0-3
HU0-3
LU0-3
HU0-3
Core Utilization Log for Low-level Module
Core Utilization Log for High-level Module

Figure 4.11.: Online Timing Analysis explained in A4MCAR

which is 1 if the deadline is missed for a process, and 0 if the deadline is not missed. By using the sum of every deadline flag DF, the DLM is calculated as follows:

$$DLM = \frac{100}{N_{\text{process}}} \sum_{n=0}^{N_{\text{process}}} DF_{\text{n}} \tag{4.3}$$

In the following chapter, the evaluation of distributions and results is given and explained.

5. Evaluation and Results

5.1. System Limitations and Factors that Affect the Results

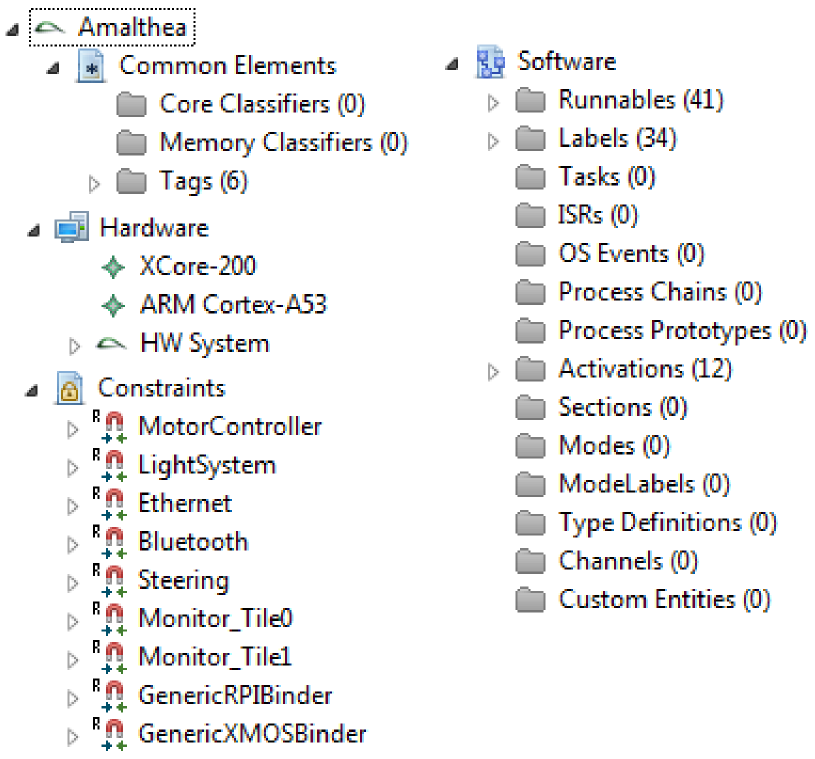
In this chapter, system modeling with APP4MC and evaluations of mapping outcomes are discussed. The A4MCAR, like every embedded system, has its limitations related to both hardware and software. APP4MC' primary objective is serving industry-related applications, supported with industry-related tools with precise tracing and distribution interfaces. However, applying APP4MC solutions for custom open-source and commercial tools requires some effort regarding developing tool support for dealing with the issues such as modeling, tracing, and distribution. Furthermore, overheads and limitations are present for custom open-source platforms and tools. The following list explains the limitations that affect the demonstrative purposes of the project and the results of the software distribution:

- **Model limitations and overheads in the system**: Although the created AMALTHEA model contains most of the information related to runnables, it does not contain information such as OS scheduling, reading/writing to/from files, kernel overheads, and tracing overheads. Furthermore, in order to ease the modeling process, some trivial shared resources and runnables that mainly belong to Linux system's internal operation are not modeled. Therefore, created AMALTHEA model is not 100% precise. This situation will create non-deterministic error in the outcomes that should be noted.
- **Sporadic activations**: Since some tasks especially in the low-level module are activated randomly rather than periodically, the system and the model has non-deterministic behavior to some degree. That is one of the limitations which constrain the output from APP4MC.
- **Limitations of perf the profiler**: The perf profiler uses dynamic analysis to get the granularity of tasks [91]. Due to the profiler overhead, the resulting granularities that are observed with the help of the perf profiler might contain errors. However, it is assumed in this work that granularities of tasks being relative would not produce errors in the partitioning.

- **Limitations of process-based distribution**: Since in this work not very fine-grained processes are distributed, the load balancing by 100% would not be possible. As an example, image processing process contains a significantly higher instruction size than any of the other processes. Therefore, the goal is to achieve the best possible load balancing with the obtained granularity data.
- **Limitations of XTA**: The XTA tool, as mentioned in Section 4.2.1, generates hardware related unresolvable errors when analyzing the granularities for runnables. In this work, for the tasks that the timing analysis resulted in unresolvable errors, the disassembly instructions are counted. This approach disregards loops and might lead to non-accurate granularities.
- **Hardware limitations**: Hardware limitations affect the outcomes when the energy consumption is a concern. Since the default DVFS option for Raspberry Pi 3 only allows underclocking to 600MHz clock frequency and other frequencies are not supported, the default underclocking is investigated to obtain reduced energy consumption.
- **XMOS Multicore Design Rules and Third-party library conflicts:** In the low-level module, which uses the XMOS xCORE-200 eXplorerKIT multi-core development board as its basis, some tasks could not distributed effectively along individual cores of the system. This is because of the conflict between XMOS multi-core design rules and the used third party libraries. According to XMOS multi-core design rules which are explained in [40], only a combinable function can be distributed to a single core. However, due to the fact that core implementations of some library functions do not have a combinable nature, those functions were not able to be distributed. Furthermore, the library functions which use multiple cores were not also able to be distributed manually to cores. Although these changes are reflected to the model, due to being unable to distribute some tasks, the output from APP4MC does not present an optimal solution regarding load balancing and reduced energy consumption.
- **Deficiency in Core Utilization Tracing in XMOS:** The provided register information is not sufficient in reading core utilization information for sporadic tasks. Only periodic tasks are correctly measured with the introduced core reading methodology. For that reason, the utilization is not visualized very efficiently.

5.2. Modeling the A4MCAR using APP4MC

In Section 2.10, motivations and design techniques of APP4MC were introduced. The users have to start designing the parallel application by making use of the modeling functionality of APP4MC. In order to evaluate how APP4MC performs in regard to the A4MCAR, the design starts by identifying several attributes that are related to software and hardware. With the help of the modeling functionality of APP4MC, A4MCAR's hardware details, software details, constraints, and common elements (such as tags) are described. It is important to mention that APP4MC can be used to describe several types of models such as Components Model, OS Model, Mapping Model, Stimuli Model, and Event Model. However, the model in this work explains the minimalistic model that can be used to make use of APP4MC's partitioning and mapping features. The models contain XML-based hierarchy with elements having their childs and attributes. With this hierarchy, the containments of the elements are described easily. The following list briefly explains the initially created model, which is also shown in the *AMALTHEA Contents tree window* given by the Figure 5.1:



- Amalthea
  - Common Elements
    - Core Classifiers (0)
    - Memory Classifiers (0)
    - Tags (6)
  - Hardware
    - XCore-200
    - ARM Cortex-A53
    - HW System
  - Constraints
    - MotorController
    - LightSystem
    - Ethernet
    - Bluetooth
    - Steering
    - Monitor_Tile0
    - Monitor_Tile1
    - GenericRPIBinder
    - GenericXMOSBinder
  - Software
    - Runnables (41)
    - Labels (34)
    - Tasks (0)
    - ISRs (0)
    - OS Events (0)
    - Process Chains (0)
    - Process Prototypes (0)
    - Activations (12)
    - Sections (0)
    - Modes (0)
    - ModeLabels (0)
    - Type Definitions (0)
    - Channels (0)
    - Custom Entities (0)

Figure 5.1.: AMALTHEA Contents tree window for the created model for A4MCAR

- **Hardware Model:** The hardware model consists of two processor types: *xCORE-200* for the low-level module (XMOS board), and *ARM Cortex-A53* for the high-level module (RPI3 board). The *HW System* element contains ECUs (*XMOS* and *RPI3*) with each of the ECUs having their respective microcontrollers defined. For the A4MCAR, the *XMOS* ECU has two tiles defined as microcontrollers (*Tile0* and *Tile1*) , whereas the *RPI3* ECU has only one microcontroller element which is *CortexA53*. Each microcontroller element has their respective clocks and individual cores defined under them. That is, *Tile0* has 8 cores, *Tile1* has 8 cores, and *CortexA53* has 4 cores. Furthermore, the default clock setup (when energy consumption is not a concern) requires *Tile0* and *Tile1* to have 500MHz system frequency in the model whereas *CortexA53* is defined as 1.2GHz. The defined hardware model is used in APP4MC for the mapping of software processes to the hardware cores.

- **Software Model:** The software model defines the pairwise relationship between runnables, activation conditions of runnables, labels (memory read and write accesses), events and interrupts. A minimalistic software model in APP4MC should have runnables, labels and activations defined which are shown in the Figure 5.1.

  In chapter 4, the information tracing of multi-tasked systems is discussed. By making use of the aforementioned techniques, runnables, labels, and activations should be defined. Even in a distributed architecture such as the A4MCAR, all the runnables from each ECU are listed under the *Runnables* element. Runnables usually are the smallest execution units of a task. However, for the A4MCAR some processes, events, and tasks are modeled as runnables due to the ease of modeling. For the A4MCAR, an initial analysis led up to a model with 41 runnables. Each runnable listed under *Runnables* element has its own granularity (i.e number of instructions) defined. Furthermore, labels are used to define shared variables and inter-process communication. Labels are defined e.g. in bit size and memory size and each runnable is configured depending on their read and write accesses to labels. One should take a look at the Figures 3.19 and 3.25 in order to see label accesses that are modeled in A4MCAR's low-level and high-level module. Finally, activations of each runnable are listed. Commonly, activations can either be periodic or sporadic (random).

  After the minimalistic software model is ready, performing partitioning and tracing features will improve the existing model. As an example, *Process Prototypes* will be automatically generated after the partitioning.

- **Constraints Model:** The contraints model commonly is used for defining target core dependencies and pairings for the generated partitions. In A4MCAR's model, runnables that functionally belong together are paired using constraints model (by using *Runnable Pairing Constraint*). As an example, in Figure 5.1 it is shown that tasks related to e.g. *Bluetooth*, *Ethernet* and *Steering* are bound together. Thus, the created partitions and tasks would consider this runnable binding. Furthermore, target core requirements for *Monitor_Tile0* task and *Monitor_Tile1* task are defined using the constraints model, since *Monitor_Tile0* task should be on one of the *Tile0* cores and *Monitor_Tile1* task should be on one of the *Tile1* cores. Finally, generic core specifications are defined (*GenericRPIBinder* and *GenericXMOSBinder*) . Since the A4MCAR has a distributed architecture, it is important to make sure that low-level module tasks are mapped to low-level module cores and high-level module tasks are mapped to high-level cores by specifying those binders. As a general note, the information that is specified in the constraints model is used in the pre-partitioning, partitioning, and mapping phases.

