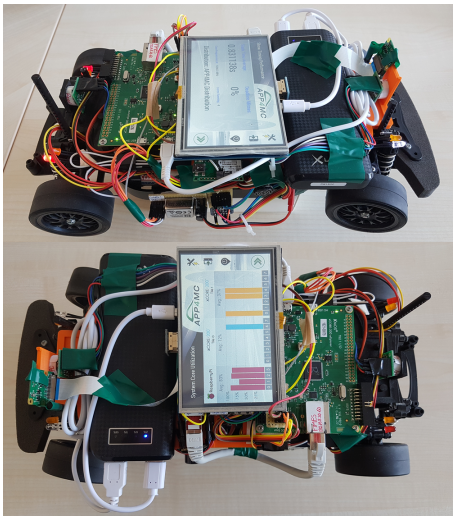
A4MCAR Documentation

Documentation on Implementation and Parallelism Evaluation of a APP4MC-based Multi-core Demonstrator

Version *August 18, 2017*
Author *Mustafa Özcelikörs* *mozcelikors@gmail.com*
Supervision & revision *Robert Höttger* *robert.hoettger@fh-dortmund.de*

University of Applied Sciences and Arts Dortmund
IDiAL Institute, Project AMALTHEA4public
BMBF Fund.Nb. 01—S14029K

Also granted by Google Inc. and The Eclipse Foundation during the participation on Google Summer of Code 2017.



Fachhochschule Dortmund
University of Applied Sciences and Arts

IDiAL Institute for Digital Transformation of Application and Living Domains

Contents

1 Scope **4**

2 Introduction **4**
2.1 Motivation . . . 4
2.2 Objective and Contributions . . . 4
2.3 Methodology . . . 5
2.4 Events and Publications . . . 6

3 Preliminaries **7**
3.1 Introduction to Parallelism . . . 7
3.2 Essential Concepts in Parallelization . . . 7
3.3 Design Techniques in Parallelization . . . 7
3.4 Analysis of Scheduling . . . 8
3.5 Optimization and Evaluation of Parallel Software . . . 10

4 Design Using APP4MC **11**
4.1 Related Work . . . 12

5 Implementations on A4MCAR **15**
5.1 A4MCAR Features . . . 15
5.2 Low-level Infrastructure . . . 16
5.3 High-level Infrastructure . . . 18
5.4 Low-level Module Implementation . . . 20
5.5 High-Level Module Implementation . . . 21
5.5.1 Overview . . . 21
5.5.2 Implemented Online Timing Features and Making Processes Schedulable . . . 22
5.5.3 Core Reader . . . 26
5.5.4 Ethernet (TCP) Client Implementation . . . 26
5.5.5 Web Server . . . 27
5.5.6 Web Page Design and Implementation . . . 27
5.5.7 Controlling A4MCAR via Web Page . . . 28
5.5.8 Camera Streaming . . . 30
5.5.9 Core Utilization Display . . . 31
5.5.10 Dummy Loads and Dummy Graph . . . 31
5.5.11 Image Processing with OpenCV . . . 32
5.6 Touchscreen Display . . . 32
5.6.1 Touchscreen Display Features . . . 32
5.6.2 Touchscreen Display Implementation . . . 34
5.6.3 VNC Server . . . 37
5.7 Android Application Implementation . . . 37

6 Exploring Tracing, Mapping, and Energy Consumption Features **40**
6.1 Introduction . . . 40
6.2 Low-Level Module Information Tracing and System Management . . . 41
6.2.1 Static Binary Analysis via XTA . . . 41
6.2.2 Distribution of Tasks to Cores . . . 43
6.2.3 System Monitoring in xCORE . . . 43
6.2.4 Discovering Energy Consumption Features . . . 44
6.3 High-Level Module Information Tracing and System Management . . . 46
6.3.1 Binary Analysis of Instructions . . . 46
6.3.2 Process Management and Monitoring . . . 46
6.3.3 System Monitoring for Linux Platform . . . 48
6.3.4 Tracing the System to Obtain Scheduling Information . . . 49
6.3.5 Process and Thread Mapping . . . 53
6.3.6 Discovering Energy Consumption Features . . . 55
6.3.7 Online Timing Analysis Features in A4MCAR . . . 56

7 Modeling and Results **57**
7.1 System Limitations and Factors that Affect the Results 57
7.2 Modeling the A4MCAR using APP4MC 57
7.3 Partitioning and Mapping 59
7.4 Evaluation of Different Distributions 60
7.4.1 APP4MC Results for the Distribution HL_Distr_wStream 62
7.4.2 APP4MC Results for the Distribution HL_Distr_wImageProc 62
7.4.3 APP4MC Results for the Distribution HL_Distr_AvgStress 63
7.4.4 APP4MC Results for the Distribution HL_Dist_FullStress 63
7.4.5 Comparison of High-level Module Distributions and Results 63

8 About Using the Software **67**
8.1 Preliminaries 67
8.2 Re-using the Software 67
8.3 Re-using the Scripts 68

1 Scope

This report is dedicated to explaining the A4MCAR demonstrator: motivations behind it, objectives, software and hardware features, tool support for tasks such as binary analysis, profiling, tracing, monitoring, and model-based parallelization fundamentals using the APP4MC platform.

2 Introduction

2.1 Motivation

Developing and distributing effective software 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 created a great demand for parallel software development in the last ten years. 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.

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. 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 [17]. 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 researched 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 could also make use of less energy consumption features by applying techniques such as under-clocking a processor. To summarize, developing efficient parallel software is not only useful in achieving advanced computing capability 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.

2.2 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 determine and plan not only the hardware constraints but also the software constraints in order to create efficient and reliable systems.

Before its execution, parallel software has to be delicately planned. The first stage of the parallel software development, the planning stage, involves several activities such as Modeling, Partitioning, Task generation and Mapping. 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.

While there exist some commercial tools that provide easement in the parallel software development, recent study done in Germany, namely AMALTHEA4public [61] [45], aims to provide planning and tracing tools especially for multi-core developments in automotive domain with several open source development tools. The branch of AMALTHEA4public, the APP4MC project [19] 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 [19]:

- 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 [19].

This documentation 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, 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 more optimized distributed and parallel software. Furthermore, the Eclipse APP4MC community will 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.

2.3 Methodology

Automotive or any vehicle control related field tends to require very complex systems. In a real-life automotive application, amount of hardware nodes and software nodes are high in number. Since the main focus of the APP4MC environment is to provide parallel computation tools for automotive domain, a demonstrator is required that is closely related to automotive domain and that can be 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 matching a vehicle's distributed architecture such as the AUTOSAR used in vehicles. Furthermore, 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) and a 4-core single board computer (Raspberry Pi 3) 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 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 report, 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.

2.4 Events and Publications

This project has been published partially in several forms. With the supervision of Robert Höttger, the author submitted a paper on mixed-critical parallelization of distributed and heterogeneous systems [62] 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 close partnership of APP4MC project to Eclipse Foundation, the author of this paper published a contribution called "Developing a multi-core enhanced RC-CAR using APP4MC" on 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 in 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 in 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" [22] 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 (EPL). Thus, the intention of helping Real-time and parallel software development community was supported crucially during the participation on GSoC'17.

3 Preliminaries

3.1 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 computer systems 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 [73], 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 [73].

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 could be time consuming. In this regard, one should know how 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 [73].

3.2 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**. 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** [73]. 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** can be defined as a program in execution that is assigned to execution resources for execution. 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 [73].

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 on 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. Usage of these techniques for parallel design brings their own 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 [65]. **Synchronization** is also an important job which defines the organized communication between processes or threads [73]. 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.

Finding a useful scheduling and mapping strategy is key to 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 [73], for the quantitative evaluation of parallel programs, measures such as **speedup** and **efficiency** can be used which are the measures that compare parallel execution time of a software with its sequential execution time.

3.3 Design Techniques in Parallelization

Parallelization can be defined as the transformation of a sequential program into a parallel program [73]. Although different sources ([73] and [65]) 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 1):

- **Partitioning of the problem:** The application that addresses a problem should be decomposed into smaller pieces, i.e. runnables that are considered to be smallest pieces in a software. In some cases, this

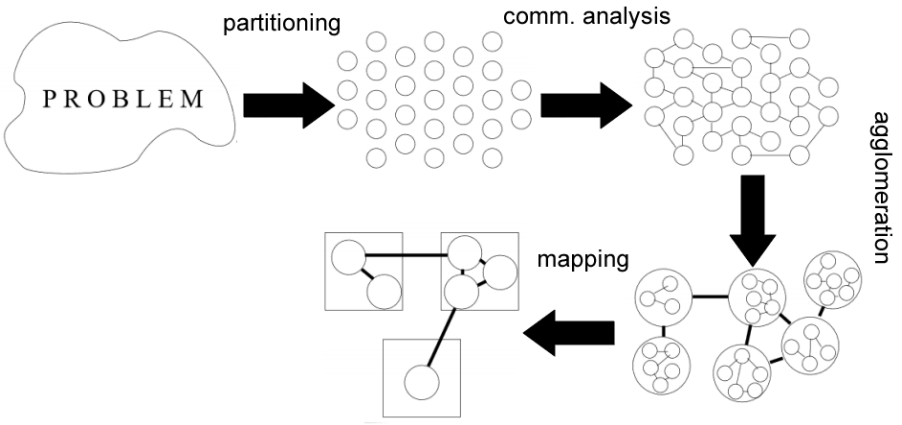


Figure 1: Illustration of design techniques in parallelization [65]

decomposition can be at task-level. A runnable or a small task can be a function that involves a single read or write access 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. The goal of this decomposition (according to [73]) is to make sure that all the applications are fine-grained enough that the load balancing can be achieved. In other words, runnables have small granularity and with the help of this the generated tasks can be distributed more efficiently.

- **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.
- **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 it should consider load balancing.
- **Assignment of tasks to processes or threads:** According to [73], 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 is the step which makes the software mapping-ready. In the cases of some multi-core microcontrollers, 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 is 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. Mapping phase in computer systems is usually done by the operating system but the users can intervene. The main goal of mapping step is to get an equal utilization while keeping the communication overhead smallest [73] [65].

3.4 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 are scheduled properly. To meet the optimization goals, this work involves evaluation of several software distributions.

According to [65], Quality of Service (QoS) of scheduling could involve the following goals:

- Even load-balancing
- Efficient resource usage
- Maximizing throughput (completed processes per time unit) and utilization (percentage a processor is used)
- Minimizing response time and latency

- Maximizing 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 later sections, the optimization goals for parallel software will be 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. In the figure, it is seen that two tasks are given with Task B having a higher priority but lesser execution time. Timing properties of the given timeline could be defined as follows [32] [65]:

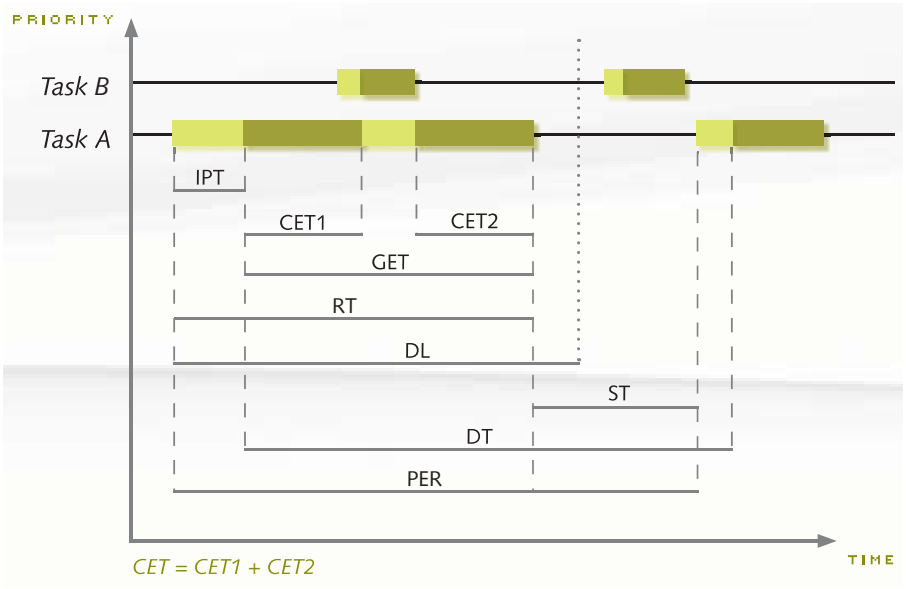


Figure 2: Timing properties for scheduling in multi-tasked systems [32]

- **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.
- **Gross Execution Time (GET)**: 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)**: Response time is calculated by adding IPT and GET.
- **Deadline (DL)**: Deadline of a task or an event is the amount of time a task should be completed in order to meet the real-time requirements of the task. Therefore, a reliable task should 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.
- **Delta Time (DT)**: Dela time of a task can be defined as the time between two gross executions of a task.
- **Slack Time (ST)**: 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. Slack time describes the flexibility of a task, thus it is the time that 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)**: 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 report 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.

3.5 Optimization and Evaluation of Parallel Software

Not every parallel program are beneficial. In order for a parallel program to be useful, it should be optimized. There are four main reasons a parallel software should 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 3) may 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 3. 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. However, the figure given shows the basic software optimization goals that are related to this work.

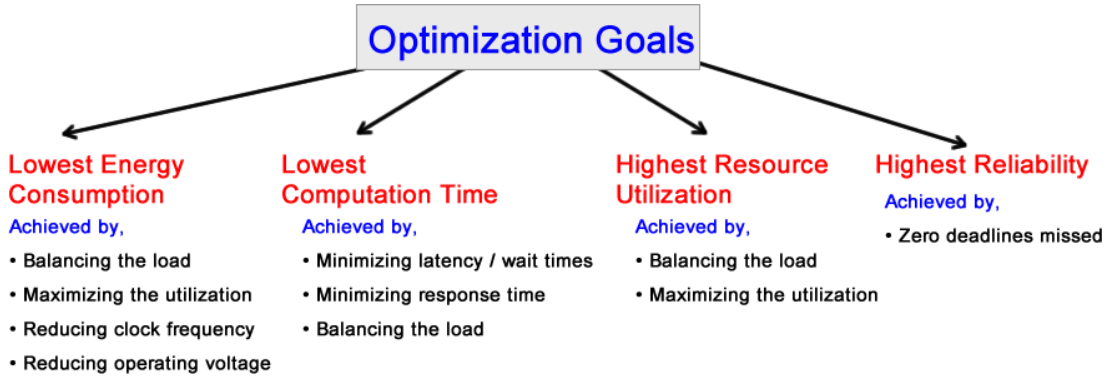


Figure 3: Optimization goals of parallel software [65]

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 [65]. Those strategies involve Linear Programming, Integer Linear Programming, Mixed Integer Linear Programming, Quadratic Programming, Evolutionary Algorithm, and Simulated Annealing [65]. 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 helped 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 [73]. Speedup is calculated as follows [73], given that $T^*(n)$ is the sequential run-time and $T_{\text{p}}(n)$ is the parallel run-time:

$$S_{\text{p}}(n) = \frac{T^*(n)}{T_{\text{p}}(n)} \tag{1}$$

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

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

4 Design Using APP4MC

As introduced in the Introduction chapter, this work uses Eclipse APP4MC development tool for software parallelization. **APP4MC (Application Platform Project for MultiCore)** [61] [29] is an Eclipse-based project that aims to achieve an open, consistent, expandable tool platform for embedded software engineering [30]. APP4MC targets multi-core and many-core platforms, while the main focus is the optimization of embedded multi-core systems [30]. 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 [29]. Additionaly, since APP4MC uses Eclipse platform to its purposes, the development environment has a complete open-source nature under Eclipse Public License (EPL) [31].

Eclipse APP4MC platform editor window can be seen in the Figure 4 [30]. In the figure, 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 [30].

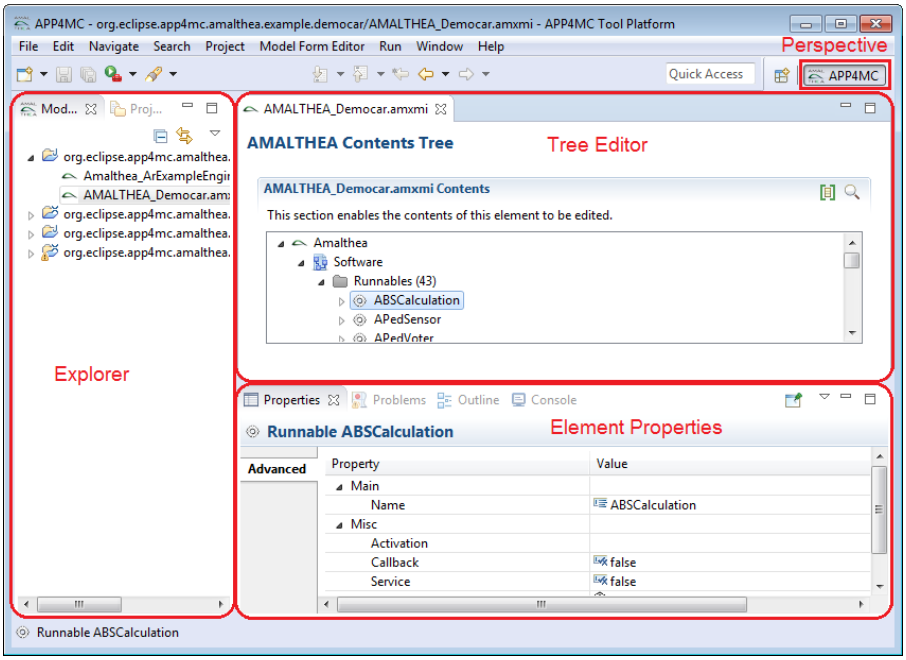


Figure 4: Eclipse APP4MC platform Editor Window [30]

APP4MC is a project that has a lot of synergies with its predecessor AMALTHEA4public [45] project. APP4MC uses AMALTHEA models, which are XML models that describe software components and hardware platforms. Main operation of APP4MC involves modeling the system by creating AMALTHEA models and performing partitioning, mapping, optimization on parallel programs [61]. APP4MC also has the ability to trace simulate parallel programs. Basic ingredients for an AMALTHEA model is illustrated in the Figure 5 [29]. It is seen that AMALTHEA model can contain software decisions, costs, constraints, as well it can contain hardware platform information [29]. 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 the Figure 6. Studying the illustration given in the Figure 6 in combination with the parallel design techniques, 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. Hardware model involves information such as number of processor cores, system clock frequency for processors, and memory details. In the software model, runnables are modeled. Runnables are found with the help of binary analysis tools and by using the decomposition technique. Each runnable are modeled by making use of information such as granularity (number of instructions) and label accesses (memory read-write). In the early development stages, model contains a rough model of the software, but the model is constantly

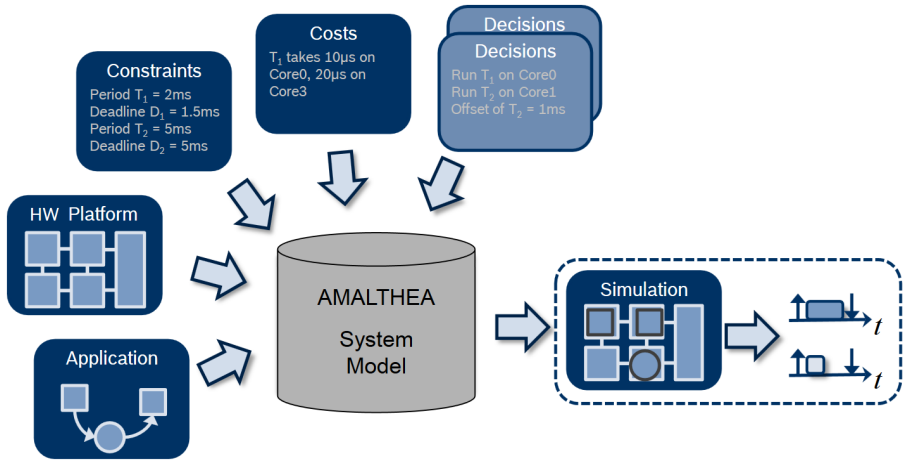


Figure 5: AMALTHEA Model for APP4MC [29]

improved by using the Tracing functionality of the APP4MC as well it can be improved by using several other tracing software.

- **Partitioning:** Partitioning stage in APP4MC-aided parallel design corresponds to identification of initial tasks. After an initial AMALTHEA model is constructed, the 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 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 Theory [79] which is commonly used in hardware and software co-design. Partitioning algorithms are used for analysis of the granularity and communication costs of individual runnables and create best possible parallelized 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 (with Simulation and Optimization):** 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), 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 are available. If desired, APP4MC generates tasks that are written in C language by using the AMALTHEA system model.

- **Tracing:** By making use of binary tracing, AMALTHEA trace model can be observed and can be 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 A4MCAR, a demonstrator RC-Car. Further sections will involve design, modeling, and evaluation of software distributions for this demonstrator.

4.1 Related Work

There are many studies done in the direction of efficient software parallelization methodology and tooling such as [80], [60], [75], [62], [63], [70], [56], [81], [68], and [72]. The works that involve APP4MC-based and AMALTHEA model-based parallelization are e.g. [80], [60], [75], [62], whereas some useful publications in the same direction that doesn't involve APP4MC or AMALTHEA can be given as [63], [70], [56], [81], [68], and [72].

The work by Carsten Wolff et al. [80] introduces the evolution of AMALTHEA tool project. 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

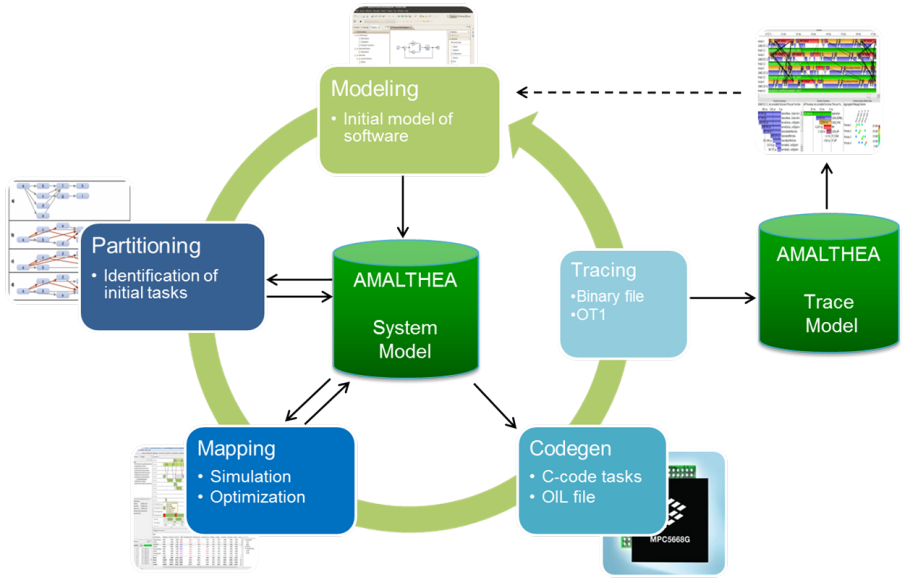


Figure 6: Illustration of how parallel software are designed using APP4MC platform [45]

AMALTHEA's close relevance to APP4MC project, 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. [60] 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 Integer Linear Programming 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. [75] 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, 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 exchange format of AMALTHEA 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 AUTOSAR compatibility of AMALTHEA within the project AMALTHEA4public and AMALTHEA being a relatively new but open-source supplementary tool to the automotive world is highlighted in the paper [75].

One work that the author of this report is also involved [62] 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 work, 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 [60]. 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 [62] 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 [62].

As mentioned, there are several other work that is in the same direction as AMALTHEA or APP4MC.

As an initial example, Devika et al. [63] explains the implementation of AUTOSAR Multicore Operating System. In their work, Devika et al. talks about real-time operating system OSEK/VDX, standards set by 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 reads could be given as the contributions done by Navet et al. [70] and Alfranseder et al. [56]. 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. [56] 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. [81], Lu et al. [68], and Nilakantan et al. [72] could also be nice reads.

5 Implementations on A4MCAR

5.1 A4MCAR Features

As A4MCAR targets automotive industry and parallelization studies done by 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. One could see the featured applications for the A4MCAR in Figure 7. In the figure, it is seen that the low level module of A4MCAR, built using xCore-200 eXplorerKIT targets

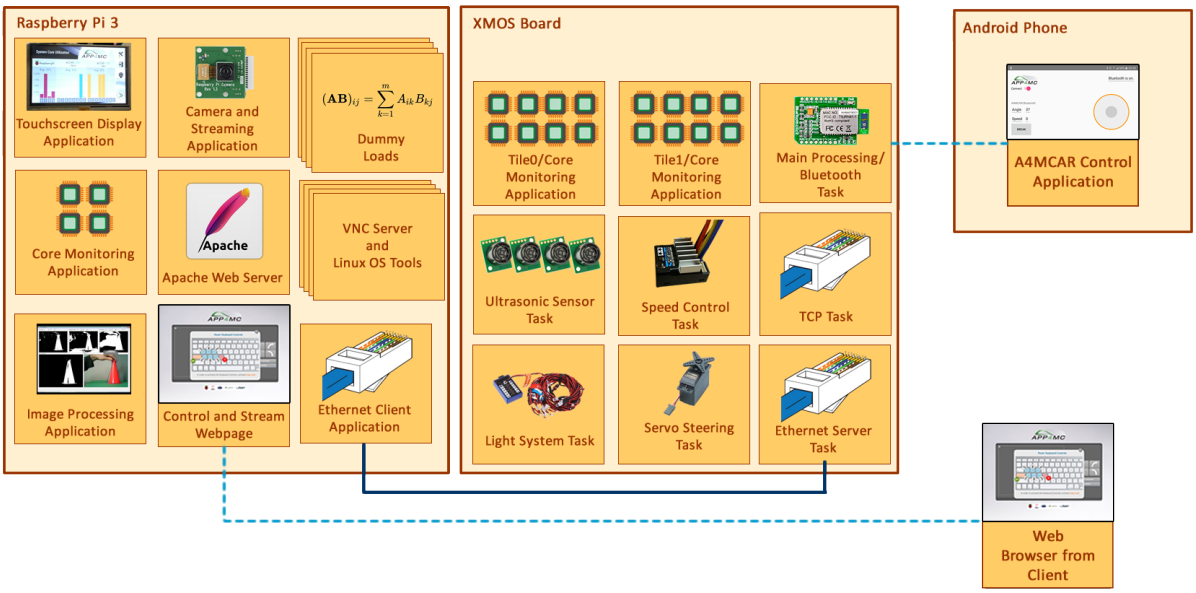


Figure 7: Applications developed and/or maintained for A4MCAR

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 usages.
- Bluetooth task to configure bluetooth module in slave mode and receive data over UART interface. v
- 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 drive speed controlling motor.
- Steering task that controls a servo motor using PWM signaling in order to steer the A4MCAR using external inputs.
- 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 high level module.
- TCP task and several other ethernet configuration related tasks to configure ethernet module (PHY) drivers and establish proper TCP connection.

In order to investigate parallelization outcomes on Real-time Linux and make use of high level features such as web server, image processing and touchscreen interface high-level module is introduced to the system. High-level module is designed so that it can communicate with low-level module over TCP in order to send driving information and retrieve core information from 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 7, full feature list can be given as follows:

- Core monitoring application that calculates the average core usages 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 writes 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 usages 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 high-level module, allocation of processes on 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 high-level module.
- Dummy load processes that perform random matrix multiplication in order to find performance indicators in full utilization.
- Several Linux processes that run Linux OS kernel and VNC server process that provides PC connection via SSH connection.

5.2 Low-level Infrastructure

XMOS xCore-200 eXplorerKIT features XEF216-512, a powerful multi-core microcontroller that provides sixteen 32-bit logical cores that are divided into tiles [41], which are identical units that contain processing unit, cache memory and switch mechanism [78]. XMOS xCore-200 eXplorerKIT contains two tiles with eight logical cores in each tile. It is important to add that logical cores of eXplorerKIT provides 2000 MIPS (Million Instructions per Second) processing power 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 GPIO, XMOS xCore-200 eXplorerKIT features 100/1000Mbps Ethernet module, high speed USB interface, a 3D accelerometer, a 3-axis gyroscope, and six servo interfaces which makes the kit useful in a wide variety of applications that include robotics, automotive, signal processing and communication applications [41].

As the name of the development kit suggests, XEF216-512 uses XMOS' xCore-200 architecture. An illustration of xCore-200 architecture is given in Figure 8.