- **Common Elements:** In A4MCAR, tags are used from the common elements model. Tags define binders for runnables that are considered in the pre-partitioning phase. Thus, the pre-partitioning phase would generate partitions that have the same tags. In A4MCAR, by making use of two tags which are *LLM_Task* and *HLM_Task*, it is made sure that generated partitions and tasks will not have runnables that run on the other module. This means that low-level module runnables and high-level module runnables are isolated.

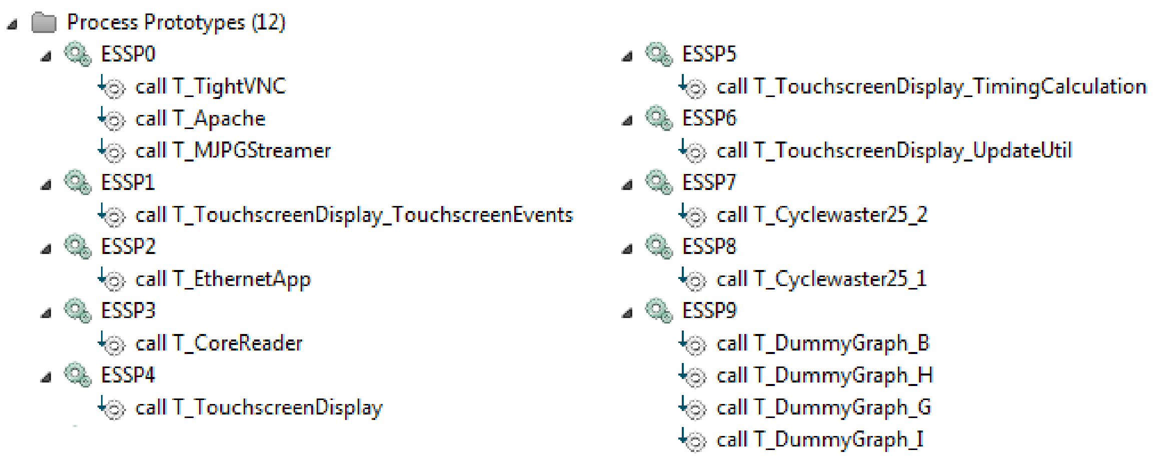
Although the previously explained model includes both low-level module and high-level module components, to generate partitioning and mapping outputs, the model is seperated to two AMALTHEA models, each of which containing the elements for respective modules (high-level and low-level). The reason to use this approach was the lack of the implementation of tag-based (in our case with respect to modules) partition grouping in the version of APP4MC (0.8.1) that was used.

5.3. Partitioning and Mapping

One important thing to consider when partitioning the runnables, especially in the Critical Path Partitioning, is that when label accesses are strictly modeled, the partitioning output might not be ideal for parallelization towards load balancing. The reason is that APP4MC, in the partitioning stage puts all the runnables that belong to the critical path to one single partition. Our experiments showed that A4MCAR's real-world application encountered to this problem that results in poor load balancing initially due to the fact that task dependencies are strictly considered in APP4MC in contrast with OS-based scheduling. To solve this issue, two solutions are considered: (1) - Removing non-critical label accesses, (2) - Defining non-critical label accesses as *Access Precedence* to prevent APP4MC from considering label accesses strictly in model, and do the partitioning towards load-balancing.

After the modeling, partitioning (pre-partitioning is performed automatically before partitioning), task generation, and mapping are performed by using APP4MC to obtain the distribution results. Obtained results will be shown in the following sections.

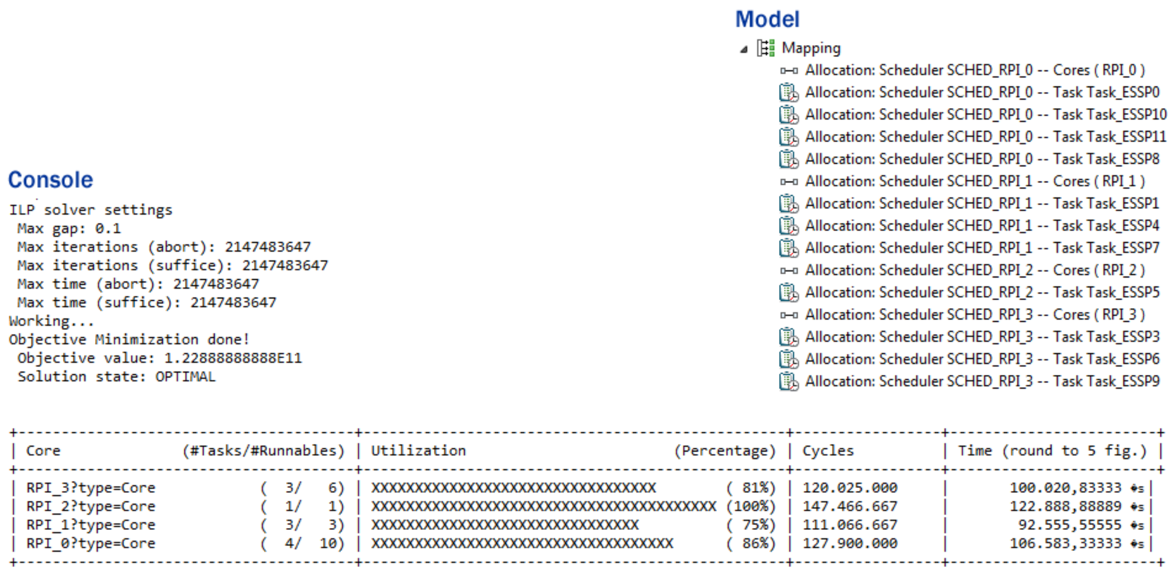
For the sake of completeness, APP4MC's partitioning and mapping outcomes are briefly discussed. After the model is complete, by using APP4MC multicore sections drop-down menu, one can perform several operations. After the partitioning is complete on the initally created model, a new model that includes the process prototypes is generated in the `output` folder in the APP4MC project. Under *Process Prototypes*, partitions can be seen which shows all the runnables that are in a partition. The Figure 5.2 depicts an example partitioning outcome.



```
Process Prototypes (12)
  ESSP0
      call T_TightVNC
      call T_Apache
      call T_MJPGStreamer
  ESSP1
      call T_TouchscreenDisplay_TouchscreenEvents
  ESSP2
      call T_EthernetApp
  ESSP3
      call T_CoreReader
  ESSP4
      call T_TouchscreenDisplay
  ESSP5
      call T_TouchscreenDisplay_TimingCalculation
  ESSP6
      call T_TouchscreenDisplay_UpdateUtil
  ESSP7
      call T_Cyclewaster25_2
  ESSP8
      call T_Cyclewaster25_1
  ESSP9
      call T_DummyGraph_B
      call T_DummyGraph_H
      call T_DummyGraph_G
      call T_DummyGraph_I
```

Figure 5.2.: Example partitioning output from APP4MC

After the partitioning, one should generate tasks using the same drop-down menu. On the created new model, one can perform the mapping process. It is important to know that all the steps such as partitioning, task generation, and mapping can be handled by using the *MC Wizard* tool of APP4MC. The mapping generates the utilization information shown at the console for the given model as well as the mapping output on the model as shown in Figure 5.3. In the figure, allocation of created partitions to the cores is illustrated. If desired, using these mapping outcomes and using a custom scheduler, the simulation of the schedule of tasks on cores are also possible by using the APP4MC's *Visualize Task Execution* feature. Since in this work, real traces are used for visualizing the task execution, this feature is not used.



Model

```
Mapping
  Allocation: Scheduler SCHED_RPI_0 -- Cores ( RPI_0 )
  Allocation: Scheduler SCHED_RPI_0 -- Task Task_ESSP0
  Allocation: Scheduler SCHED_RPI_0 -- Task Task_ESSP10
  Allocation: Scheduler SCHED_RPI_0 -- Task Task_ESSP11
  Allocation: Scheduler SCHED_RPI_0 -- Task Task_ESSP8
  Allocation: Scheduler SCHED_RPI_1 -- Cores ( RPI_1 )
  Allocation: Scheduler SCHED_RPI_1 -- Task Task_ESSP1
  Allocation: Scheduler SCHED_RPI_1 -- Task Task_ESSP4
  Allocation: Scheduler SCHED_RPI_1 -- Task Task_ESSP7
  Allocation: Scheduler SCHED_RPI_2 -- Cores ( RPI_2 )
  Allocation: Scheduler SCHED_RPI_2 -- Task Task_ESSP5
  Allocation: Scheduler SCHED_RPI_3 -- Cores ( RPI_3 )
  Allocation: Scheduler SCHED_RPI_3 -- Task Task_ESSP3
  Allocation: Scheduler SCHED_RPI_3 -- Task Task_ESSP6
  Allocation: Scheduler SCHED_RPI_3 -- Task Task_ESSP9
```

Console

```
ILP solver settings
 Max gap: 0.1
 Max iterations (abort): 2147483647
 Max iterations (suffice): 2147483647
 Max time (abort): 2147483647
 Max time (suffice): 2147483647
Working...
Objective Minimization done!
 Objective value: 1.22888888888E11
 Solution state: OPTIMAL
```

```
+-------------------------------------+-------------------------------------------------+---------------+------------------------+
| Core             (#Tasks/#Runnables) | Utilization                       (Percentage) | Cycles        | Time (round to 5 fig.) |
+-------------------------------------+-------------------------------------------------+---------------+------------------------+
| RPI_3?type=Core          (  3/   6) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX        ( 81%) | 120.025.000   |     100.020,83333 ♦s|
| RPI_2?type=Core          (  1/   1) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX (100%) | 147.466.667   |     122.888,88889 ♦s|
| RPI_1?type=Core          (  3/   3) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX          ( 75%) | 111.066.667   |      92.555,55555 ♦s|
| RPI_0?type=Core          (  4/  10) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX      ( 86%) | 127.900.000   |     106.583,33333 ♦s|
+-------------------------------------+-------------------------------------------------+---------------+------------------------+
```

Figure 5.3.: Example mapping outputs from APP4MC (ESSP partitioning, 10 partitions)

In the APP4MC's mapping stage, GA-based mapping algorithm towards load balancing goal has been used, due to a minor bug at the ILP-based algorithm at the time. The ILP-based mapping algorithm of APP4MC features a 1-step algorithm that is used toward the goal of minimizing the total computation time. After this goal is achieved, remaining processes are more or less distributed randomly. In the case of A4MCAR's applications, GA-based algorithm and ILP-based algorithm only had very minor utilization differences. However, Lukas Krawczyk et al. [103] states that with a 2-step or n-step algorithm, the load on cores could be balanced more optimally. APP4MC's mapping results present a suggestive mapping outcome to the user. User could decide to use lesser number of cores or under-clock the cores to reduce power consumption. Further evaluations in this report shows the results of mapping using GA-based load balancing algorithm.