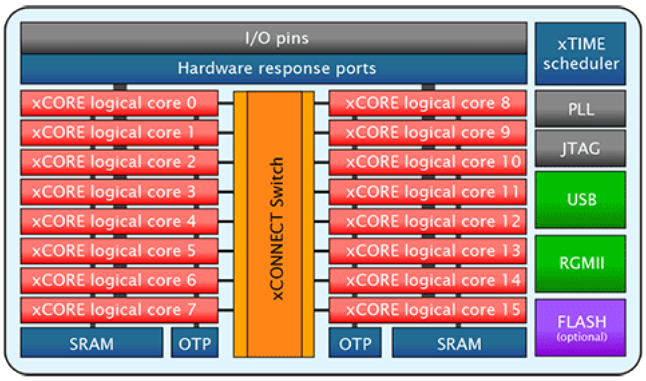


Figure 8: Illustration of XMOS' xCore-200 Architecture [54]

In 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 **xCONNECT Switch**. While this is the case for tasks that are distributed to seperate cores, for tasks that are placed in the same core **xTIME Scheduler** automatically schedules tasks by synchronizing events. The xTIME Scheduler works similar to RTOS in traditional microcontrollers and uses **Round-robin scheduling method** [55] [64] 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 [64].

In xCORE architecture, the synchronization of task communication is handled by events rather than ISRs (Interrupt Service Routines) as 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 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 [55].

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 9 [54]. 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 [54].

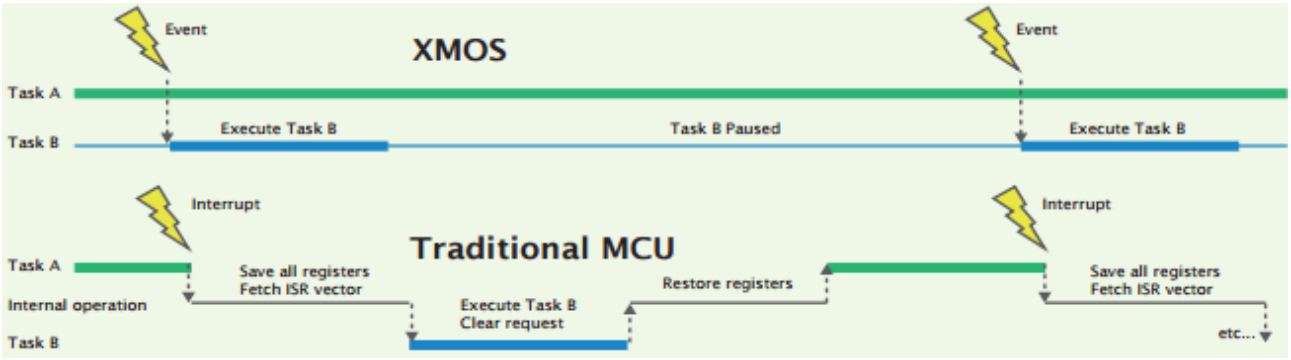


Figure 9: XMOS vs Traditional Microcontroller [54]

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 [59]. Via instruction pipelining, processors make use of the stages in order to use the clock to its full performance 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 10 [54].

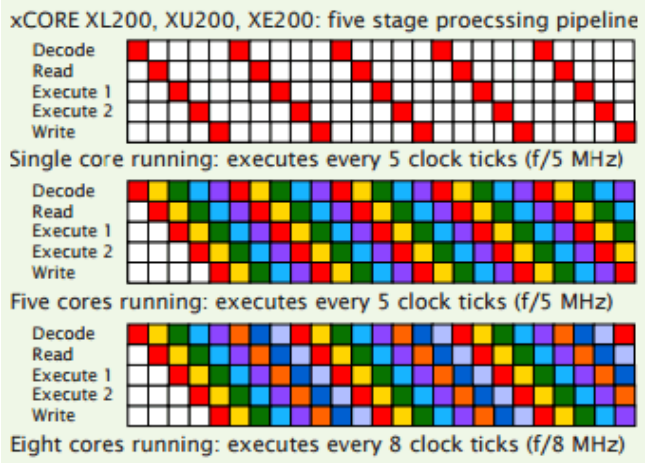


Figure 10: Pipelining Explained on XMOS [54]

Traditionally, XMOS based microcontrollers are programmed via 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**[54]. As A4MCAR needs to make use of timing and performance values, some tracing tools and XTA has been widely used during the development. xTIMEComposer development environment windows are shown and illustrated in Figure 11.

In the Figure 11 it is seen that the main development environment consists of the following windows:

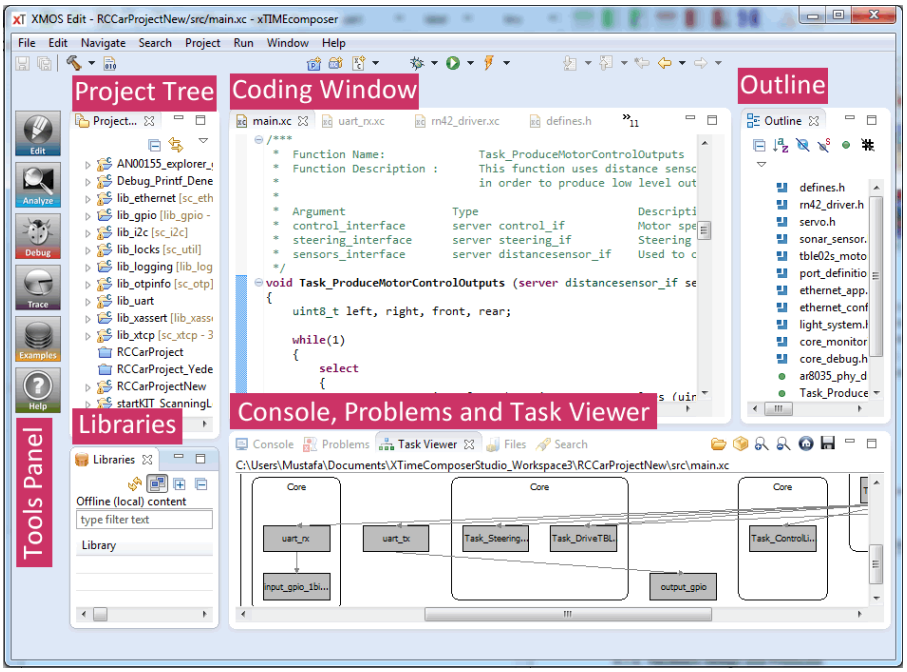


Figure 11: xTIMEComposer 14.2.3 Development Environment Windows

- **Project Tree:** This window is used for managing projects and source, include, binary and configuration files within projects.
- **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 it is 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:** 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) [55]. The aforementioned xC language features three main keywords in order to represent task communication. To represent an interface that sends data to another task, **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**. XMOS Programming Guide [39] 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 [39] 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 us develop event-based software. These features of xC make multi-core programming easy and robust on xCORE processors.

5.3 High-level Infrastructure

High-level processing unit of A4MCAR, Raspberry Pi 3, is a widely used single board computer in embedded applications. It has 1.2GHz 64-bit quad-core processor with ARMv8 architecture, 1GB of RAM, VideoCore IV

3D graphics core and several interfaces such as 40 GPIO pins, 4 USB ports, HDMI port, ethernet port, audio jack, camera interface (CSI), display interface (DSI), micro SD card slot [35]. 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.

Raspberry Pi 3 can be booted with modern Linux-based operating system distributions such as Debian-based Raspbian OS [34] and Ubuntu-based Ubuntu MATE[25] [35]. It should be noted that in A4MCAR, Raspbian OS has been used due to its wide software repository and driver support. The fact that Raspberry Pi 3 functions as a Linux computer helps in developing high-level applications that require operating system presence. The open-source nature of Linux and its software ecosystem provides flexible and traceable software development. In 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 in Raspberry Pi. In A4MCAR, programming languages such as C, C++, Python, Bash, HTML, JavaScript has been used in order to develop high-level module.

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, high-level overview of the structure of the **Linux kernel** and high-level layers in a Linux system which is given in Figure 12 should be considered [69].

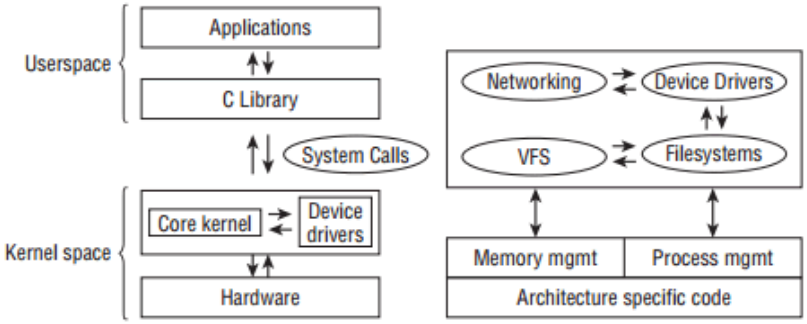


Figure 12: High-level Linux system architecture [69]

According to Mauerer [69], 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. Kernel is also the instance in a Linux system which shares available resources such as CPU, memory, and network; which is why it should be addressed while working with parallel applications.

In the figure, it is seen 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 run continuously is referred to as **processes** and they are scheduled by Linux kernel. The multi-tasking of processes are done by a mechanism that is called **task-switching** or **context-switching** and this is achieved to ensure that CPU performs according to the scheduled tasks. The concept of **scheduling** in a Linux system is also handled by 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 big parts of multi-tasking which are also handled by kernel. Threads share the same data and resources but they have different execution paths through program [69].

In A4MCAR high-level module, we mostly deal with processes and investigate ways to efficiently parallelize 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 [69] and Ward [77]. Figure in 13 depicts an illustration of how process life cycle works [69].

In Figure 13, 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 [69]:

- If the process is being currently executed, process is in **Running** state.
- If the process is not being executed because it is waiting for CPU to finish executing another process, it is in **Waiting** state.
- If the process is waiting for an external event such as a periodic activation or a sporadic activation, it is in **Sleeping** state. Notice that 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.

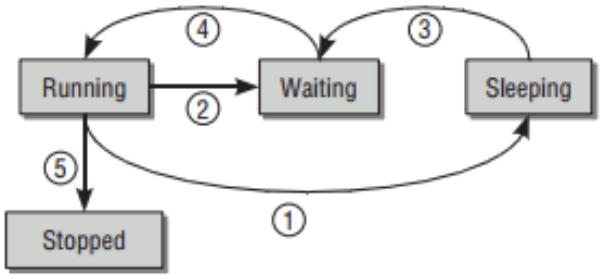


Figure 13: Process Life Cycle in a Linux System [69]

- 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, transition from Running state to Zombie state is also possible.

Raspberry Pi is conventionally programmed through Linux shell which is programmed and commanded with the help of 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. It is important to note that in the development of 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.

5.4 Low-level Module Implementation

Low-level module software has been implemented on 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 [58] diagram as shown in Figure 14.

In xC, two essential concepts are worthy to explain in order to understand multi tasked development. First is how 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 which is also some of what is implemented for A4MCAR are shown in Listing 1 and Listing 2, respectively.

```
1 [[combinable]]
2 void Task_GetRemoteCommandsViaBluetooth(client uart_tx_if uart_tx,
3                                         client uart_rx_if uart_rx,
4                                         client control_if control_interface,
5                                         client steering_if steering_interface,
6                                         server ethernet_to_cmdparser_if
                                           cmd_from_ethernet_to_override,
7                                         client lightstate_if
                                           lightstate_interface);
```

Listing 1: An Example Task Decleration in xC

In the code given with the Listing 1, it is seen that the function prototype has several arguments 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.