5.4. Evaluation of Different Distributions

Several evaluation metrics are used for evaluating different distributions, using the results from sequential, OS-based, and APP4MC-based software deployment outcomes. These metrics are presented previously and involve average slack time, gross execution time, computation time, utilization percentages in cores, deadline misses, current utilization used by the processor (in relation to power consumption). Furthermore, tracing results are be visualised using Eclipse TraceCompass for key distributions in the following subsections.

5.4.1. Evaluation of the High Level Module Distributions

In this section, APP4MC outcomes are presented and compared to Sequential and GNU/Linux OS outcomes. For the sake of clarity, this section introduces all the implemented and modeled runnables with their instruction sizes (obtained using `perf stat` tool) and activations in the Table 5.1. Because of the aforementioned limitations (causal order is not necessary in many applications, and OS does not ensure causal order), only the dependency graph for the Dummy Graph process is modeled, which is shown in the Figure 3.33, to demonstrate partitioning outcomes.

Later in this section for specific distributions, some threads and processes will be picked and partitioning and mapping outcomes will be discussed for those process and thread groups. Furthermore, the visualizations of the system traces are presented to depict how distribution briefly looks and to what extent the CPU is utilized.

Three mapping results are compared for every high-level distribution at the end of this section: OS distribution in which core affinities are not constrained (that is, processes and threads are not bound on a single core, they can be moved between the cores 0-3 at any time by the kernel when scheduling), sequential mapping where core affinities are forced to only one single core, and APP4MC mapping where core affinity results are obtained by making use of APP4MC's partitioning and mapping results.

To have flexibility in the mapping stage, the partitioning feature of APP4MC is configured to create 10 partitions that are created using ESSP (Earliest Start Schedule Partitioning). In APP4MC's ESSP algorithm, runnables that have the same activation period or activation type are placed to seperate partitions if not configured otherwise. With partitions that have same activation period, runnables are placed to partitions to reach load balancing. For the mapping stage, for all distributions GA-based load balancing technique is chosen from APP4MC which is currently optimized towards reducing the overall computation time.

5.4.1.1. APP4MC Results for the Distribution HL_Distr_wStream

The initial distribution for the high-level module (`HL_Distr_wStream`) involves the processes and threads that actually contribute to the functionality of the A4MCAR. In other words, the A4MCAR is not stressed by using any additional dummy load processes. Since both the image processing process and the camera stream use the Raspberry Pi camera, this distribution demonstrates the mapping results when the camera is used by the `Camera Stream` process. This means that the `Image Processing` process is not active in this distribution but is active with another distribution.

In the `HL_Distr_wStream` distribution, the following processes are considered active (running, contributes to the processing, and therefore modeled) from the process and thread list given in Table 5.1: `Camera Stream`, `Web Server`, `Core Recorder`, `Ethernet Client`, `VNC Server`, `Touchscreen`, `Dummy Graph`.

Partitioning and mapping of this distribution using APP4MC with no constraints resulted in the partitions that are shown in the Table 5.2.

5.4.1.2. APP4MC Results for the Distribution HL_Distr_wImageProc

The process list in which the `camera stream` is not active but the `image processing` is active is also modeled and evaluated in this distribution. In the distribution `HL_Distr_wImageProc`, the following processes are considered active from the process and thread list given in Table 5.1: `Web Server`, `Core Recorder`, `Ethernet Client`, `VNC Server`, `Touchscreen`, `ImageProcess`, `Dummy Graph`.

Partitioning and mapping of this distribution using APP4MC with no constraints resulted in the partitions that are shown in the Table 5.3.

Process / Thread Name	**Granularity**	**Activation**
Web Server	500000	Sporadic [1s periodic]
Core Recorder	525000	Periodic 3s
Ethernet Client	120000	Periodic 0.01s
VNC Server	10000000	Sporadic [1s periodic]
Camera Stream	1500000	Sporadic [1s periodic]
ImageProcess	450000000	Periodic 0.65s
dummy_load_25_1	198000000	Periodic 1.4s
dummy_load_25_2	198000000	Periodic 1.4s
dummy_load_25_3	198000000	Periodic 1.4s
dummy_load_25_4	198000000	Periodic 1.4s
dummy_load_25_5	198000000	Periodic 1.4s
dummy_load_100	198000000	Periodic 0.5s
Touchscreen	*threads given below*	*threads given below*
MainThread	110000000	Periodic 0.5s
UpdateUtil	150000000	Periodic 2s
TimingCalculation	158000000	Periodic 2.8s
TouchEvents	10000000	Periodic 0.1s
Dummy Graph	*threads given below*	*threads given below*
A	9000000	Periodic 0.5s
B	27000000	Periodic 0.5s
C	36000000	Periodic 0.5s
D	81000000	Periodic 0.5s
E	9000000	Periodic 0.5s
F	18000000	Periodic 0.5s
G	45000000	Periodic 0.5s
H	27000000	Periodic 0.5s
I	18000000	Periodic 0.5s
J	36000000	Periodic 0.5s

Table 5.1.: All processes and threads with their granularity and activation information (with Sporadic activation assumptions shown with square brackets)

5.4.1.3. APP4MC Results for the Distribution HL_Distr_AvgStress

In the distribution `HL_Distr_AvgStress`, the A4MCAR is partially stressed. This is achieved by introducing two dummy loads. In this distribution, the following processes are considered active from the process and thread list given in Table 5.1: `Web Server`, `Core Recorder`, `Ethernet Client`, `VNC Server`, `Dummy Graph`, `Touchscreen`, `Camera Stream`, `dummy_load_25_1`,

Partition Name	Process (Threads) List	Allocated Core
ESSP0	Touchscreen (TouchEvents)	1
ESSP1	Ethernet Client	1
ESSP2	Core Reader	2
ESSP3	Touchscreen (MainThread)	1
ESSP4	Touchscreen (TimingCalculation)	0
ESSP5	Touchscreen (UpdateUtil)	1
ESSP6	Dummy Graph (A, E, C, F, D, J)	2
ESSP7	Dummy Graph (B, H, G, I)	3
ESSP8	Web Server, VNC Server	1
ESSP9	Camera Stream	1

Table 5.2.: Partitioning and mapping results of HL_Distr_wStream using APP4MC

Partition Name	Process (Threads) List	Allocated Core
ESSP0	VNC Server, Web Server	0
ESSP1	Touchscreen(TouchEvents)	0
ESSP2	Ethernet Client	1
ESSP3	Core Reader	1
ESSP4	Touchscreen (MainThread)	0
ESSP5	Touchscreen (TimingCalculation)	3
ESSP6	Touchscreen (UpdateUtil)	1
ESSP7	ImageProcess	2
ESSP8	Dummy Graph (A, E, C, F, D, J)	1
ESSP9	Dummy Graph (B, H, G, I)	0

Table 5.3.: Partitioning and mapping results of HL_Distr_ImageProc using APP4MC

`dummy_load_25_2`.

Partitioning and mapping of this distribution using APP4MC with no constraints resulted in the partitions that are shown in the Table 5.4.

5.4.1.4. APP4MC Results for the Distribution HL_Dist_FullStress

The distribution `HL_Dist_FullStress` represents the distribution in which nearly all the created processes and threads are running. Thus, the A4MCAR's high-level module is in the most stressed state. In this distribution, the following processes are considered active from the process and thread list given in Table 5.1: `Web Server`, `Core Recorder`, `Ethernet`

Partition Name	Process (Threads) List	Allocated Core
ESSP0	VNC Server, Web Server, Camera Stream	2
ESSP1	Touchscreen (TouchEvents)	1
ESSP2	Ethernet Client	1
ESSP3	Core Reader	3
ESSP4	Dummy Graph (all threads)	1
ESSP5	Touchscreen (MainThread)	3
ESSP6	Touchscreen (TimingCalculation)	0
ESSP7	Touchscreen (UpdateUtil)	2
ESSP8	dummy_load_25_2	1
ESSP9	dummy_load_25_1	3

Table 5.4.: Partitioning and mapping results of HL_Distr_AvgStress using APP4MC

`Client`, `VNC Server`, `Dummy Graph`, `Touchscreen`, `Camera Stream`, `dummy_load_25_1`, `dummy_load_25_2`, `dummy_load_25_3`, `dummy_load_25_4`, `dummy_load_25_5`, `dummy_load_100`.

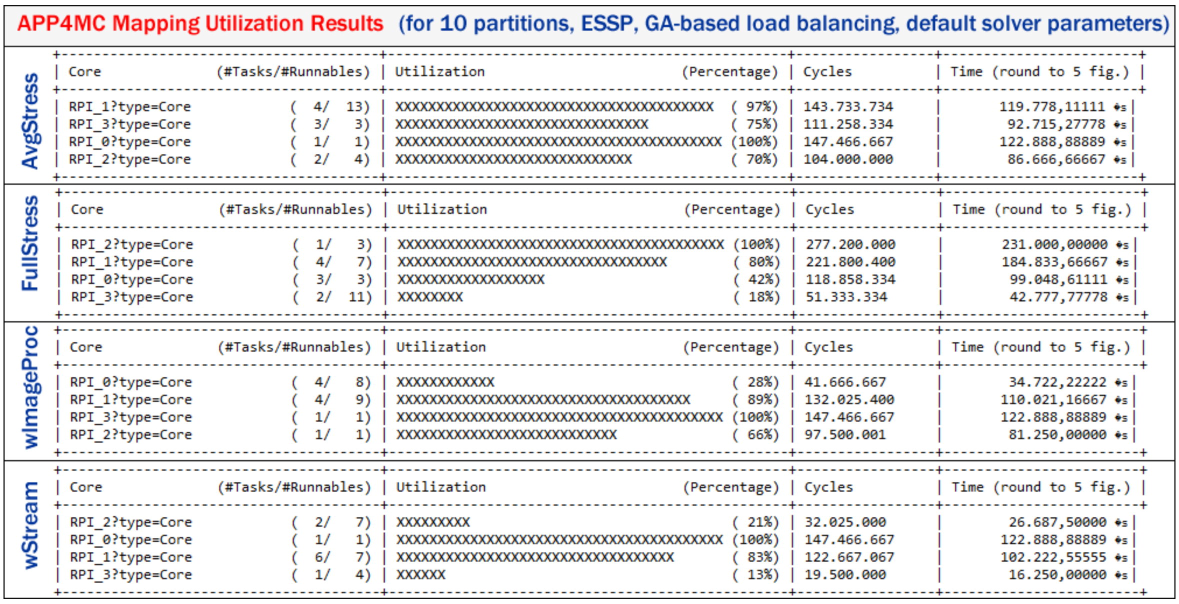
Partitioning and mapping of those processes and threads using APP4MC with no constraints resulted in the partitions that are shown in the Table 5.5.