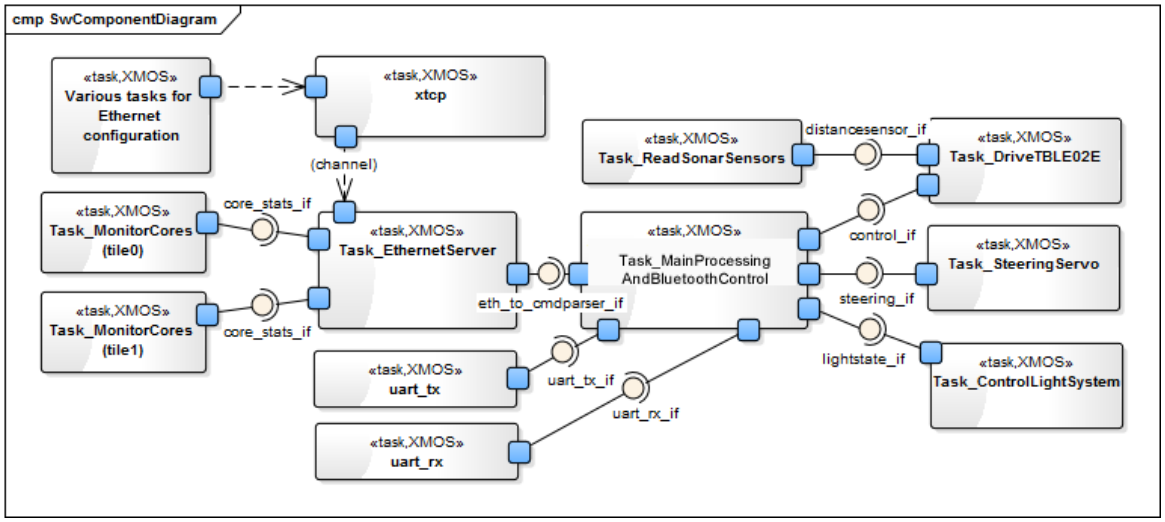


Figure 14: Brief block diagram for the developed tasks and interfaces for low-level module

```
1  par {
2       // I2C Task
3       on tile[0] : Task_MaintainI2CConnection(i2c_client_device_instances, 1,
         PortSCL, PortSDA, I2C_SPEED_KBITPERSEC);

5       // Motor Speed Controller (PWM) Tasks
6       on tile[0].core[4] :          Task_DriveTBLE02S_MotorController(
         PortMotorSpeedController, control_interface, sensors_interface);

8       // Steering Servo (PWM) Tasks
9       on tile[0].core[4] :          Task_SteeringServo_MotorController (
         PortSteeringServo, steering_interface);

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

The Listing in 2 shows in which tile and at which core a task will be places. 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 15.

5.5 High-Level Module Implementation

5.5.1 Overview

High-level module of the A4MCAR is composed of several processes and threads running under Raspbian [34] distribution of Linux Operating System that is designed for Raspberry Pi 3. During the compilation, debugging and execution of the developed processes, several development platforms such as Python 2.7 shell [7], GNU C Compiler (GCC) [11] have been used. Although it should be noted that remote development using Eclipse IDE [48] is also possible, the development of 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 [42] 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 [37] and jQuery [14] 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 16.

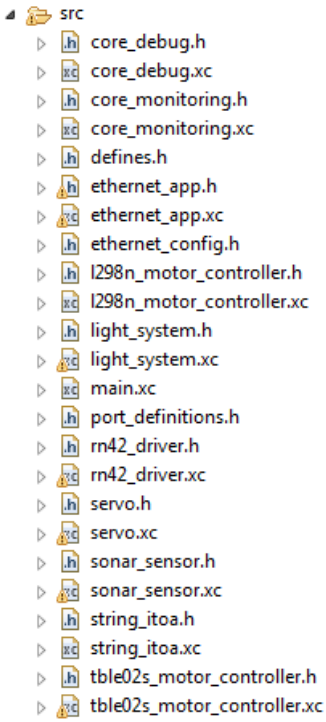


Figure 15: Full file tree for all the tasks developed for low-level module

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

In the Figure 16, it is also seen that the communication between user developed processes are handled with mostly file accesses. All file accesses are asynchronous and there is no event to wait for data or require data within some time as it is 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 the Figure in 16 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 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 should 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 [69]. However, it can be commented that the locking of the files are handled with the locks from OS kernel in our case and there is no need to create new locks in the applications for file accesses.

The way multi-tasking is handled with this constructed software architecture (in the Figure 16) is that every process is run by an external script at the 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 the Figure **??**. It is seen in the figure that interfaces such as HDMI, SPI, and CSI have been utilized. In the following sections, hardware communication and the related software will be further explained.

5.5.2 Implemented Online Timing Features and Making Processes Schedulable

In order to seek an assessment technique to compare timing performance of different distributions and 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 A4MCAR. Thus, while applications are running, a performance evaluation could be 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 are timed.

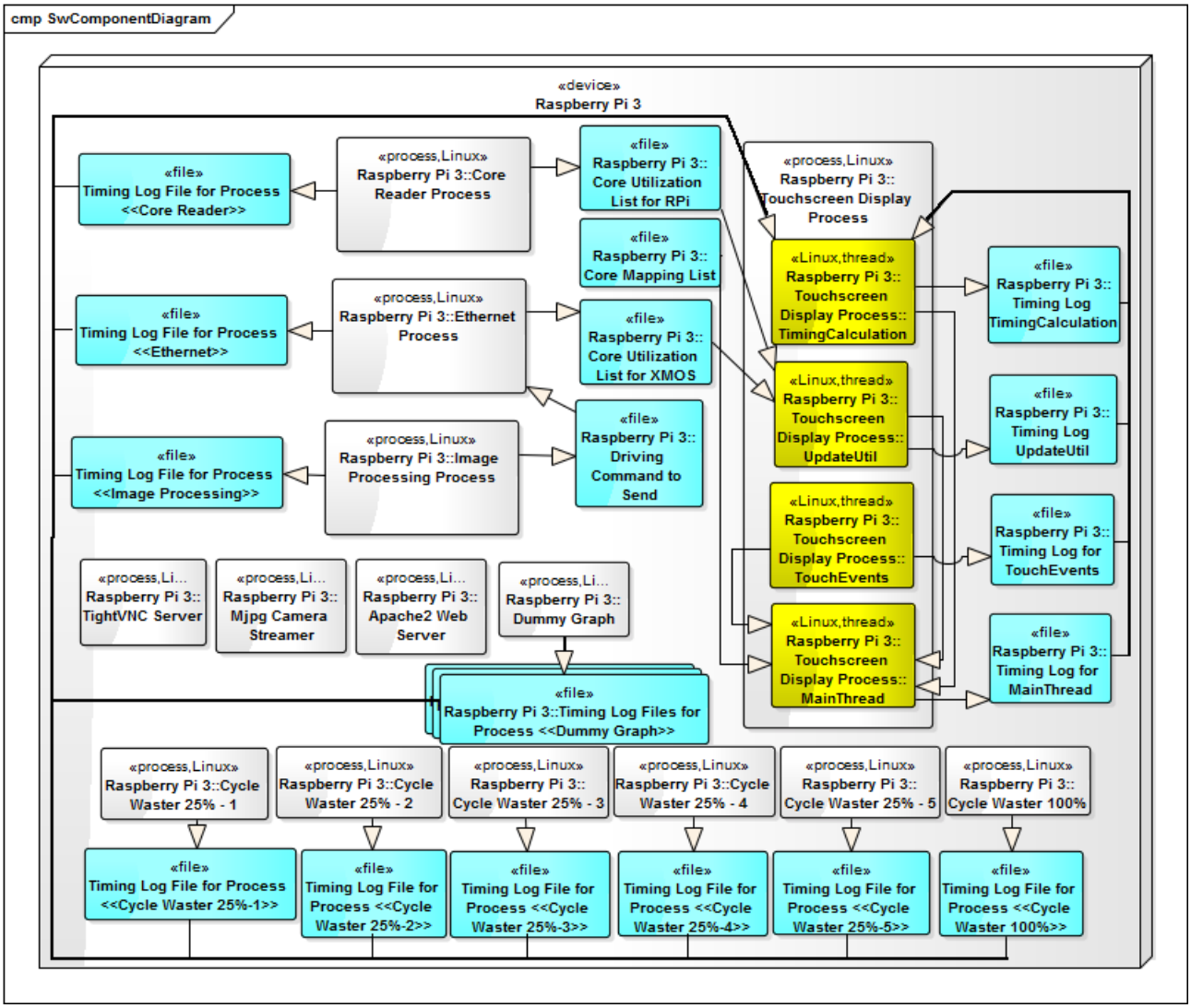


Figure 16: High-level module software component diagram including files and file accesses

Recall that in Figure 2 the timing properties in a scheduled system is explained. By making use of 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) are out of our reach since they are hidden in the Linux kernel, in the online timing analysis features that are implemented, the aforementioned values have been neglected. With offline scheduling analysis, however, CET values could be easily obtained.
- Recording the the start and end times of execution using an accurate clock. For that purpose, in computers there are two types of clocks: (1)- User CPU clock and (2)- System CPU clock. While 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 user CPU clock would not 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 [5] [23].
- Finding the execution time (ET) of one 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 seconds.

$$execution\_time = end\_time - start\_time \tag{2}$$

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

$$execution\_time = \frac{end\_time - start\_time}{CLOCKS\_PER\_SEC} \tag{3}$$

- Finding the slack time (ST) is one of the most important tasks that is within the scope of the online timing features because 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 that task is better utilized compared to before as the idle time the CPU is doing some other task is more than what it used to be. 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. 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.

$$previous\_slack\_time = start\_time - end\_time \tag{4}$$

- In order to keep a constant period while the process is being 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:

$$delay\_time = period - execution\_time \tag{5}$$

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

- As seen in 16, there are many timing log files that are created. Those timing log files are created within every user-defined process and later are used in the Touchscreen Display application.

While each processes and thread are constructed in the manner that is explained, the overall online evaluation is handled within the Touchscreen Display process in the TimingCalculation thread (shown in Figure 16).

An example of the timing skeleton for Python-running processes are given in the Listing 3. 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.

```
1  #!/usr/bin/env python
2  import psutil
3  import time
4  import string
5  import numpy

7  #Initialization
8  _DEADLINE = 1.40
9  _START_TIME = 0
10 _END_TIME = 0
11 _EXECUTION_TIME = 0
12 _PREV_SLACK_TIME = 0
13 _PERIOD = 1.40

15 def CreateTimingLog(filename):
16   global _START_TIME
17   global _DEADLINE
18   global _END_TIME
19   global _EXECUTION_TIME
20   global _PREV_SLACK_TIME
21   global _PERIOD

23   try:
24     file_obj = open(str(filename), "w+r")
25   except Exception as inst:
26     print inst
27   _END_TIME = time.time()
28   _EXECUTION_TIME = _END_TIME - _START_TIME
29   try:
30     file_obj.write(str(_PREV_SLACK_TIME)+' '+str(_EXECUTION_TIME)+' '+str(
       _PERIOD)+' '+str(_DEADLINE))
31     file_obj.close()
32   except Exception as inst:
33     print inst

35 while True:
36     #Timing Related
37   _START_TIME = time.time()
38   _PREV_SLACK_TIME = _START_TIME - _END_TIME

40   #####
41   #TASK CONTENT GOES HERE
42   #####

44   #Timing Related
45   CreateTimingLog("deadline_logger_burn_cycles_around25_1.inc")

47   #Sleep
48   if(_PERIOD>_EXECUTION_TIME):
49           time.sleep(_PERIOD - _EXECUTION_TIME)
```

Listing 3: Online timing features implemented in Python language

In the given code by the Listing 3, following remarks should 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 is 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 (4.5). This delay operation is also given in the Listing 3 at the Lines 48 through 49.

5.5.3 Core Reader

To help with the utilization assessment and visualization purposes, a core reading process is developed that monitors cores every three seconds and writes the core usage information to a text file. For that purpose, the 'psutil' module [20] 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 4. It should be noted that 'd' in the listing is an array with 4 elements, each of which indicating core usage for individual cores.

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

Listing 4: Psutil function to retrieve core utilization information

5.5.4 Ethernet (TCP) Client Implementation

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 Python language offers very stable and easy-to-use threading support and exception handling, the TCP client implementation is done using Python. The 'socket' library [9] in Python is capable of delivering several functions and objects that are used 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 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 by the Figure 17.

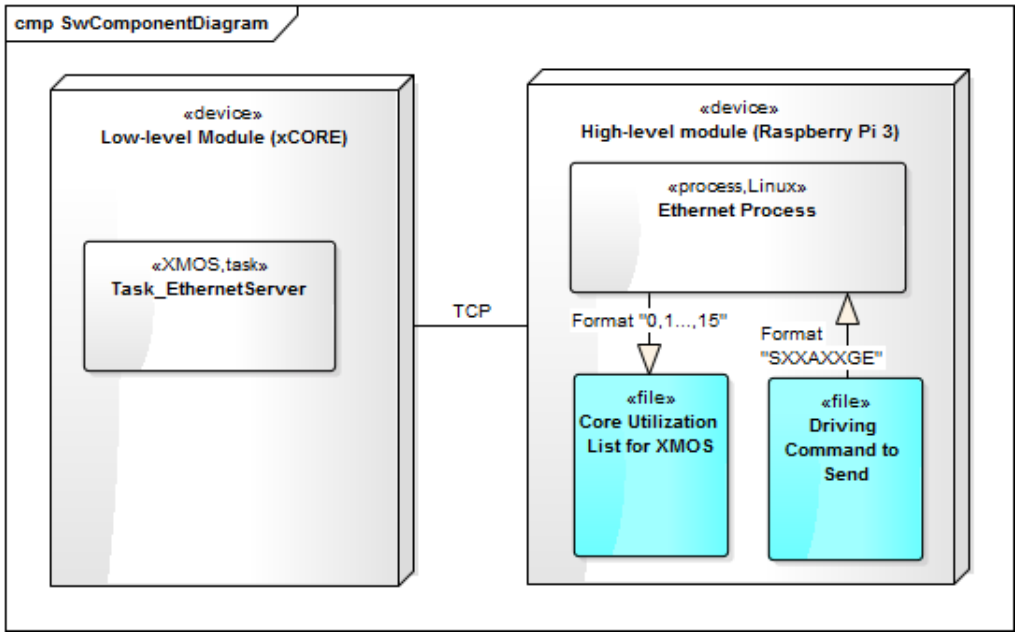


Figure 17: Deployment diagram showing Ethernet communication

5.5.5 Web Server

In order to develop a web interface for the A4MCAR, a web server is installed and configured to the high-level module. Web servers are responsible for processing HTTP requests and delivering HTTP responses [67]. The HTTP requests and responses are usually visualized using a web browser from clients in the form of web pages [67]. In A4MCAR high-level module, 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 in 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) [47] format unlike the TCP (Transmission Control Protocol) format. How web servers and web browsers work in order to help us visualize web pages is illustrated with the Figure 18 [47].

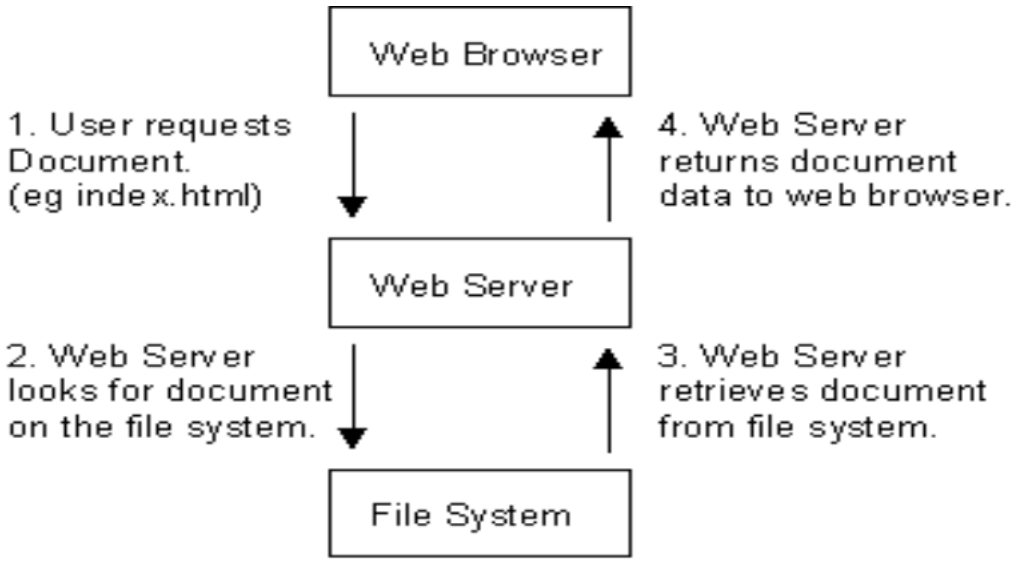


Figure 18: How web servers and web browsers work illustrated [47]

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 [14] 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 [37] 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, conditional events could be sent to server and processed. The returned data could be processed using JavaScript in order to dynamically update the page content [37]. This mentioned working principle is illustrated in the Figure 19 [37].

5.5.6 Web Page Design and Implementation

The web page that has been designed for users is shown in the Figure 20. In the web page, it is seen that 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.

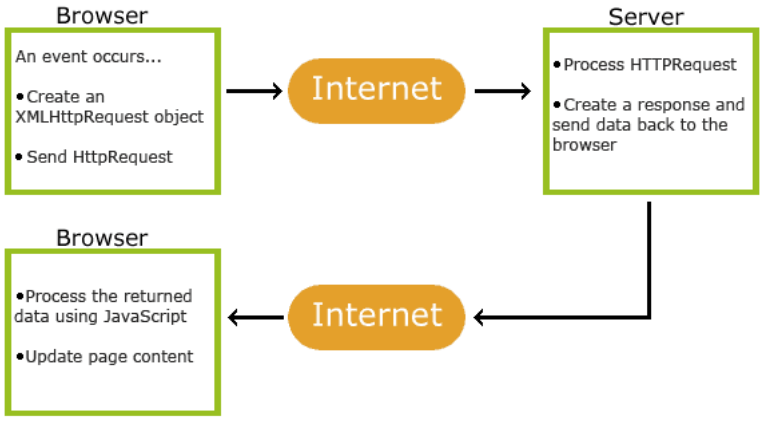


Figure 19: How AJAX works [37]

Figure 20: Web interface of A4MCAR

An alternative to this interface have 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 21

The overall dynamical behavior of the web page is illustrated in component diagram that is seen in the Figure 22. In the diagram, it should be noticed 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 A4MCAR, core utilization display, and camera streaming. In the following subsections, each of these tasks will be explained using the component diagram shown in the Figure 22.

5.5.7 Controlling A4MCAR via Web Page

At the top right of the Figure 20, the controls to drive the A4MCAR over web interface is shown. It is seen that there are gear selection buttons such as Forward (FWD) and Reverse (REV), along with two sliders. The

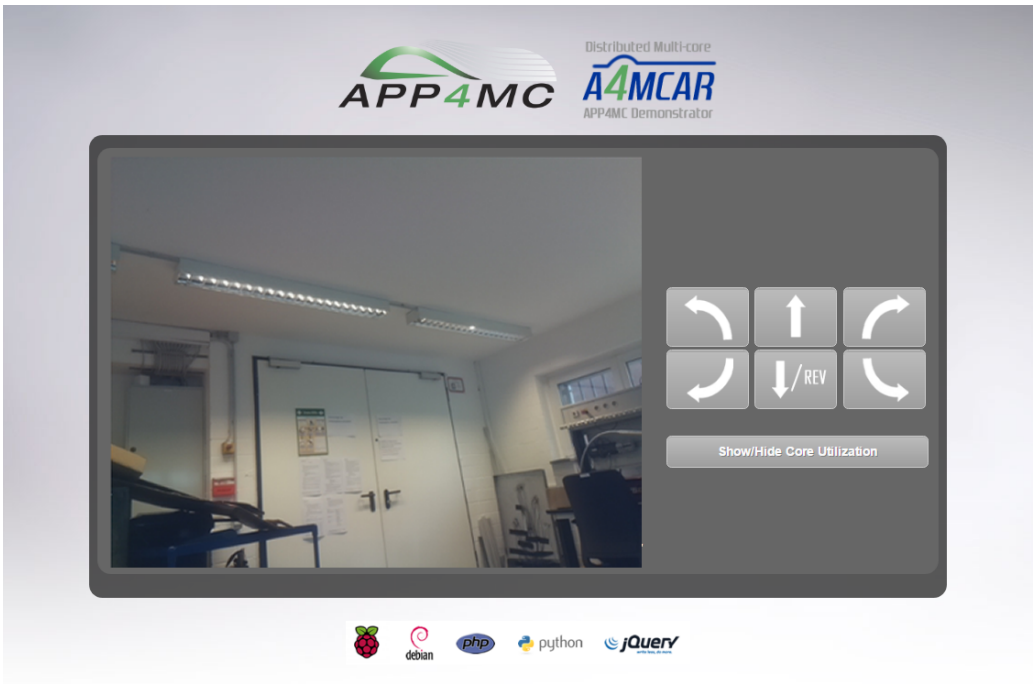


Figure 21: Alternative web interface of A4MCAR

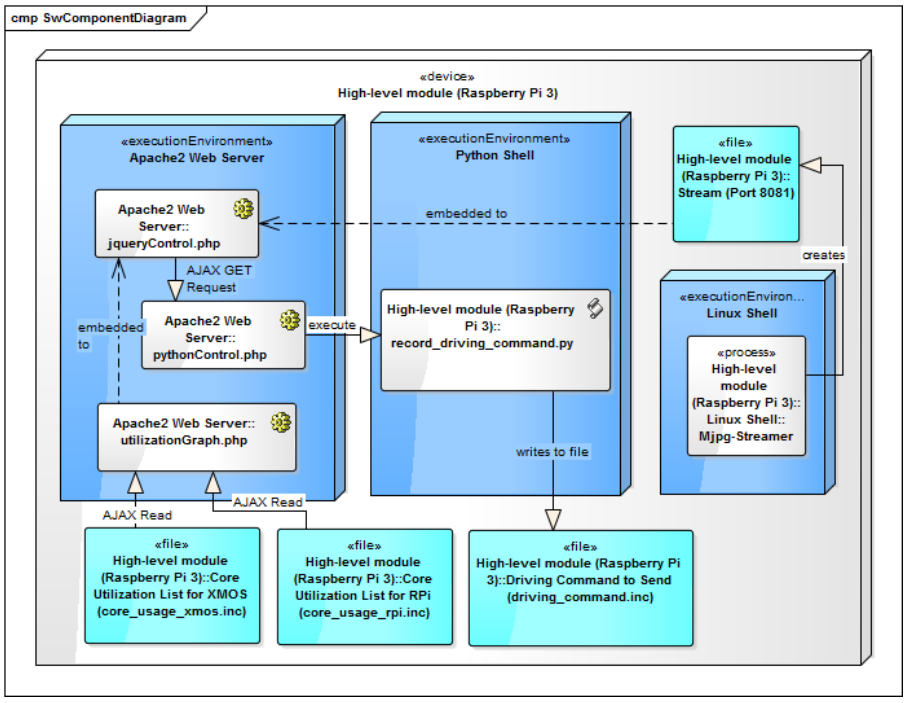


Figure 22: Component diagram showing how communication inside the created web-interface works

sliders are created using the third party script 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 could be used.

With the help of jQuery and AJAX' ability to create event handlers within 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 an HTTP "GET" request dynamically to another server page called pythonControl.php (shown in Figure 20). With the idea of demonstrating a basic AJAX request, an example of this is given in the Listing 5.