Partition Name	Process (Threads) List	Allocated Core
ESSP0	VNC Server, Web Server, Camera Stream	1
ESSP1	Touchscreen (TouchEvents)	3
ESSP2	Ethernet Client	1
ESSP3	Core Reader	0
ESSP4	Dummy Graph (all threads)	3
ESSP5	Touchscreen (UpdateUtil)	0
ESSP6	dummy_load_25_5, dummy_load_25_3, dummy_load_25_1	2
ESSP7	dummy_load_25_4, dummy_load_25_2	1
ESSP8	Touchscreen (MainThread)	0
ESSP9	dummy_load_100	1

Table 5.5.: Partitioning and mapping results of HL_Distr_FullStress using APP4MC

5.4.1.5. Comparison of High-level Module Distributions

APP4MC is able to display theoretical load utilization among modeled cores of a system. The experiments done using GA-based mapping approach with 10-partitions partitioned using ESSP algorithm resulted in the utilization given in the Figure 5.4.



APP4MC Mapping Utilization Results (for 10 partitions, ESSP, GA-based load balancing, default solver parameters)

AvgStress

```
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
| Core                (#Tasks/#Runnables) | Utilization                                          (Percentage) | Cycles        | Time (round to 5 fig.) |
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
| RPI_1?type=Core              (  4/  13) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX   ( 97%) | 143.733.734   |        119.778,11111 ♦s |
| RPI_3?type=Core              (  3/   3) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX           ( 75%) | 111.258.334   |         92.715,27778 ♦s |
| RPI_0?type=Core              (  1/   1) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  (100%) | 147.466.667   |        122.888,88889 ♦s |
| RPI_2?type=Core              (  2/   4) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXX             ( 70%) | 104.000.000   |         86.666,66667 ♦s |
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
```

FullStress

```
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
| Core                (#Tasks/#Runnables) | Utilization                                          (Percentage) | Cycles        | Time (round to 5 fig.) |
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
| RPI_2?type=Core              (  1/   3) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  (100%) | 277.200.000   |        231.000,00000 ♦s |
| RPI_1?type=Core              (  4/   7) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX         ( 80%) | 221.800.400   |        184.833,66667 ♦s |
| RPI_0?type=Core              (  3/   3) | XXXXXXXXXXXXXXXXXX                        ( 42%) | 118.858.334   |         99.048,61111 ♦s |
| RPI_3?type=Core              (  2/  11) | XXXXXXXX                                  ( 18%) | 51.333.334    |         42.777,77778 ♦s |
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
```

wImageProc

```
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
| Core                (#Tasks/#Runnables) | Utilization                                          (Percentage) | Cycles        | Time (round to 5 fig.) |
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
| RPI_0?type=Core              (  4/   8) | XXXXXXXXXXXX                              ( 28%) | 41.666.667    |         34.722,22222 ♦s |
| RPI_1?type=Core              (  4/   9) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX      ( 89%) | 132.025.400   |        110.021,16667 ♦s |
| RPI_3?type=Core              (  1/   1) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  (100%) | 147.466.667   |        122.888,88889 ♦s |
| RPI_2?type=Core              (  1/   1) | XXXXXXXXXXXXXXXXXXXXXXXXXXX               ( 66%) | 97.500.001    |         81.250,00000 ♦s |
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
```

wStream

```
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
| Core                (#Tasks/#Runnables) | Utilization                                          (Percentage) | Cycles        | Time (round to 5 fig.) |
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
| RPI_2?type=Core              (  2/   7) | XXXXXXXXX                                 ( 21%) | 32.025.000    |         26.687,50000 ♦s |
| RPI_0?type=Core              (  1/   1) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  (100%) | 147.466.667   |        122.888,88889 ♦s |
| RPI_1?type=Core              (  6/   7) | XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX        ( 83%) | 122.667.067   |        102.222,55555 ♦s |
| RPI_3?type=Core              (  1/   4) | XXXXXX                                    ( 13%) | 19.500.000    |         16.250,00000 ♦s |
+--------------------------------------+--------------------------------------------------------------+---------------+-----------------------+
```

Figure 5.4.: Resulted Mapping Utilizations from Distributions

It must be observed that due to the several limitations mentioned below, the obtained results do not have a complete load balance. Moreover, the utilizations presented in the table are theoretical results with one core running at 100% everytime because the mapping is implemented this way to reduce computation time. A 100% percentage is adapted towards total time for the specific distribution, not toward deadlines and activations. Using a Linux scheduler on top of these results would alter the utilizations obtained in actual case.

- By default, partitioning considers load-balancing, but with only for partitions that have the same type of activation. However, one can deactivate activation grouping in APP4MC. Deactivating activation grouping may result in better load balancing but unfeasible executions. In the case of A4MCAR, activation grouping caused partitions to have improperly load balanced partitions.
- Partitioning gives higher priority to sequences, since considering synchronization between runnables will result in reduced computation time. However, it is important to note that dependencies in sequences are not ensured in OS.
- The mapping approach in APP4MC focuses on reducing overall computation time rather than achieving pure load balance. Reducing the overall computation time is focused on improving deadline misses, whereas load balancing helps with energy efficiency and slack time.
- Modeled runnables are not 100% fine-grained as it is in the theoretical or industrial applications. The reason it is not 100% fine-grained as in the industry is the lack of low-level open-source real-time tools. Since the functionality of the application is complex and open-source tools are used, the amount of overhead is significantly higher than

it is in industrial software systems. This could also lead to load imbalance between cores.

By applying APP4MC-based, OS-based, and Sequential core affinities to the processes and threads for the A4MCAR, Table 5.6 is obtained. In Table 5.6, distributions are evaluated by changed the distribution type (APP4MC, OS, Sequential) and CPU clock speed (f_{clk} = 1.2GHz or 600MHz). Obtained performance indicators involve, total gross execution time (GET), slack time average (ST_{avg}), deadline miss percentage (DLM), utilization in each core ($U_{0\text{-}3}$), and the current drained by the board (I_{DD}) as an indicator for power consumption. When evaluating the high-level distributions, the following assessments should be outlined:

No	Distr.Name	Distr.Type	f_{clk}	GET	ST_{avg}	$U_{0\text{-}3}$ (%)	DLM	I_{DD}
1	HL_Distr_wStream	OS	1.2GHz	2-2.5s	0.71s	varies	0 %	0.79-0.85A
2	HL_Distr_wStream	Sequential	1.2GHz	14.50s	0.42s	0/0/0/100	43 %	0.77A
3	HL_Distr_wStream	APP4MC	1.2GHz	1.88s	0.73s	25/25/55/35	0 %	0.75-0.81A
4	HL_Distr_wImageProc	OS	1.2GHz	3.4s	0.65s	varies	0 %	0.890-0.920A
5	HL_Distr_wImageProc	Sequential	1.2GHz	14.5s	0.38s	0/0/0/100	35 %	0.8A
6	HL_Distr_wImageProc	APP4MC	1.2GHz	6s	0.55s	80/55/57/30	5-23 %	0.85A
7	HL_Distr_AvgStress	OS	1.2GHz	3.34s	0.71s	varies	0 %	0.800-0.950A
8	HL_Distr_AvgStress	Sequential	1.2GHz	21.55s	0.38s	0/0/0/100	50 %	0.750A
9	HL_Distr_AvgStress	APP4MC	1.2GHz	8.66s	0.55s	30/100/5/35	22-33 %	0.760A
10	HL_Distr_FullStress	OS	1.2GHz	9.59s	0.57s	100/100/96/100	18 %	0.940.975A
11	HL_Distr_FullStress	Sequential	1.2GHz	59s	0.26s	0/0/0/100	68 %	0.750A
12	HL_Distr_FullStress	APP4MC	1.2GHz	13.48s	0.45s	75/100/85/95	18-22 %	0.88A
13	HL_Distr_wStream	OS	600MHz	2.2-2.6s	0.69s	varies	0 %	0.77-0.82A
14	HL_Distr_wStream	Sequential	600MHz	14.66s	0.43s	0/0/0/100	43 %	0.76A
15	HL_Distr_wStream	APP4MC	600MHz	1.94s	0.72s	25/25/55/35	0 %	0.73-0.75A
16	HL_Distr_wImageProc	OS	600MHz	4.0s	0.58s	varies	0 %	0.87-0.91A
17	HL_Distr_wImageProc	Sequential	600MHz	15.15s	0.38s	0/0/0/100	35-43 %	0.79A
18	HL_Distr_wImageProc	APP4MC	600MHz	6.1s	0.55s	80/55/57/30	31 %	0.82-0.85A
19	HL_Distr_AvgStress	OS	600MHz	3.40s	0.70s	varies	0 %	0.760A
20	HL_Distr_AvgStress	Sequential	600MHz	21.05s	0.34s	0/0/0/100	52 %	0.739A
21	HL_Distr_AvgStress	APP4MC	600MHz	8.70s	0.53s	30/100/5/35	22-33 %	0.74A
22	HL_Distr_FullStress	OS	600MHz	9.7s	0.54s	100/100/100/100	22 %	0.940A
23	HL_Distr_FullStress	Sequential	600MHz	57-59s	0.18-0.28s	0/0/0/100	68-75 %	0.740A
24	HL_Distr_FullStress	APP4MC	600MHz	13.70s	0.41-0.44s	75/100/85/95	18-31 %	0.87A

Table 5.6.: Distributions compared in High-level module

- Resulted APP4MC timing performance by looking at slack time averages and overall execution time is better than Sequential timing in all cases as expected.
- In a lot of cases, the Linux OS context-switching mechanism with no core affinity constraints performed better than APP4MC-based core affinity distributed results. However, we see that in `HL_Distr_wStream`, APP4MC scored better than OS distribution. Therefore, we can say that achieving a better performance than OS-based distribution in Linux-based systems is possible, but not guaranteed. In order to ensure to obtained

best results every time, models should be made precise and application needs to be as fine-grained as possible.

- As shown the table, in almost every distribution, APP4MC improved current and thereby power consumption by around 0.10-0.15A. It is observed that OS-based distribution resulted in the highest power consumption, whereas power consumption in APP4MC and Sequential distributions were similar. It should be noted that this is due to intensive context-switching in OS-based distribution when core affinities not constrained.
- It is also shown in table that changing the clock frequency alone improved the current consumption, but very slightly. One can observe this improvement from table which is about 0.01-0.05A. By using a lower chip voltage would make this power improvement a lot better. Due to safety concerns, this approach has not been applied to this work.
- One should also note that achieving significantly reduced power consumption is possible through load balancing. Since approaches used in APP4MC does not concentrate on pure load-balancing in non industry-type systems, the power consumption improvement observed in this evaluation is smaller.
- It must be noted that sequential distributions led to huge stability issues which were noticable from the system operation and are observable from deadline misses given in the table.
- Experiments made with ILP-based mapping algorithm and GA-based mapping algorithm gave similar results. Due to bugs in the ILP-based mapping algorithm at the time of evaluation (such as partitions not being mapped to a core at all), GA-based results are used.
- It can be seen that power consumption is higher when core usage is high. However, having high core usage improves the system timing performance. The reason why load balancing is important lies in this fact to have timing performance and power consumption improved at the same time.
- OS-based distribution scored mostly 0% deadline misses. However, APP4MC introduced slight deadline misses to the system and resulted in higher execution times. This can be reasoned by the following:
  - The real-time capability of Raspberry Pi is very low because the scheduler used in Raspbian (CFS) has high fairness. Furthermore, implemented schedulability and tracing features as well as Linux kernel introduces overheads. This effects the obtained results. Using a real-time kernel with minimal overhead would produce better results.

- Main purpose with APP4MC is that intends to move the embedded development in the direction of model-based development. It is used as a supplementary tool for standards such as AUTOSAR. Using the tool with a non-industry board and distribution (Raspberry Pi, Raspbian) is effecting the results because scheduling is left entirely to Linux kernel which does not ensure the causal order (dependencies) of runnables and it is not necessarily real-time.
- The fact that APP4MC scored better in `HL_Distr_wStream` indicate that when processes and threads are more fine-grained and dependency of processes and threads are lesser (thus, more parallelization potential due to non-sequential runnables), it is possible that APP4MC can produce better results than OS-based distributions.
- Technologies used in APP4MC require actual runnables to be partitioned and mapped. Since this is not possible for the Raspberry Pi, and only thread and process can be distributed among cores, threads and processes are modeled as runnables. Since runnable distribution and tracing in the lowest level would require too much effort for the scope of thesis, threads and processes are used. With this approach, dependencies between runnables and dependencies between threads are counted as the same. But in fact, it may not. APP4MC is built for more low-level design but in this work it is used in high-level design.
- APP4MC's partitioning algorithm uses activation period-based grouping by default. Having no activation grouping may result in better load balancing but unfeasible executions. Since in our particular application, periods used in are runnables are quite different, partitioning gave results that are not aimed toward load-balancing.
- Furthermore, APP4MC's mapping algorithm is not entirely focused on load balancing. The main optimization goal used in APP4MC's GA-based mapping algorithm (in version 0.8.1) is reducing the overall computation time. Partitions are distributed randomly otherwise. Therefore, OS scored better performance than APP4MC in timing because the load is not sufficiently balanced in APP4MC (comparing $U_{0\text{-}3}$ entries in Table 5.6).
- APP4MC focuses on ideal vision of embedded computing, that is using static scheduling to base parallelization foundations based on models. OS systems such as Linux uses dynamic scheduling and migrates all the tasks between several cores. Using static scheduling with models for specific optimization goals can suggest better outcomes.

To make sure that distributed processes (and not threads) affect the results, a second experiment has been conducted with only threads, formed by a *dummy graph* (given in Fig 3.33), running in the Linux system. This experiment also showed that the OS is capable of performing better in terms of timing. For example, an average slack time of 0.48s is obtained with OS-based distribution whereas APP4MC was able to score 0.44s. Thus, the results shown in this section are considered to show the actual performance of APP4MC. Furthermore, the Linux OS scheduler is able to split the graph more fine-grained because of the fact that it can migrate tasks between cores at any time.

To demonstrate how affinity constraints affected system performance, the work done in the tracing chapter is used and system traces have been taken. The 10 second system traces for entire Linux system are taken with perf [90], converted to CTF format [98], and visualized using Eclipse TraceCompass [95]. The Figure 5.5 demonstrates how OS-based, APP4MC-based, and sequential distributions for the `HL_Distr_AvgStress` is scheduled by Linux kernel among the Raspberry Pi's cores. Moreover, the distribution when APP4MC performed better than OS is also given with the Figure 5.6.

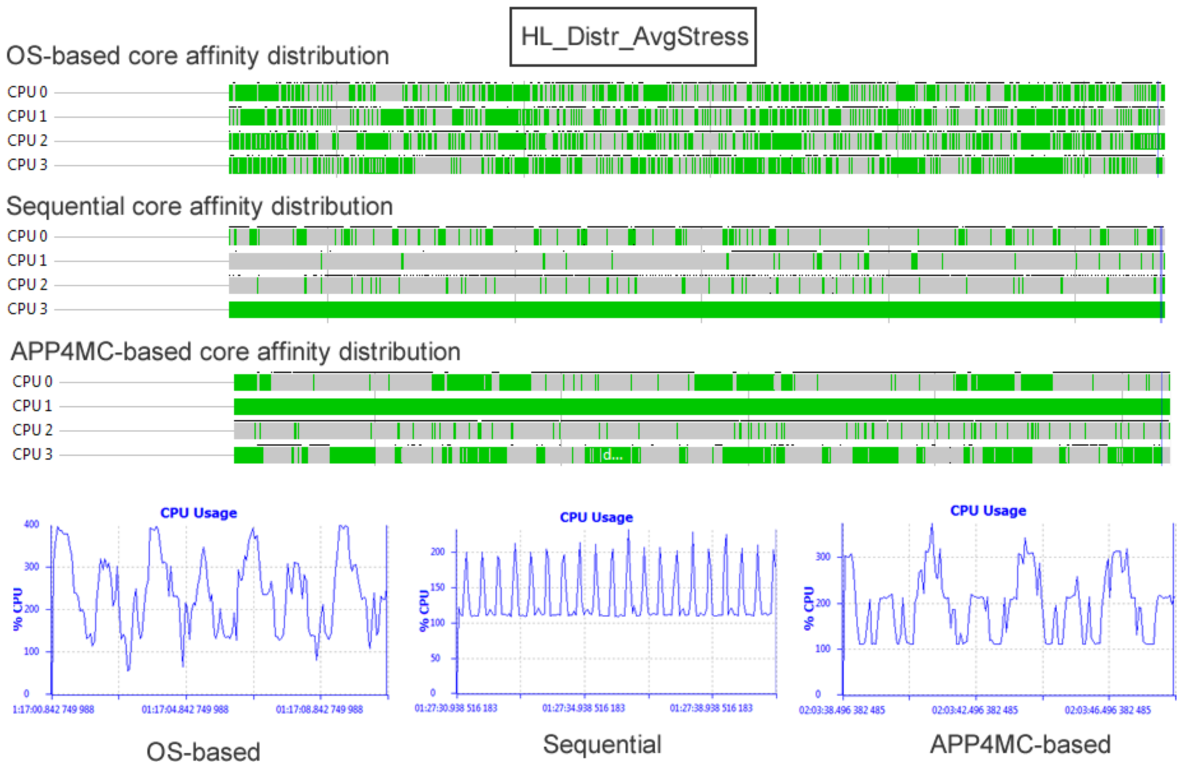


Figure 5.5.: System trace showing how processes and threads are distributed and how CPU performed in HL_Distr_AvgStress

In the aforementioned figures, it is shown that the load is most efficiently distributed in OS-based distribution, whereas APP4MC lacked balanced load in `HL_Distr_AvgStress`. In `HL_Distr_wStream`, APP4MC's load distribution looks more balanced. Therefore it can be shown that it is able to perform better than OS-based distribution slightly. This can be reasoned by stating that non fine-grained processes and threads such as dummy loads and

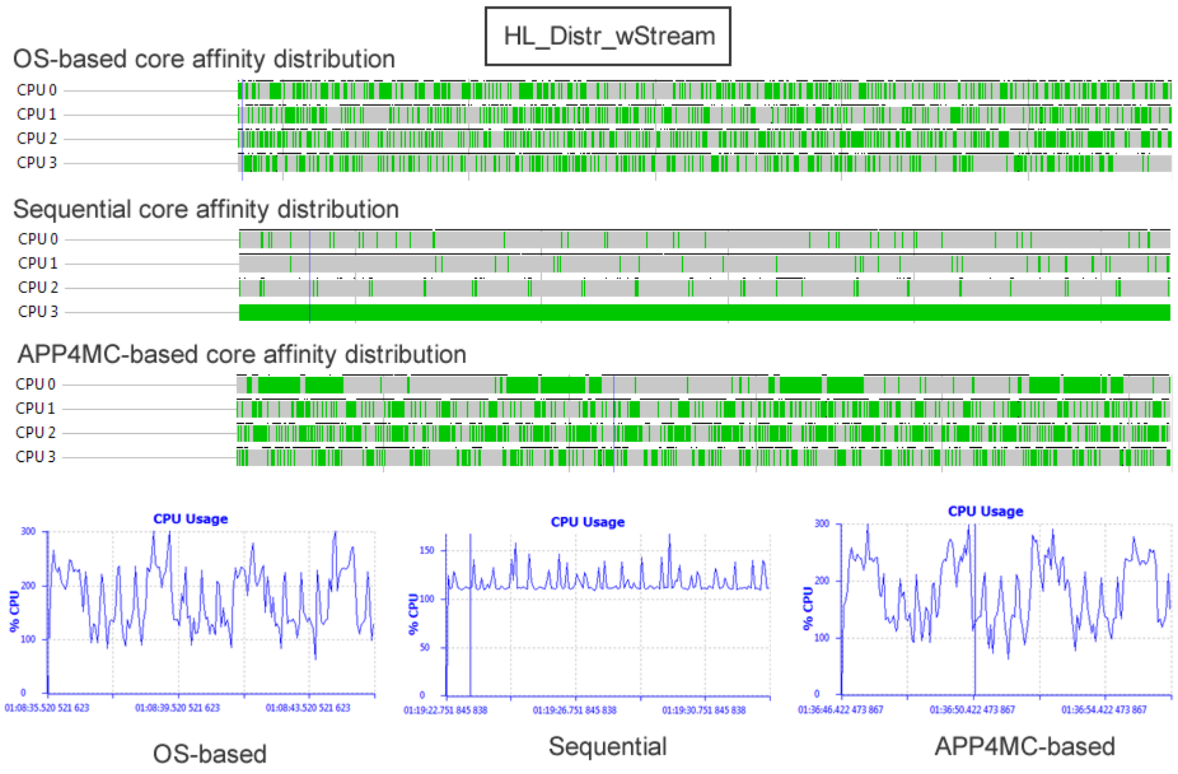


Figure 5.6.: System trace showing how processes and threads are distributed and how CPU performed in HL_Distr_wStream

image processing are not involved in `HL_Distr_wStream`.