```
$.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 5: Sending dynamic HTTP GET requests using jQuery

As it is seen from the Listing 5 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 given in **??**. In the Figure 20, 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.

5.5.8 Camera Streaming

For the camera streaming, the third party module mjpg-streamer [13] has been used. This module is able to communicate over CSI interface in order to generate a stream on a network port, which then could be embedded to web pages. It is seen on the Figure 22 how this module works along with the Apache2 Web Server.

For the stream, Raspberry Pi camera version 2.0 with CSI interface is used which is shown in the Figure 23.

Figure 23: 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 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:

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

5.5.9 Core Utilization Display

Core utilization display is shown at the bottom of the Figure 20. 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 the Figure 22. 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' [49] 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.

5.5.10 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 use a certain percentage of the cores. Although the initial distribution does not involve dummy load processes due to increased responsiveness, the dummy loads is 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 load 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 higher than the previous option. This fluctuation is trivial in our application as most of the processes behave this way.

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

Listing 6: 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 6.

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 1 is a list of all dummy processes that are created in order to help with the utilization research:

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 1: Dummy load processes running in high-level module

Although the aforementioned processes have been created to stress the Raspberry Pi to find out the parallelization behavior in full utilization, several other purposes have also been considered. One very important distinction that seperates APP4MC's industrial use from A4MCAR is that industry has complicated tracing and distribution tools and standards such as AUTOSAR. Thus, in industry, fine-grained runnables can be created and distributed easily. However in A4MCAR, experiments showed that distributing and tracing such runnables are not so easy with real-world applications. This means that granularity of processes and threads might 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 industry application, a software graph that is called 'Dummy Graph' has been created using Python threads. The created software graph is illustrated in the Figure 24.

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, dummy graph has been completed. 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.

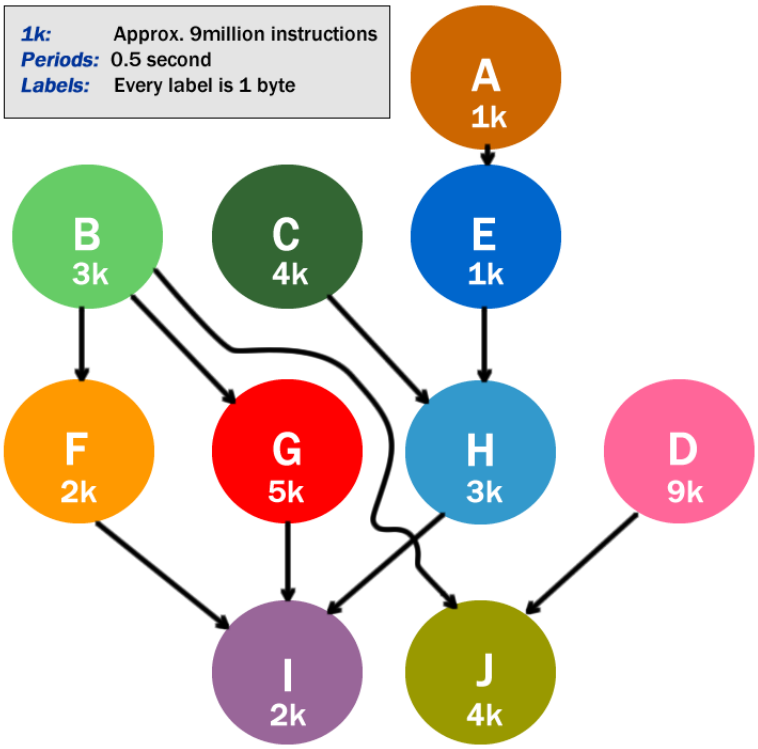


Figure 24: Dummy Graph that is created and its details

5.5.11 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 in A4MCAR.

For 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 [4]. The developed application makes use of 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. An example of detections are given in the Figure 25.

The applied transformations and how the traffic cone is detected as demonstration is illustrated with the Figure 26. The idea that is depicted is to retrieve contours of the possible objects and then determining whether it is a traffic cone or not by filtering those contours by their sizes and aspect ratios. In the figure, it is seen that several transformations are applied for this purpose. 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. 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 on their sizes and aspect ratios the desired objects are detected.

5.6 Touchscreen Display

5.6.1 Touchscreen Display Features

Touchscreen display that is embedded onto the Raspberry Pi 3 features several functions. It can not only show core utilization graph, average utilization percentages, timing performance, but also can be used to manage and allocate the processes of the high-level module. It also features connectivity settings in order to connect to an

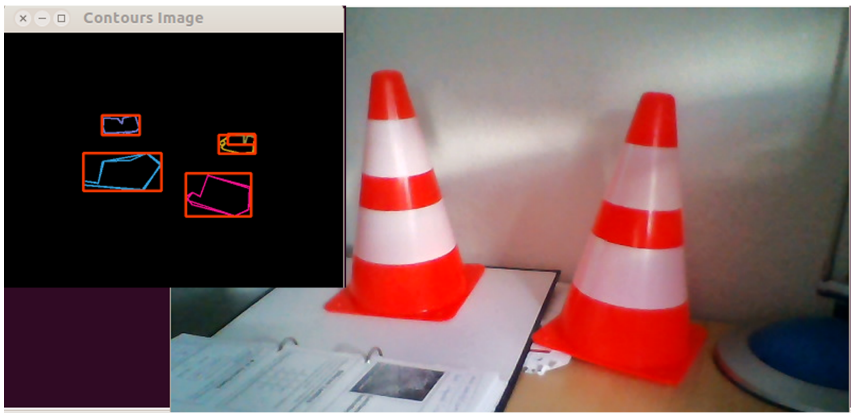


Figure 25: Developed Image Processing Application

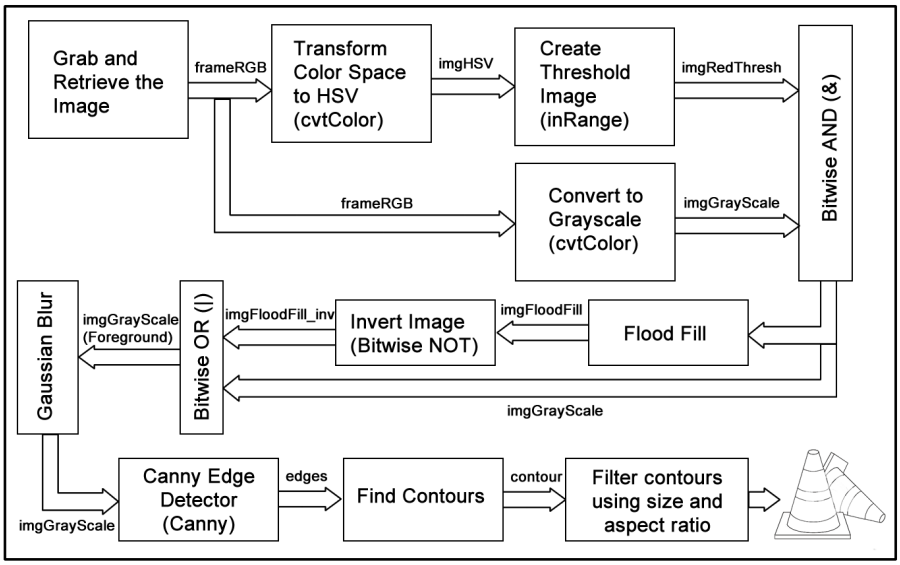


Figure 26: Applied Functions in OpenCV to Detect a Traffic Cone

access point. Main interface and buttons are shown in the Figure 27. Using those buttons users can switch between display modes that are mentioned, as well as go to the Settings menu, exit the touchscreen application, and shutdown the Raspberry Pi.

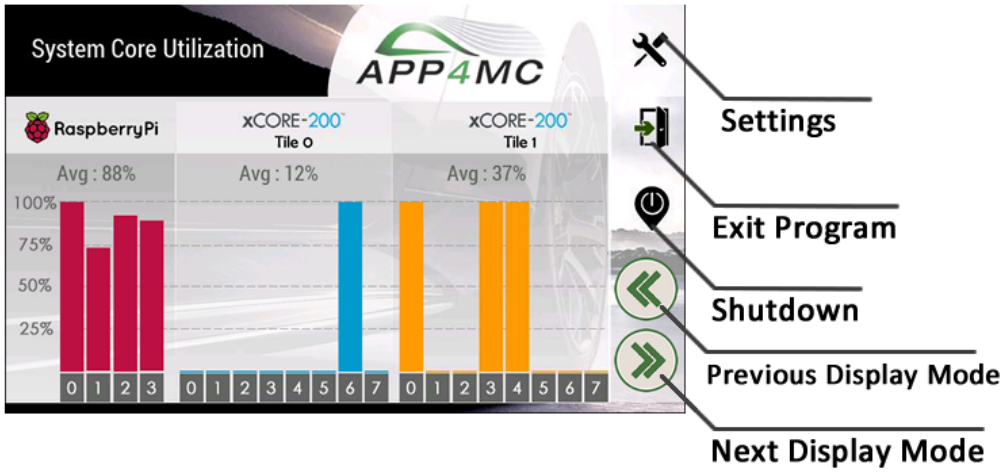


Figure 27: Button functions of A4MCAR Touchscreen Display

5.6.2 Touchscreen Display Implementation

As for the hardware, 5 inch HDMI touchscreen module from Waveshare has been used. This module can act as a primary monitor for Raspberry Pi. Addinitionally, the module features touchscreen controls using SPI pins of the Raspberry Pi GPIO. The module driver is installed and calibrated on 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 [33]. 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.