New algorithms are now being developed for APP4MC to ignore dependencies to make sure APP4MC could be used for OS-based systems as well rather than low-level systems that ensure the causal order (dependencies) of runnables. Ignoring dependencies using bin-packing algorithms [104] will be used in APP4MC for producing results for applications running in complex OS-based systems.

5.4.2. Evaluation of Low Level Module Distributions

In the low-level module, the contribution of APP4MC is more essential because of the fact that the number of tasks exceed the number of cores and there is no automatic mapping implemented in xCORE. If the APP4MC is not used, the code can't even be compiled because some tasks should be squeezed in cores. If cores are not specified, xCORE gives an error stating that the number of cores required exceeded. The mapping is done manually at the compile-time, not automatically at run-time. Therefore, developers have to decide by themselves which tasks should be placed on which cores. Considering that the goal is minimizing the overall computation time and achieving a better parallel performance, APP4MC results help greatly in terms of low-level implementations.

In this context, to find the correct distribution to reduce the number of cores used and to compile the code properly, two distributions are presented. The first distribution, `LLM-Distribution-Unconstrained`, is the unconstrained version of the created APP4MC model. In this model, constraints that are used are kept at minimal level in order to see what would happen if the model is not properly engineered.

In xCORE-based software development, the coding experiments show that some types of tasks should be placed in cores seperately regardless of their load balancing such as the tasks that are more sporadically activated rather than periodically. In order not to interfere with the functionality of the applications, these apps are constrained in the model to be placed in seperate cores. The distribution that involves this approach is given with `LLM-Distribution- Constrained`.

5.4.2.1. LLM-Distribution-Unconstrained

In the first subsection of Section 5.4.2, the scope of `LLM-Distribution-Unconstrained` distribution is explained. The mapping results of this distribution using APP4MC could be seen in the Table 5.7. It is shown that in this model, the distribution is loosely constrained.

5.4.2.2. LLM-Distribution-Constrained

In the first subsection of Section 5.4.2, the scope of `LLM-Distribution-Constrained` distribution is explained. The additional constraints are applied to the existing model regarding sporadically activated tasks and the model details are shown with the Table 5.8.

5.4.2.3. Results of Low-level Evaluation

The low-level module distributions are compared at Table 5.9. Due to the limited available timers, the low-level module is evaluated with the available metrics such as slack time, execution time, and average utilization. The table shows that the unconstrained model is less utilized than the constrained model. Slack time of the unconstrained model being less than that of the constrained model suggests that timing performance of the constrained model is also better. It is important to add that unconstrained model results in functionality problems. As a result, one can learn that platform-specific and OS-specific constraints should be well engineered in the APP4MC model in order to achieve a higher performance and a more stable embedded system.

Task Name	**Granularity**	**Activation**	**Mapping**	**Core**	**Tile**
EthernetServer.TimerEvent	1000	Periodic 5s	Manual	1	0
EthernetServer.ShareCoreUsage0	50	Sporadic	Manual	1	0
EthernetServer.ShareCoreUsage1	50	Sporadic	Manual	1	0
EthernetServer.xtcp_event	100	Sporadic	Manual	1	0
ControlLightSystem.ST_Timer	1001	Periodic 0s-0.020s	Manual	0	0
ControlLightSystem.TH_Timer	1001	Periodic 0s-0.020s	Manual	0	0
ControlLightSystem.ShareState	50	Sporadic	Manual	0	0
Bluetooth.UART_RXDataReady	1017	Sporadic	Manual	4	0
Bluetooth.SendCmdEvent	127	Periodic 0.050s	Manual	4	0
Bluetooth.TimerEvent	1000	Periodic 0.050s	Manual	4	0
ServoController.ShareSteering	30	Sporadic	Manual	2	0
ServoController.TimerEvent	987	Periodic 0s-0.020s	Manual	2	0
ReadSonarSensors	519	Periodic 0.200s	Manual	-	0
DriveTBLE02S.ShareDistance	50	Sporadic	Manual	3	0
DriveTBLE02S.ShareDirection	50	Sporadic	Manual	3	0
DriveTBLE02S.ShareSpeed	50	Sporadic	Manual	3	0
DriveTBLE02S.TimerEvent	1001	Periodic 0s-0.020s	Manual	3	0
output_gpio	44	Sporadic	Manual	2	0
input_gpio	15	Sporadic	Manual	2	0
i2c_master	1188	Sporadic	Manual	7	0
uart_rx	448	Sporadic	Manual	2	0
uart_tx	71	Sporadic	Manual	2	0
xtcp	2000	Sporadic	Manual	-	0
MonitorCores0	245	Periodic 1s	Manual	6	0
MonitorCores1	245	Periodic 1s	Manual	-	1
smi	225	Sporadic	Manual	-	1
rgmii_ethernet_mac	4000	Sporadic	Manual	-	1
ar8035_phy_driver	75	Periodic 1s	Manual	-	1

Table 5.7.: LLM-Distribution-Unconstrained details in Low-level Module

Task Name	Granularity	Activation	Mapping	Core	Tile
EthernetServer.TimerEvent	1000	Periodic 5s	Manual	-	0
EthernetServer.ShareCoreUsage0	50	Sporadic	Manual	-	0
EthernetServer.ShareCoreUsage1	50	Sporadic	Manual	-	0
EthernetServer.xtcp_event	100	Sporadic	Manual	-	0
ControlLightSystem.ST_Timer	1001	Periodic 0s-0.020s	Manual	7	0
ControlLightSystem.TH_Timer	1001	Periodic 0s-0.020s	Manual	7	0
ControlLightSystem.ShareState	50	Sporadic	Manual	7	0
Bluetooth.UART_RXDataReady	1017	Sporadic	Manual	1	0
Bluetooth.SendCmdEvent	127	Periodic 0.050s	Manual	1	0
Bluetooth.TimerEvent	1000	Periodic 0.050s	Manual	1	0
ServoController.ShareSteering	30	Sporadic	Manual	4	0
ServoController.TimerEvent	987	Periodic 0s-0.020s	Manual	4	0
ReadSonarSensors	519	Periodic 0.200s	Manual	-	0
DriveTBLE02S.ShareDistance	50	Sporadic	Manual	-	0
DriveTBLE02S.ShareDirection	50	Sporadic	Manual	-	0
DriveTBLE02S.ShareSpeed	50	Sporadic	Manual	-	0
DriveTBLE02S.TimerEvent	1001	Periodic 0s-0.020s	Manual	-	0
output_gpio	44	Sporadic	Manual	-	0
input_gpio	15	Sporadic	Manual	0	0
i2c_master	1188	Sporadic	Manual	-	0
uart_rx	448	Sporadic	Manual	0	0
uart_tx	71	Sporadic	Manual	-	0
xtcp	2000	Sporadic	Manual	-	0
MonitorCores0	245	Periodic 1s	Manual	-	0
MonitorCores1	245	Periodic 1s	Manual	-	1
smi	225	Sporadic	Manual	-	1
rgmii_ethernet_mac	4000	Sporadic	Manual	-	1
ar8035_phy_driver	75	Periodic 1s	Manual	-	1

Table 5.8.: LLM-Distribution-Constrained details in Low-level Module

Distr. Name	GET	ST_{avg}	Avg. Utilization
LLM-Distribution-Unconstrained	0.14378s	0.02s	10 %
LLM-Distribution-Constrained	0.11930s	0.03s	12 %

Table 5.9.: Distributions compared in Low-level module

6. Conclusion

APP4MC is a platform that is used for engineering embedded multi-core and many-core embedded systems. It emphasizes especially on the automotive domain where standards such as AUTOSAR are highly involved. From the results, one can differentiate APP4MC from other available tooling as follows. With this report, several conclusions should be pointed out:

- In this work, APP4MC is evaluated using non-industry tools and platforms on a higher level. Therefore, using more lower level industry tools and platforms should produce better results. Moreover, Raspberry Pi's real-time capabilities are proven to be very poor because the scheduler used in Raspbian (CFS) has high fairness. Using a real-time kernel (such as preemptRT [105]), one should expect more well utilized software with APP4MC results.
- APP4MC focuses on static scheduling. By migrating tasks in between cores, OS dynamic schedulers introduce non-determinism in the design. However, with APP4MC every step is well-known and well-modeled. This way, many suggestive distributions can be generated using APP4MC. The developers should make design choices based on their optimization goals. Developers might redirect their projects towards load balancing, power optimization, total utilization etc. using APP4MC.
- With the involvement of the model-based approach, every step of the embedded design should be carefully modeled. With the results of the low-level module evaluations, it is shown that constraining the model for better accuracy improves the utilization outcome that is produced by the APP4MC. Involving model-based development aspect, on a more lower level, software design and deployment is more automatized once model is created successfully.
- Results show that APP4MC is able to result in core affinity restrictions that can produce better results than an operating system. In the experiments, operating systems mostly produced better utilization. However, using an accurate model and well-engineered fine-grained software, one can achieve even better performance than the operating systems.

- Achieving significantly reduced power consumption is made possible through load balancing. Since approaches used in APP4MC do not concentrate on pure load-balancing in non industry-type systems, the power consumption improvement observed in this evaluation is smaller. Yet, observed results with APP4MC improved the power consumption compared to the OS-based distributions.
- Invoking the underclocking capabilities, a slight improvement in the power consumption is observed. One can improve power consumption efficiency by activating Active Energy Consumption (AEC) modes or involving voltage reduction in the Dynamic Voltage and Frequency Scaling (DVFS) features as well.
- APP4MC's algorithms have the already mentioned limitations (listed below) in version 0.8.1
  - APP4MC's partitioning algorithm uses activation period-based grouping by default. Having no activation grouping may result in better load balancing but unfeasible executions. Since in our particular application, periods used in runnables are quite different, partitioning gave results that are not aimed toward load-balancing.
  - Furthermore, APP4MC's mapping algorithm is not focused on load balancing. The main optimization goal used in APP4MC's mapping algorithm is reducing the overall computation time. Partitions are distributed randomly otherwise. Therefore, OS scored better performance than APP4MC in timing in most cases because the load is not sufficiently balanced in APP4MC.
  - A4MCAR's software development involves non-determinism. For example, some tasks are not always deterministically activated. Especially in our applications, many tasks are sporadically activated. However, APP4MC is not able to process sporadically activated tasks and therefore assumptions have been made for these non-deterministic model elements.

  However, these algorithms are still in the improvement stage. New algorithms are now being developed for APP4MC to ignore dependencies to make sure APP4MC could be used for OS-based systems as well rather than low-level systems that ensure the causal order (dependencies) of runnables. Ignoring dependencies using bin-packing algorithms will be used in APP4MC for producing results for applications running in complex OS-based systems.

In this work, necessary tracing, distribution, and evaluation features are discovered and implemented regarding timing and power consumption in order to contribute to the open-source tool APP4MC. Required feedback is given to APP4MC community for progress. Furthermore, a base platform is developed for embedded multi-core development studies. With parallel implementations on the A4MCAR as well as other demonstrators, C++ and Python language-based multi-processed and multi-threaded embedded system is developed and maintained. Aforementioned contributions to the scientific community and open-source community are done.

The contributions to APP4MC demonstrators by this thesis' author are maintained in the following web links:

- **The APP4MC Demonstrator repository:**
  `https://git.eclipse.org/r/app4mc/org.eclipse.app4mc.examples`
- **Commits:**
  `http://git.eclipse.org/c/app4mc/org.eclipse.app4mc.examples.git/log/`
- **Source Tree:**
  `http://git.eclipse.org/c/app4mc/org.eclipse.app4mc.examples.git/tree/`
- **Documentation:**
  `https://mozcelikors.github.io/a4mcar/`

Multi-core processing is without a doubt today's and future's technology for information processing. Tools such as APP4MC are useful in terms of easing this technology for developers in a model-driven manner.

7. List of Figures

8. List of Tables

9. Listings

A. Bibliography

[1] NASA. What is NASA Doing with Big Data Today. https://open.nasa.gov/blog/what-is-nasa-doing-with-big-data-today/, accessed 08/2017.

[2] The Eclipse Foundation. APP4MC. https://projects.eclipse.org/proposals/app4mc, accessed 06/2017.

[3] Robert Höttger, Lukas Krawczyk, and Burkhard Igel. Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems. In *International Conference on Parallel, Distributed Systems and Software Engineering*, volume 2 of *ICPDSSE'15*, pages 2643–2649. World Academy of Science, Engineering and Technology, 2015.

[4] Robert Bosch GmbH. AMALTHEA4public. http://www.amalthea-project.org/, accessed 06/2017.

[5] XMOS Ltd. xCORE-200 eXplorerKIT. https://www.xmos.com/support/boards?product=18230, accessed 06/2017.

[6] Raspberry Pi Foundation. Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, accessed 06/2017.

[7] Robert Höttger, Mustafa Özcelikörs, Philipp Heisig, Lukas Krawczyk, Carsten Wolff, and Burkhard Igel. Constrained Mixed-Critical Parallelization for Distributed Heterogeneous Systems. In *The 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems, Technology and Applications*, 2017.

[8] Google Inc. Google Summer of Code 2017 Projects. https://summerofcode.withgoogle.com/projects/#5257433030066176, accessed 08/2017.

[9] Thomas Rauber and Gudula Rünger. *Parallel Programming*. Springer-Verlag Berlin Heidelberg, 2013.

[10] Miro Samek. Embedded Real-Time Systems vs General-Purpose Computers. http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/CUJ/2003/0302/cuj0302samek/cuj0302samek_s1.htm, accessed 09/2017.

[11] Dr. Wen-Chi Hou. Interconnection Networks. http://www2.cs.siu.edu/~cs401/Textbook/ch5.pdf, accessed 08/2017.

[12] CS-550: Distributed Shared Memory [SiS '94]. Distributed Resource Management: Distributed Shared Memory.

[13] Lukas Krawczyk, Robert Hoettger, and Uwe Lauschner. Introduction in DPS Architecture, Distributed and Parallel Systems Lecture Notes, ESM DPS WS 2015/2016.

[14] Jernej Barbic. Multi-core architectures. 2006.

[15] Gliwa GmbH embedded systems. Timing Poster. https://www.gliwa.com/downloads/Timing%20Poster.pdf, accessed 07/2017.

[16] expresslogic. What Is an RTOS and Why Use One? http://rtos.com/PDFs/What_Is_An_RTOS_and_Why_Use_One_Embedded.com_.pdf, accessed 08/2017.

[17] ARM Ltd. CMSIS-RTOS Documentation. http://www.keil.com/pack/doc/CMSIS/RTOS/html/index.html, accessed 09/2017.

[18] Python Software Foundation. threading - Higher-level threading interface. https://docs.python.org/2/library/threading.html, accessed 08/2017.

[19] Stack Exchange Inc. What makes a kernel/OS real-time? https://stackoverflow.com/questions/22241264/what-makes-a-kernel-os-real-time?rq=1, accessed 08/2017.

[20] SCALE 13X Steve Doran. How to Perform Real-Time Processing on the Raspberry Pi. https://www.socallinuxexpo.org/sites/default/files/presentations/Steven_Doran_SCALE_13x.pdf, accessed 08/2017.

[21] Robert Warschofsky. AUTOSAR Software Architecture. https://hpi.de/fileadmin/user_upload/fachgebiete/giese/Ausarbeitungen_AUTOSAR0809/\Software_Architecture_Warschofsky.pdf, accessed 06/2017.

[22] The Eclipse Foundation. Application Platform Project for MultiCore (APP4MC). https://www.eclipse.org/app4mc/, accessed 06/2017.

[23] Eclipse APP4MC. APP4MC 0.8.0 Documentation. https://www.eclipse.org/app4mc/help/app4mc-0.8.0/index.html, accessed 08/2017.

[24] The Eclipse Foundation. Eclipse Public License - v 1.0. https://www.eclipse.org/legal/epl-v10.html, accessed 06/2017.

[25] Robin J. Wilson. *Introduction to Graph Theory: Fourth Edition*. Prentice Hall, 1996.

[26] C. Wolff, L. Krawczyk, R. Höttger, C. Brink, U. Lauschner, D. Fruhner, E. Kamsties, and B. Igel. AMALTHEA - Tailoring tools to projects in automotive software development. In *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 2, pages 515–520, Sept 2015.

[27] Robert Höttger, Lukas Krawczyk, and Burkhard Igel. Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 9(1):268 – 274, 2015.

[28] A. Sailer, S. Schmidhuber, M. Hempe, M. Deubzer, and J. Mottok. Distributed Multi-Core Development in the Automotive Domain - A Practical Comparison of ASAM MDX vs. AUTOSAR vs. AMALTHEA. In *ARCS 2016; 29th International Conference on Architecture of Computing Systems*, pages 1–8, April 2016.

[29] Devika K and Syama R. An Overview of AUTOSAR Multicore Operating System Implementation. In *International Journal of Innovative Research in Science, Engineering, and Technology Vol. 2, Issue 7*, July 2013.

[30] N. Navet, A. Monot, B. Bavoux, and F. Simonot-Lion. Multi-source and multicore automotive ECUs - OS protection mechanisms and scheduling. In *2010 IEEE International Symposium on Industrial Electronics*, pages 3734–3741, July 2010.

[31] M. Alfranseder, M. Deubzer, B. Justus, J. Mottok, and C. Siemers. An efficient spin-lock based multi-core resource sharing protocol. In *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, pages 1–7, Dec 2014.

[32] F. W. Yu, B. H. Zeng, Y. H. Huang, H. I. Wu, C. R. Lee, and R. S. Tsay. A Critical-Section-Level timing synchronization approach for deterministic multi-core instruction-set simulations. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 643–648, March 2013.

[33] Y. Lu, T. Nolte, I. Bate, J. Kraft, and C. Norström. Assessment of trace-differences in timing analysis for Complex Real-Time Embedded Systems. In *2011 6th IEEE International Symposium on Industrial and Embedded Systems*, pages 284–293, June 2011.

[34] S. Nilakantan, K. Sangaiah, A. More, G. Salvadory, B. Taskin, and M. Hempstead. Synchrotrace: synchronization-aware architecture-agnostic traces for light-weight multicore simulation. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 278–287, March 2015.

[35] David Wentzlaff, Patrick Griffin, and Henry Hoffmann et al. On-chip Interconnection Architecture of the Tile Processor. pages 15–31. IEEE Computer Society, 2017.

[36] XMOS Ltd. xCORE Architecture Flyer. `http://www.xmos.com/download/private/xCORE-Architecture-Flyer%281.1%29.pdf`, accessed 06/2017.

[37] XMOS Ltd. XE216-512-TQ128-Datasheet. `http://www.xmos.com/download/private/XE216-512-TQ128-Datasheet%281.12%29.pdf`, accessed 06/2017.

[38] Leonard Kleinrock. Analysis of A time-shared processor. *Naval Research Logistics Quarterly*, 11(1):59–73, 1964.

[39] Carl Hamacher. *Computer Organization (5th Edition)*. McGraw Hill Higher Education, 2001.

[40] XMOS Ltd. XMOS Programming Guide. `https://www.xmos.com/download/private/XMOS-Programming-Guide-(documentation)(E).pdf`, accessed 06/2017.

[41] Raspberry Pi Foundation. Raspbian. `https://www.raspberrypi.org/downloads/raspbian/`, accessed 06/2017.

[42] Ubuntu MATE Team. Ubuntu MATE for the Raspberry Pi 2 and Raspberry Pi 3. `https://ubuntu-mate.org/raspberry-pi/`, accessed 06/2017.

[43] kernel.org. CFS Scheduler. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`, accessed 08/2017.

[44] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing, Inc., 2008.

[45] Brian Ward. *How Linux Works*. No Starch Press, 2014.

[46] SparkFun. Pulse Width Modulation. `https://learn.sparkfun.com/tutorials/pulse-width-modulation`, accessed 06/2017.

[47] Rudra Pratap Suman. UART. `http://students.iitk.ac.in/eclub/assets/lectures/summer12/uart.pdf`, accessed 06/2017.

[48] SparkFun. I2C. `https://learn.sparkfun.com/tutorials/i2c/all.pdf`, accessed 06/2017.

[49] TechTarget. Ethernet. `http://searchnetworking.techtarget.com/definition/Ethernet`, accessed 06/2017.

[50] Nortel Networks. Unit 4 : Introduction to TCP/IP. `http://k-12.pisd.edu/currinst/network/if4_1st.pdf`, accessed 06/2017.

[51] SparkFun. Serial Peripheral Interface. https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all.pdf, accessed 06/2017.

[52] PishRobot. SRF02 Ultrasonic range finder Technical Specification. http://www.pishrobot.com/files/products/datasheets/srf02.pdf, accessed 06/2017.

[53] Adafruit. 4-channel I2C-safe Bi-directional Logic Level Converter. https://www.adafruit.com/product/757, accessed 06/2017.

[54] XP Power. JCA Series. http://www.xppower.com/pdfs/SF_JCA04-06.pdf, accessed 06/2017.

[55] Tamiya. TT-01 TYPE-E Chassis. http://www.tamiya.com/english/rc/rcmanual/tt01_type_e.pdf, accessed 06/2017.

[56] Sanford Friedenthal, Alan Moore, and Rick Steiner. OMG Systems Modeling Language Tutorial. 2006-2009.

[57] Roving Networks. RN-41-EK RN-42-EK Evaluation Kit User's Guide. http://ww1.microchip.com/downloads/en/DeviceDoc/rn-4142-ek-ug-1.0.pdf, accessed 07/2017.