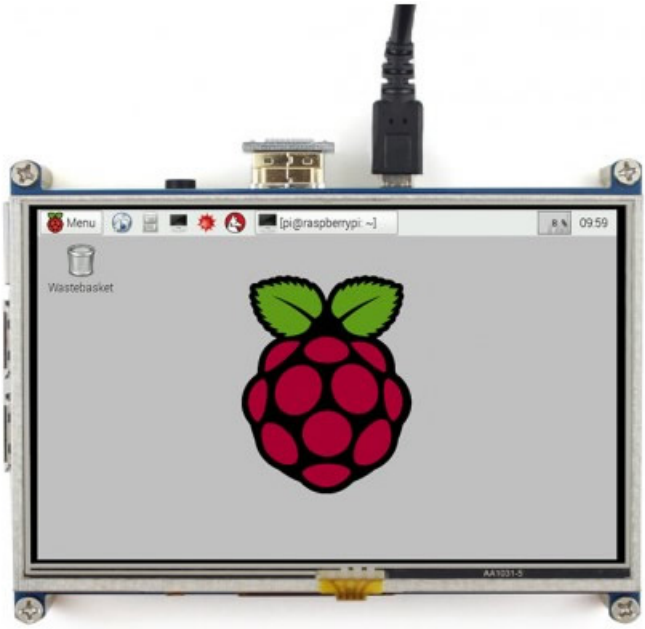


Figure 28: 5 inch Touchscreen module from Waveshare

Pages that are developed for the touchscreen module are shown in the Figure 30. In order to understand the behavioral operation of the touchscreen process, the Figure in 29 should be observed in parallel with the Figure 30.

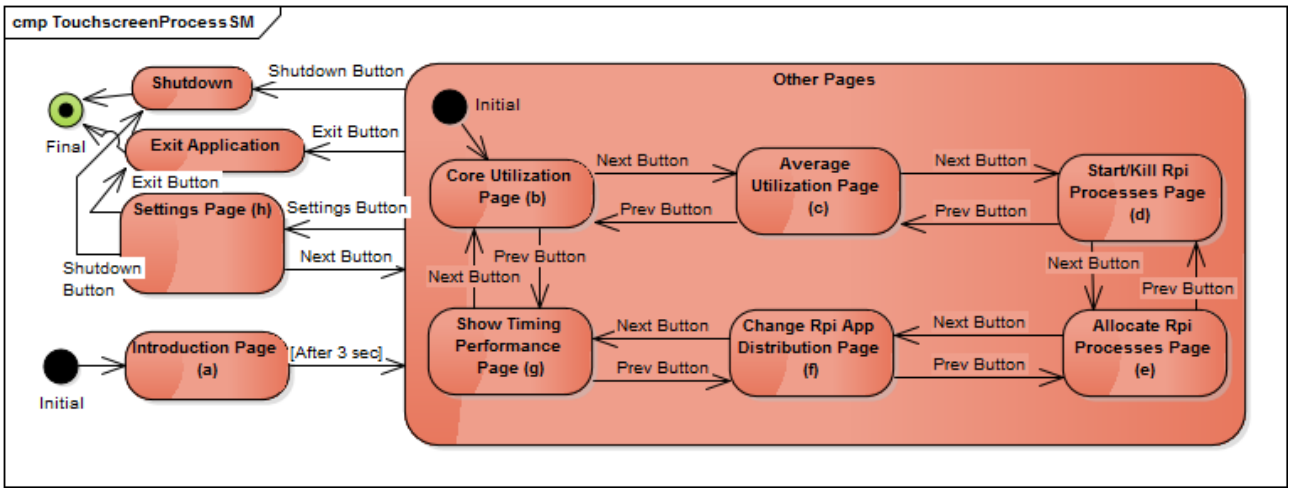


Figure 29: State machine of touchscreen process for pages as modes

The introduction page that is shown in Figure 30 (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 29) is navigated

Figure 30: Display modes from A4MCAR Touchscreen Display

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.

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 will be displayed and traced. The touchscreen display software is designed in the modular fashion that it automatically generates all the pages using the aprocess object list. 'aprocess' class have the following attributes and functions to make process handling easier in 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`)
- **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()`.

Touchscreen display process have a multi-threaded design. In this design Python's threading library [10] has been 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()' [10] of Python's threading library. The following listing shows how locking works with Python's threading library:

```
1 lock.acquire()
2 # Access to the shared variable
3 lock.release()
```

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()'.

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, 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 is resolved. The four threads that touchscreen display has 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 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 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 could show the results.

For the sake of informational completeness, how the schedulable Python threads are created should be depicted. Therefore, the Listing 7 should be explained.

```
1  def Thread_Name():
2    global aprocess_list
3    global aprocess_list_len
4    global SharedVariable2

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)

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 7: Thread skeleton in Python

In the Listing 7, a schedulable dummy thread is skeleton is shown. Between lines 2 through 4, the shared variables are defined. Lines 7 shows instantiating an 'aprocess' object by entering 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, 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.

Since the touchscreen display process is responsible from displaying many information, libraries to gather up such information have been used. Furthermore, the data that is gathered in the timing logs and 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 core mapping list from the high-level module. There is also ability to change the core frequency (Figure 30 (g)), and display gross execution time.

5.6.3 VNC Server

Virtual Network Computing (VNC) is a system that allows creating and managing virtual computers as well as connecting to them remotely [74]. While dealing with programming single board computers such as Raspberry Pi, it 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 it without having to use external hardware. The VNC server that is installed in 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.

5.7 Android Application Implementation

To control the A4MCAR remotely via communicating with the RN42 bluetooth module that is connected to the low-level module, A4MCAR control application is developed using Android [27] environment. In the Figure 7, one can see how developed A4MCAR control application interacts with the entire software that is developed for A4MCAR.

As an integrated Android development environment, Android Studio [6] is used. Using Android Studio, developers can not only design XML-based user interfaces for their applications, but also can describe the

behavior of their programs using Java programming language. Additionally, Android Studio can emulate many of the available Android devices to help the developers debug their software.

The developed Android application interface is given in Figure 31. In the figure, it is seen that the interface consists of a joystick and gear buttons that helps in constructing the driving command. Furthermore, using a bluetooth device list, A4MCAR could be paired with and then connected in order to start data communication.

For the joystick controls, a third-party Android library that is called virtual-joystick [12] 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 created which is a callback function which acts every time the joystick is moved. Using this callback function, the angle and strength information that results from the joystick has been transformed to conform to the driving command (Figure **??**). With the help of the Figure 32, this data transformation can be explained easily.

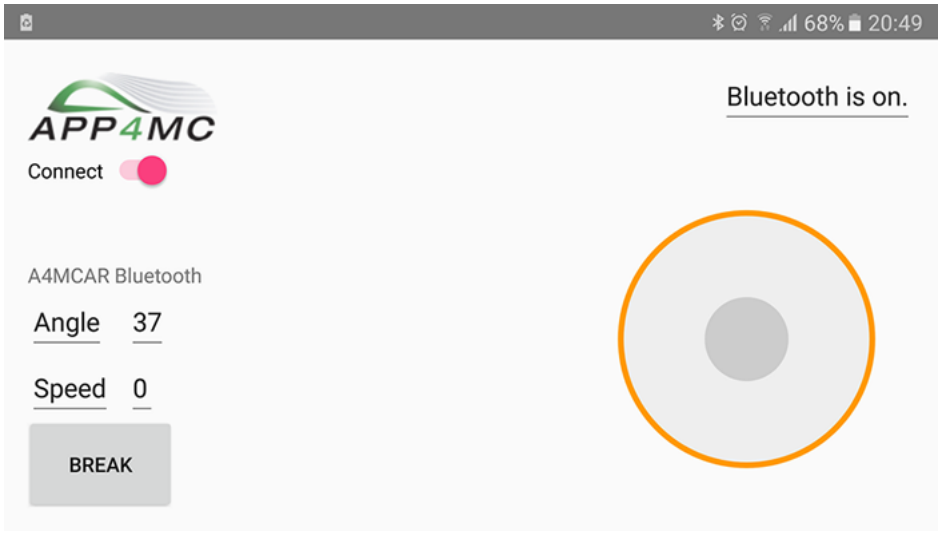


Figure 31: Android Application Developed for Driving A4MCAR Remotely

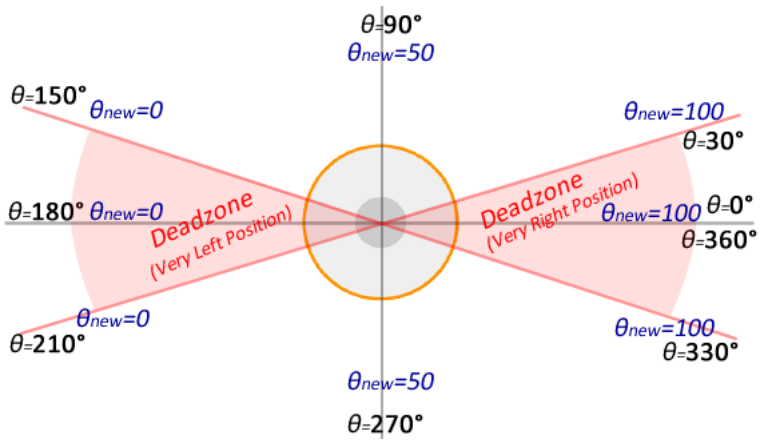


Figure 32: Joystick angle transformation to construct driving command

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 in the low-level module side.

As it is clearly shown in the Figure 32, 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:

$$\theta_{\text{new}} = \left(1 - \frac{\theta - 30}{120}\right) 100 \tag{6}$$

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

$$\theta_{\text{new}} = \left(\frac{\theta - 210}{120}\right) 100 \tag{7}$$

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.

6 Exploring Tracing, Mapping, and Energy Consumption Features

6.1 Introduction

After the distributed multi-core system is developed and defined, the evaluations regarding different software distributions are needed 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, the textbook methodology of extracting useful information from a software is to analyze 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 mostly used:

- **Static Binary Analysis:** Static binary analysis is a reverse engineering methodology that helps in finding errors in code such as the errors that involve non-determinism [28]. It is essentially analyzing 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 [28]. It is important to keep in mind that in the static binary analysis, the program is not executed [71]. Therefore, the information regarding execution and timing are not provided while the instruction information could be extracted. 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 could be observed. With the help of the binary analyzer tools, detailed information on number of function calls, nesting, and cyclic complexity could be observed [51].

- **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 [71]. 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 [51]. 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 [51]. Using dynamic binary instrumentation (DBI) tools for Linux platform this way, one can obtain information such as CPU time, execution times, memory and I/O of a program [71].

- **Tracing:** Tracing is a methodology which is often mixed with profiling. According to IPM [2], 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 are 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 said 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 [53] is done within the entire system and it is used for obtaining useful information regarding the system performance as a whole. Operating systems (especially Linux platform) usually have system logs which could be observed via system monitoring in order to extract useful information regarding the system performance. Speaking for A4MCAR, the performance values such as core frequency and core utilization information are extracted using system monitoring.

Low-level and high-level modules of A4MCAR requires several information. To start with, in order to model the software system of both modules with A4MCAR, the number of instructions, task iteration 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 XTA tool can estimate instructions, timing, and path from 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 option for easement of the process. Secondly, system monitoring is needed in both of the 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 in high-level module. System monitoring in

A4MCAR is essentially needed in order to obtain core utilization percentages, live CPU frequency 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 are obtained from profiling and tracing involve slack time, execution times, start and end times.

In this chapter, the aforementioned techniques for system analysis will be 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 evaluation of various software distributions.

6.2 Low-Level Module Information Tracing and System Management

6.2.1 Static Binary Analysis via XTA

XMOS Timing Analyzer (XTA) [40] is an Eclipse-based tool that comes with xTIMEcomposer platform which is used for analyzing the timing and the execution details of the multi-tasked software that are developed using XMOS boards and processors [40]. 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 33.

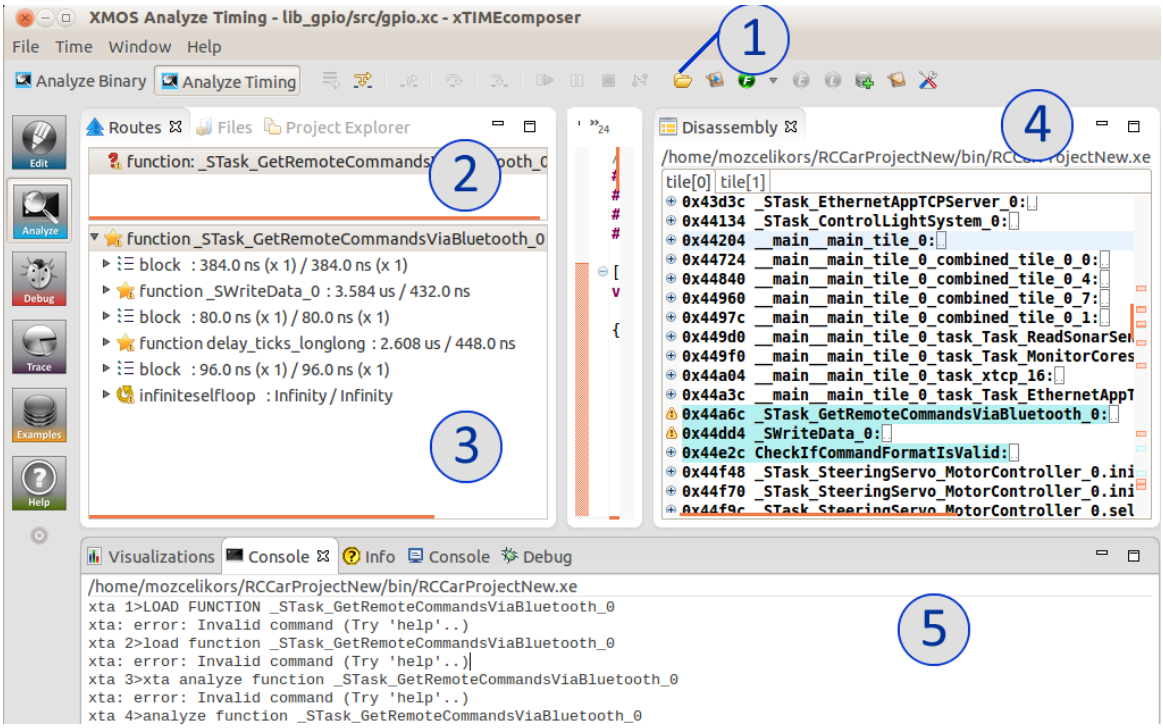


Figure 33: XMOS Timing Analyzer (XTA) screenshot

Once the code is written and built in xTIMEcomposer platform, a binary file is generated with the extension .XE. By loading this binary file using XTA, the timing analysis could be performed. In order to load the binary to XTA, "Load Binary" button (shown as 1 in the Figure 33) should 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 in runnables could be seen.

XTA tool is also able to work with specific commands by using the XTA console (shown as 5). Using the specific commands taken from XTA manual [40], timing analysis could be done easily. In order to start timing analysis, an execution path should be defined. The execution path could be determined analyzed using the following possible ways [40]:

- Via placing end points into the code using compiler directive #pragma as shown in the Listing 8.

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

Listing 8: Placing end points in xC code to define execution path

The timing analysis between two endpoints could be started by entering the following command to the XTA console:

```
1 analyze endpoints start_endpoint1 stop_endpoint1
```

- A function or a runnable with the name "Function_Name" could be analyzed from its starting point to its return point by using the following command:

```
1 analyze function Function_Name
```

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

```
1 analyze loop looppoint
```

The entire code could 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 matched [40]. Once the timing analysis starts, the selected route, i.e. a function, a loop, or a route between two endpoints are shown in the Routes window (shown as 2 in the Figure 33). The selected route is shown in blocks in another window which is shown as 3 in the figure. Here, it is seen 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 best case execution time is estimated to be 3.584 microseconds, whereas the worst case execution time is 432 nanoseconds. Further information that is provided by XTA regarding timing analysis is given in the Figure 34. As it is seen in the 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, and number of paths are also shown.

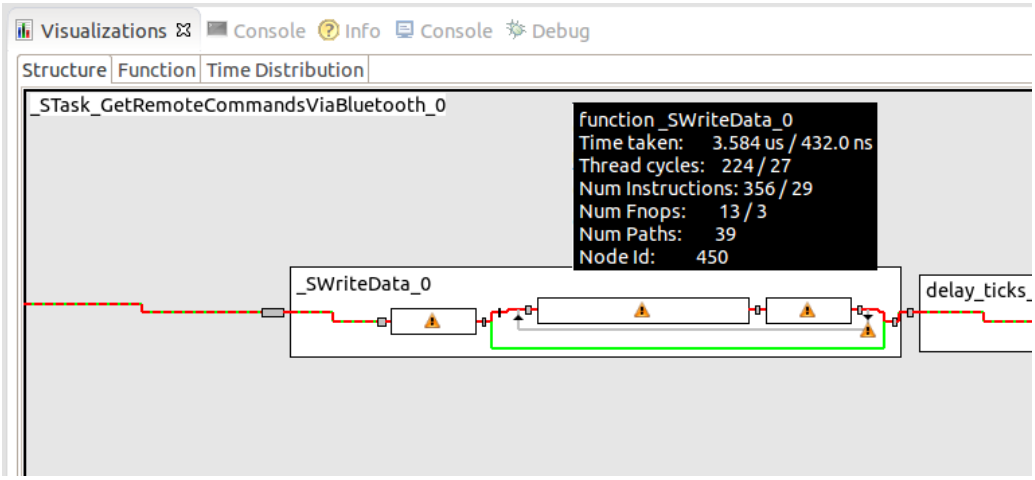


Figure 34: XTA Visualizations window with further information

For the software that is developed for the low-level module of the A4MCAR, 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-determisim 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, following techniques are used:

- The "Unresolved" error occurs usually when XTA can not resolve a branch instruction whose branch target is unknown [40]. In order to get rid of this issue, trace of the system should be printed and branch instruction target should be pointed manually. The details of how this is done is given in the XTA manual [40]. 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 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 partitioning and mapping accuracy would not change drastically if the gathered information is not exact.

6.2.2 Distribution of Tasks to Cores

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

In xTIMEcomposer platform, 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. In XMOS, placement of a function into a core is done by using "par" statement. In the main block of a software, "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 to the Listing 2, a simple example would help one better understand 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 }
```

The given code is a basic example of using xC functionality to do a task mapping. The parallel block in the code could be given as the Lines 4 through 8. It is seen 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 the 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 tiles and cores. Additionally, it should be noted that unlike 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 [39].

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

6.2.3 System Monitoring in xCORE

Besides the implemented core monitoring task, XMOS features several more means to monitor the system. First, system could be monitored at build-time using the XTA tool Binary window, Resource Usage tab. In the Figure given in 35, it is seen that the information regarding stack memory, program memory, free memory, cores, timers, and channels could be observed in one window using XTA Binary. Another mean 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 your 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 so much overhead [39].

Using the "debug_printf" function, xC-specific tools, a function has been 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 following listing, this information could be monitored at run-time:

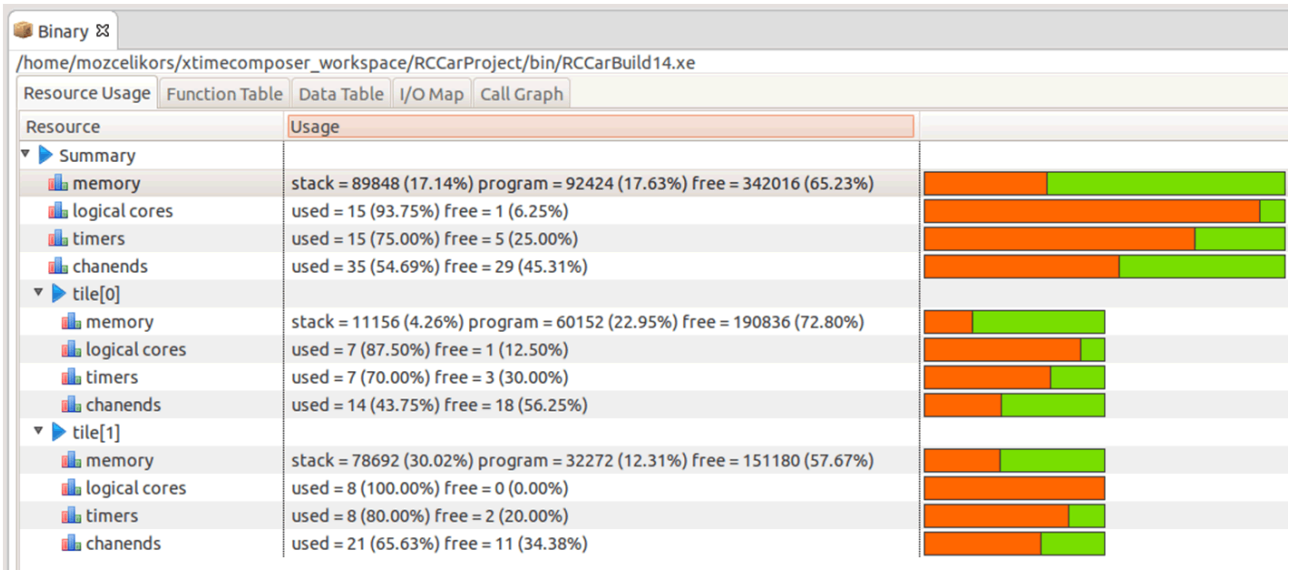


Figure 35: XTA Binary Resource Usage

```
1 int PrintCoreAndTileInformation(char * Function_Name)
2 {
3 debug_printf("Starting %s task on core ID %d on tile %x\n", Function_Name,
   get_logical_core_id(), get_local_tile_id());
4 return 1;
5 }
```

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, we make 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 could be done easily by observing the Console. An example output of the Console from final version of the low-level module software is given with the Figure 36. It is seen 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.

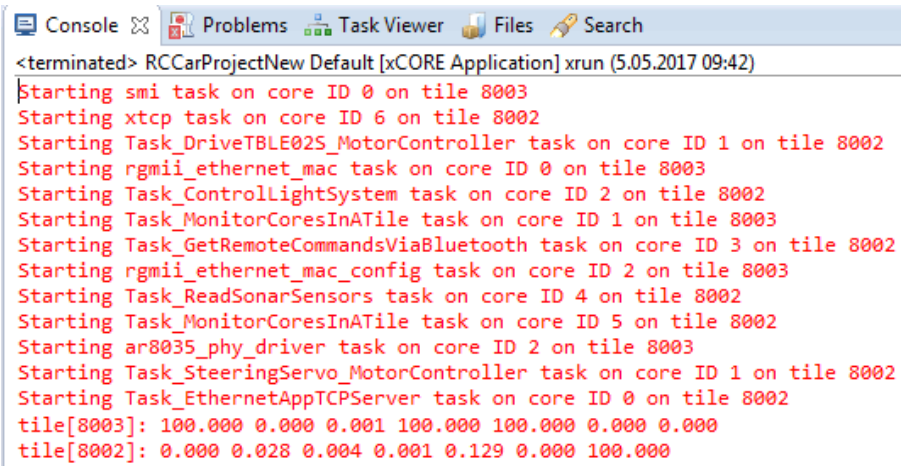


Figure 36: System monitoring implemented on xCORE

6.2.4 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 [38] should be studied well.

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 [73]. 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 our research. The dynamic power consumption of a chip is described by the following equation [76]:

$$P_{\text{dynamic}} = \alpha C_{\text{L}} {V_{\text{DD}}}^2 f_{\text{clk}} \tag{8}$$

We can see from the equation that the power depends on switching probability α, chip voltage V_{DD}, clock frequency f_{clk} and collective switching capacitance C_{L}. It is stated in the [73] that since V_{DD} linearly depends on the clock frequency f_{clk}, a cubic relationship between power consumption and clock frequency could be observed. The linear relationship between the current I_{DD} and the frequency is also depicted in the Figure 37, showing the performance values for a XS1-L device of XMOS xCORE-200 eXplorerKIT [38].

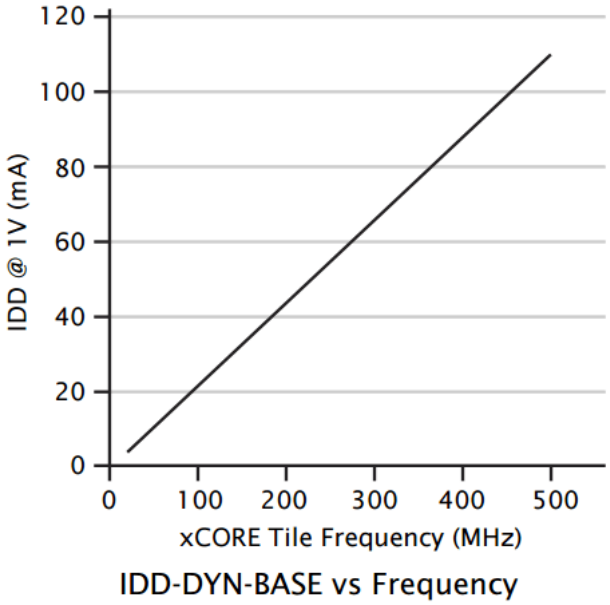


Figure 37: XS-1 Power Graph Related to Base Current for an xCORE Core

In the figure, we see that the base power consumption of one xCORE running an instruction sequence is given with respect to the clock frequency. By looking at the figure, we 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. $P = I_{\text{DD}} V_{\text{DD}}$, we can conclude 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 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 there are no deadline misses [76]. Using provided internal registers, xCORE core and tile frequencies can be reduced according to the manual [38].

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 the power consumption [38]. Therefore, two techniques with which power and energy can be reduced, can be defined. 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. In the following sections, these techniques will be referenced.

An important energy related feature that comes with XS-1 devices is the AEC (Active Energy Consumption) mode [38] 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 [38]. This could be slightly related to DVFS (Dynamic Voltage and Frequency Scaling) [57] which is a processors ability to dynamically scale its frequency and operating voltage depending on the load. DVFS is common for Linux OS running computers and it will be discussed in the next section.

6.3 High-Level Module Information Tracing and System Management

6.3.1 Binary Analysis of Instructions

In order to obtain 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 in A4MCAR in order to model the system using APP4MC.

As a static binary analysis solution, "objdump" [21] module of Linux provides disassembly information of C-based libraries and executables. After the compilation of a C/C++ program, 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 could be analyzed easily. Objdump provides not only the disassembled instruction list but also can provide information such as symbol tables and debug information. In order to record the instructions of a C/C++ application, one of the following Linux shell commands are used:

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

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" [8] 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 below:

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

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 following in a Python code:

```
1 import dis
2 dis.dis(FunctionName)
```

Perf [18] [46] 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 [18]. In A4MCAR, Perf profiler tool has been used for many purposes that include process instruction analysis, system-wide scheduling trace, 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 following listing. It should 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 should 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 process ID of a process is obtained and what it means is discussed in the next section.

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

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 the following listing, ¡timeout¿ is the period in seconds.

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

For threads, the aforementioned method is error-prone, therefore using –**per-thread** switch is more reliable when profiling threads of a process.

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

6.3.2 Process Management and Monitoring

Managing and monitoring processes and threads of a 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 [36]. By using the following, processes of the entire system could be monitored.

```
1  top
```

```
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 38: Top command output

It is seen in the Figure 38 that 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) could be monitored in this window. By using the following command, one could also see the threads of a process given its process ID:

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

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 given below in order to list processes and threads:

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

- **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 following command, the PID of a process can be obtained by the process name:

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

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 the Listing 9

```
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 9: Created Bash script to dynamically profile applications (AppMonitor.sh)

By calling bash AppMonitor.sh ¡process_name¿ from Linux shell, this script could be used to monitor applications using perf. In the script, it should be noted that command 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 10: Function to obtain process ID from Python environment

Since touchscreen display process is responsible of doing all the online timing calculations, a function has been written which returns the PID of a process if the process is running (Listing 10). 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 10).

- **Killing a Process:** Killing a process is handled through a simple Linux shell command that is given below:

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

Using the same manner that is explained by the Listing 9, 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 Linux kernel folders:** Linux kernel provides a virtual filesystem that are located under /proc directory that contain runtime system information for system, device, connectivity and process monitoring [52]. Regarding process monitoring, using the process ID as the folder name and simply viewing the files that are located under /proc/¡pid¿/ many information such as process status (observed in Figure 13), memory maps, libraries and executables, executed cpu, scheduling information can be monitored. A few examples are given in the Listing 11 with their explanations [52].