[58] Indiana University Knowledge Base. What is telnet? https://kb.iu.edu/d/aayd, accessed 07/2017.

[59] Python Software Foundation. Python 2.7.14rc1 Documentation. https://docs.python.org/2/, accessed 07/2017.

[60] Free Software Foundation Inc. GNU C Compiler. https://gcc.gnu.org/onlinedocs/gcc-6.3.0/gcc/, accessed 07/2017.

[61] Mikhail Kupchik. Raspberry Pi Eclipse Tutorial. http://www.gurucoding.com/en/raspberry_pi_eclipse/index.php, accessed 07/2017.

[62] The Linux Documentation Project. Advanced Bash-Scripting Guide. http://tldp.org/LDP/abs/html/, accessed 07/2017.

[63] Refsnes Data W3Schools. AJAX Introduction. https://www.w3schools.com/xml/ajax_intro.asp, accessed 07/2017.

[64] The jQuery Foundation. jQuery. https://jquery.com/, accessed 07/2017.

[65] University of Maryland Computer Science Dept. Computer Performance. https://cs.umd.edu/class/spring2015/cmsc411-0201/lectures/lecture04_performance_reliability.pdf, accessed 07/2017.

[66] Stack Exchange Inc. User CPU time vs System CPU time. http://stackoverflow.com/questions/4310039/user-cpu-time-vs-system-cpu-time, accessed 07/2017.

[67] Python Software Foundation. psutil. https://pypi.python.org/pypi/psutil, accessed 07/2017.

[68] Python Software Foundation. socket - Low-level networking interface. https://docs.python.org/2/library/socket.html, accessed 07/2017.

[69] Igor Ljubuncic. Apache Web Server Complete Guide. 2011.

[70] Bhupendra Ratha. Web Server. http://www.clib.dauniv.ac.in/E-Lecture/Web%20Server.pdf, accessed 07/2017.

[71] jacksonliam. mjpg_streamer. https://github.com/jacksonliam/mjpg-streamer, accessed 07/2017.

[72] jqPlot. jqPlot: pure javascript plotting. http://www.jqplot.com/, accessed 07/2017.

[73] OpenCV team. Open Source Computer Vision Library. http://opencv.org/, accessed 08/2017.

[74] Pygame. Pygame. https://www.pygame.org/, accessed 07/2017.

[75] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1), January/February 1998.

[76] Google Inc. Android. https://www.android.com/, accessed 07/2017.

[77] Google Inc. Android Studio. https://developer.android.com/studio/index.html, accessed 07/2017.

[78] controlwear. Virtual Joystick Android. https://github.com/controlwear/virtual-joystick-android, accessed 07/2017.

[79] Digital Security group Erik Poll. Static Analysis aka Source code analysis. https://www.cs.ru.nl/E.Poll/ufrj/5_StaticAnalysisPREfast.pdf, accessed 07/2017.

[80] Nicholas Nethercote. Dynamic Binary Analysis and Instrumentation. 2004.

[81] testingexcellence.com. Static Analysis vs Dynamic Analysis in Software Testing. http://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/, accessed 07/2017.

[82] The IPM developers. Profiling vs Tracing. http://ipm-hpc.sourceforge.net/profilingvstracing.html, accessed 07/2017.

[83] The Linux Documentation Project. Chapter 7: System Monitoring. http://ipm-hpc.sourceforge.net/profilingvstracing.html, accessed 07/2017.

[84] XMOS Ltd. XMOS Timing Analyzer Manual. https://www.xmos.com/download/private/XMOS-Timing-Analyzer-Manual%281.2%29.pdf, accessed 07/2017.

[85] XMOS Ltd. Estimating Power Consumption For XS1-L Devices. https://www.xmos.com/download/private/AN01005%3A-Estimating-Power-Consumption-For-XS1-L-Devices%281.0.2rc1%29.pdf, accessed 07/2017.

[86] Diary R. Suleiman, Muhammad A. Ibrahim, and Ibrahim I. Hamarash. Dynamic Voltage Frequency Scaling (DVFS) for Microprocessors Power and Energy Reduction. 2005.

[87] Shi-Hao Chen and Jiing-Yuan Lin. Implementation and verification practices of DVFS and power gating. In *2009 International Symposium on VLSI Design, Automation and Test*, pages 19–22, April 2009.

[88] sourceware.org. objdump. https://sourceware.org/binutils/docs/binutils/objdump.html, accessed 07/2017.

[89] Python Software Foundation. dis - Disassembler for Python bytecode. https://docs.python.org/2/library/dis.html, accessed 07/2017.

[90] Kernel.org. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, accessed 07/2017.

[91] Brendan Gregg. perf Examples. http://www.brendangregg.com/perf.html, accessed 07/2017.

[92] Tecmint. 12 TOP Command Examples in Linux. https://www.tecmint.com/12-top-command-examples-in-linux/, accessed 07/2017.

[93] The Linux Documentation Project. Linux Filesystem Hierarchy: /proc. http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html, accessed 07/2017.

[94] LWN.net. trace-cmd: A front-end for Ftrace. https://lwn.net/Articles/410200/, accessed 07/2017.

[95] TraceCompass. TraceCompass. http://tracecompass.org, accessed 07/2017.

[96] The LTTng Project. LTTng. http://lttng.org/, accessed 07/2017.

[97] Kernel.org. Linux kernel sources.

[98] Stack Exchange Inc. Building Perf with Babeltrace (for Perf to CTF Conversion). `http://stackoverflow.com/questions/43576997/building-perf-with-babeltrace-for-perf-to-ctf-conversion`, accessed 07/2017.

[99] Die.Net. taskset(1) - Linux man page. `https://linux.die.net/man/1/taskset`, accessed 07/2017.

[100] Etienne Le Sueur and Gernot Heiser. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[101] Debian Wiki. CPU Frequency Scaling. `https://wiki.debian.org/HowTo/CpuFrequencyScaling`, accessed 08/2017.

[102] Gateworks. Dynamic Voltage and Frequency Scaling (DVFS). `http://trac.gateworks.com/wiki/DVFS`, accessed 08/2017.

[103] Lukas Krawczyk, Carsten Wolff, and Daniel Fruhner. *Automated Distribution of Software to Multi-core Hardware in Model Based Embedded Systems Development*, pages 320–329. Springer International Publishing, Cham, 2015.

[104] Operations Research Group Bologna. Bin-packing problem. `http://www.or.deis.unibo.it/kp/Chapter8.pdf`, accessed 08/2017.

[105] Linux Foundation. Real-time Linux Wiki. `https://rt.wiki.kernel.org/index.php/Main_Page`, accessed 09/2017.

[106] Microchip. Ethernet Theory of Operation. `http://http://ww1.microchip.com/downloads/en/AppNotes/01120a.pdf`, accessed 10/2017.

[107] The Eclipse Foundation. PolarSys. `https://www.polarsys.org/`, accessed 10/2017.

B. Eidesstattliche Erklärung

Gemäß § 17,(5) der BPO erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Ich habe mich keiner fremden Hilfe bedient und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, October 9, 2017 Mustafa Özçelikörs

Erklärung

Mir ist bekannt, dass nach § 156 StGB bzw. § 163 StGB eine falsche Versicherung an Eides Statt bzw. eine fahrlässige falsche Versicherung an Eides Statt mit Freiheitsstrafe bis zu drei Jahren bzw. bis zu einem Jahr oder mit Geldstrafe bestraft werden kann.

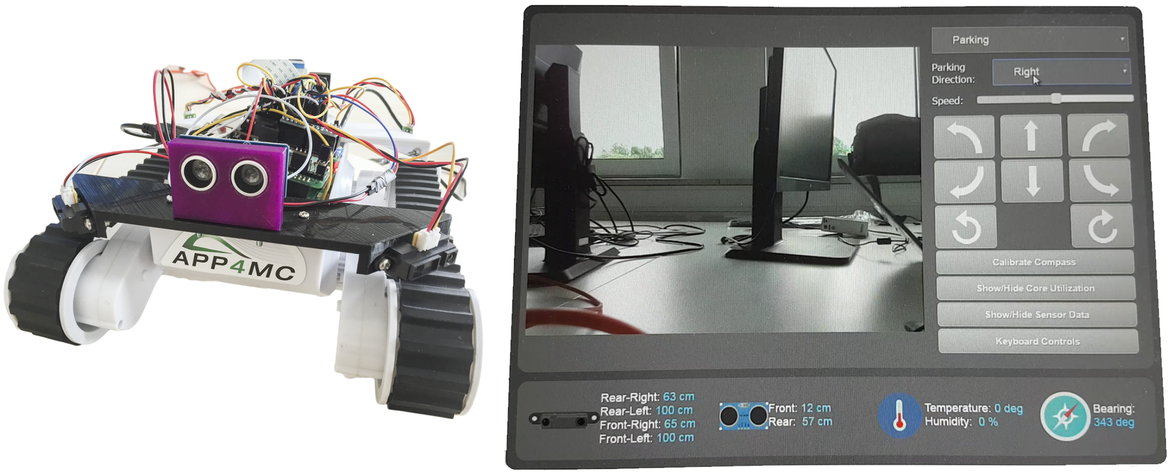
Dortmund, October 9, 2017 Mustafa Özçelikörs

C. About Rover Project

The A4MCAR is not the only demonstrator for APP4MC. In the Rover project, investigation of APP4MC's effectiveness is done by using a more stable POSIX thread-based platform using only C++ language. The developments for the Rover have lots of synergies with Eclipse's PolarSys project [107]. APP4MC Rover also requires a lot of effort towards Cloud & IoT based developments such as implementing a bidirectional data communication with an Eclipse Hono cloud instance.

The author of this thesis is also engaged with tackling tasks in the Rover project. Ensuring schedulable and traceable thread-based software architecture is one of the most crucial challenges in the Rover project. The Rover is developed with a much more advanced web interface that can display sensor information, utilization information, control the Rover, and switch between driving modes such as manual driving, Adaptive Cruise Control, and Parking. Furthermore, connectivity display on OLED, bluetooth-based driving using RFCOMM sockets, and many more features will be developed. Some figures are given with the Figure C.1 that depicts the progress in the Rover project. The Rover project is developed at IDiAL Institute of University of Applied Sciences and Arts Dortmund and it is accessible from the following link:

http://git.eclipse.org/c/app4mc/org.eclipse.app4mc.examples.git/tree/



Parking Direction: Right
Speed:

Rear-Right: 63 cm
Rear-Left: 100 cm
Front-Right: 65 cm
Front-Left: 100 cm

Front: 12 cm
Rear: 57 cm

Temperature: 0 deg
Humidity: 0 %

Bearing: 343 deg

Figure C.1.: Rover project and its web interface