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

Listing 11: Kernel virtual filesystem /proc information retrieval examples [52]

6.3.3 System Monitoring for Linux Platform

As mentioned, for system monitoring /proc folder is also widely used. The information that are stored in the virtual filesystem involve the following information [52]:

- 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 with the issues of Linux system administration those information are useful, a more easy way to obtain certain information could be done through using third party modules. In 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 following function, the core frequencies are extracted in MHz:

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

Similarly, psutil.cpu_count() function can be used to extract the number of active cores and psutil.cpu_percent() function can be used to extract the core utilization percentages in a given time period [20].

6.3.4 Tracing the System to Obtain Scheduling Information

To evaluate performance indicators and observe load balance, tracing the high-level module processes is crucial in 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 should be 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 [46]. As an example, the system trace could be obtained for 15 seconds by entering the following command into the Linux shell:

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

  Once the tracing is done, system trace is recorded to a file called "perf.data". This trace file uses 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 following command could be used which saves the full trace in a text file called fulldump.txt:

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

  A few 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 12, each line represents a kernel event. If a task is being started to execute on a core that event is referred to as 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.

  ```
  1 perf 16984 [005] 991962.879960: sched:sched_stat_runtime: comm=perf pid
     =16984 runtime=3901506 [ns] vruntime=165...
  2 perf 16984 [005] 991962.879966:       sched:sched_wakeup: comm=perf pid
     =16999 prio=120 target_cpu=005
  3 perf 16984 [005] 991962.879971:       sched:sched_switch: prev_comm=perf
     prev_pid=16984 prev_prio=120 prev_stat...
  4 perf 16999 [005] 991962.880058: sched:sched_stat_runtime: comm=perf pid
     =16999 runtime=98309 [ns] vruntime=16405...
  5 ....
  ```

  Listing 12: Perf sched script command output [46]

```
               *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 13: Perf sched map command output [46]

Since 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 following command could be used in Linux shell:

```
sudo perf sched map
```

The output of this command can be seen in the Listing 13. In the code, it is seen that each column represents a core whereas vertical axis can be thought of 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 asterisk 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.

Speaking for the example in Listing 13, if the cores were to have names 0 through 3 depending on their column index, it should 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 should be 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 [16] is yet another tool that records system trace. Trace-cmd trace could be generated into a file trace.dat by using the following command:

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

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 39. Although kernelshark along with 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 [43] is an Eclipse-based open-source platform that is used for providing views, graphs and metrics for many type of logs and traces [43] The graphical user interface of the Eclipse TraceCompass is shown in Figure 40. Since the information that is provided by Eclipse TraceCompass is more user-friendly and more detailed than the other visualization options that are discussed, in A4MCAR Eclipse TraceCompass has been used for watching the high-level module system trace.

  LTTng [3] is an open-source tracing framework that is provided for 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 watched 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 [1] with tracing options enabled and also with a tracing library that is called LibBabelTrace. The detailed information regarding this building process could be seen in [24]. Once the perf is built and installed with babeltrace, one can use the following commands 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
```

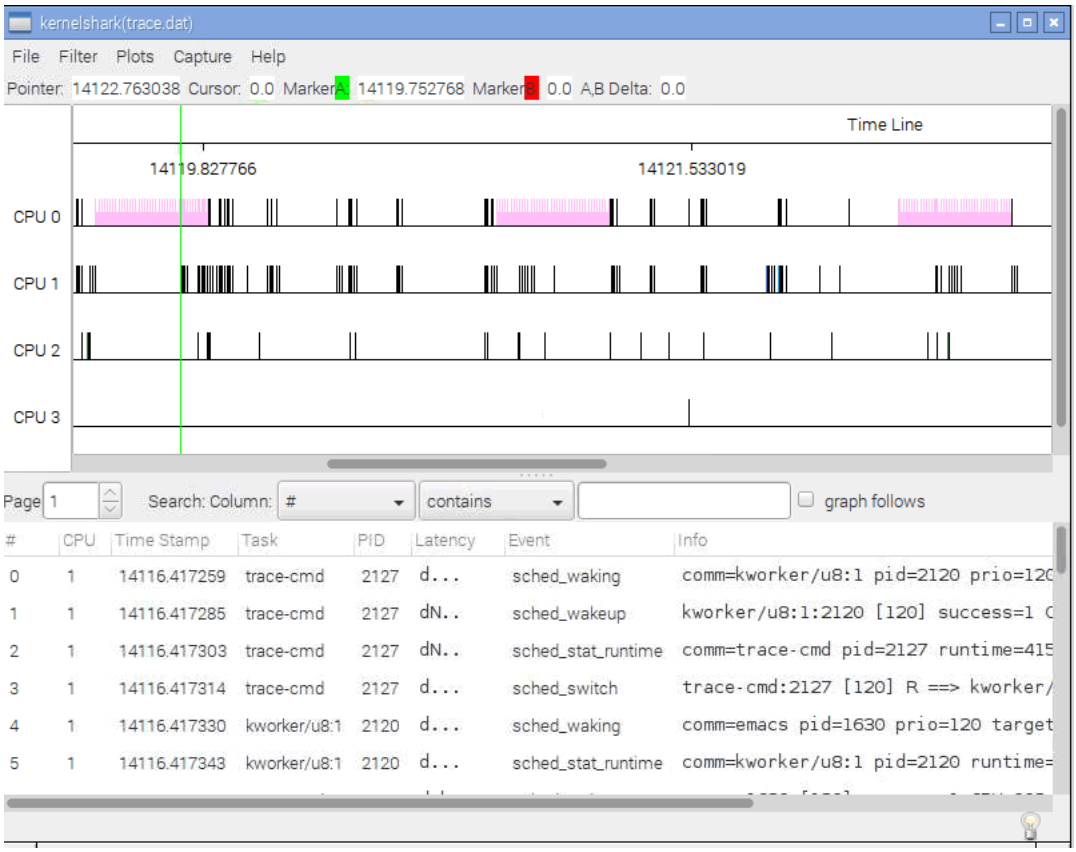


Figure 39: Kernelshark running on Linux (Raspbian) OS

Once the trace which is in CTF format is generated, it can be imported into the Eclipse TraceCompass. An example trace imported into Eclipse TraceCompass platform is given in the Figure 40. 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.

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. The Listing 14 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 is 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.

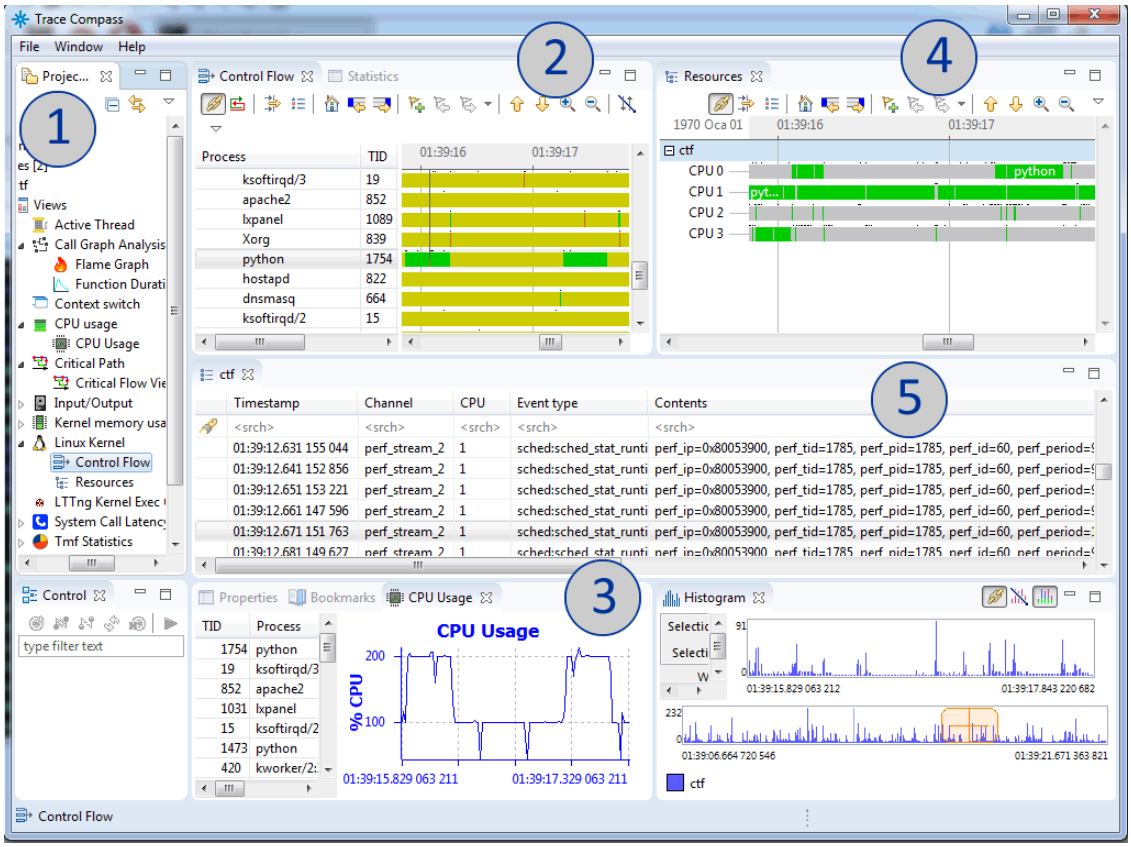


Figure 40: Eclipse TraceCompass running on Windows

```
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 14: Script to generate traces automatically

At this point, it is really important for a developer to let 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 41.

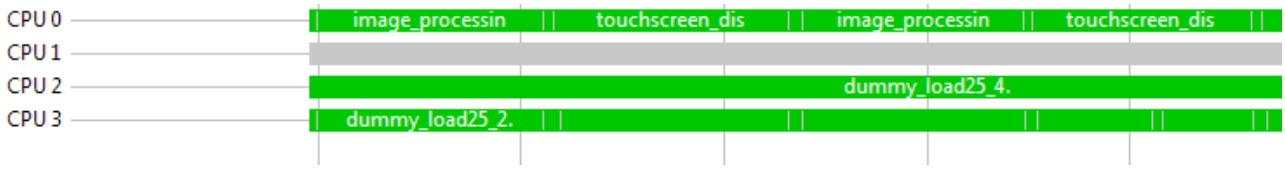


Figure 41: TraceCompass visualization of processes and threads

6.3.5 Process and Thread Mapping

After software evaluation, processes and threads should be pinned to cores properly. To place the processes and threads to cores, tasks taskset module of Linux platform is used. Taskset module [15] 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 [15]. As mentioned in the [15], Linux scheduler also supports natural CPU affinity: the scheduler attempts to keep processes on the same CPU as long as practical for performance reasons [15]. Therefore, in A4MCAR, by forcing a specific CPU affinity, we investigate if a better distribution could be accomplished than the one Linux schedules. To place a process to a core given its process ID, following command is used:

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

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. It can be understood from this example 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 15 depicts how a process is pinned to a core using its name.

```
#!/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 15: CorePlacer.sh script to pin a process to a core using its name

In A4MCAR or any other equally or more complex applications, 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). 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 by the Listing 16. It can be seen in the listing that task names and cores are listed by each line.

```
1  [COREDEF_LIST_APP4MC]
2  Assign Task Xtightvnc To Core 0
3  Assign Task mjpg_streamer To Core 0
4  Assign Task touchscreen_display To Core 0
5  Assign Task ethernet_client To Core 0
6  Assign Task core_recorder To Core 0
7  Assign Task dummy_load25_1 To Core 1
8  Assign Task dummy_load25_2 To Core 2
9  Assign Task dummy_load25_3 To Core 1
10 Assign Task dummy_load25_4 To Core 2
11 Assign Task dummy_load25_5 To Core 2
12 Assign Task dummy_load100 To Core 3
13 Assign Task apache2 To Core 1
14 Assign Thread Thread_UpdateCoreUsageInfo To Core 2
15 Assign Thread Thread_TimingCalculation To Core  3
16 Assign Thread Thread_TouchscreenEvents To Core 0
```

Listing 16: File format that contains overall process pinning information

Reading the file and making the changes required is done by using the function given with the Listing 17. In this listing, 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'. Remembering back the Figure 16, one can better understand how this procedure is located amongst other processes in the high-level module. One should also notice that in the line 25, core affinities of processes are updated.

```
1  def APP4MCDistributionActions():
2    global aprocess_list
3    global aprocess_list_len
4    process_names = []
5    process_affinities = []
6    try:
7      with open('../../logs/core_mapping/coredef_list.a4p','rb') as coredef_list
       :
8        for line in coredef_list:
9          words = line.strip('\n').split(' ')
10         if (len(words)>3):
11           process_names.append(words[2].strip('\n'))
12           process_affinities.append(words[5].strip('\n'))
13   except Exception as inst:
14     print inst
15   lock_aprocess_list.acquire()
16   for i in range(0, aprocess_list_len):
17     for k in range(0, len(process_names)):
18       if (aprocess_list[i].apname == process_names[k] and aprocess_list[i].
         aprunning == 1):
19         if (aprocess_list[i].apid != "NaN" and aprocess_list[i].apid != 0):
20           try:
21             os.system("sudo taskset -pc "+str(process_affinities[k])+" "+str(
               aprocess_list[i].apid))
22           except Exception as inst:
23             print inst
24   lock_aprocess_list.release()
25   UpdateCoreAffinityOfProcesses()
```

Listing 17: Reading coredef_list.a4p and pinning tasks with Python

6.3.6 Discovering 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) [57] 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. Here, Figure 42 [76] depicts how using DVFS can improve the energy consumption in a system.

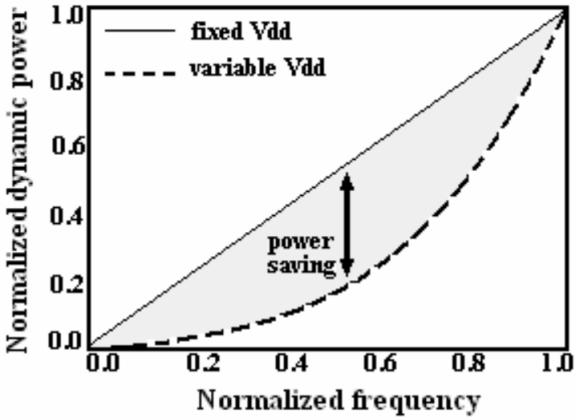


Figure 42: How DVFS reduces energy consumption explained [76]

Consider the dynamic power consumption equation that was given in 8. Considering a fixed chip operating voltage V_{DD}, it can be easily seen from the figure that the power is directly proportional to the operation frequency of the CPU. For example, a processor that is running on 500MHz will be drawing less current than the same one that is 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 to the minimum working voltage of that particular chip [76]. 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 [66]. The figure 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 [26] is provided which is 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 [26]. 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 [44]:

- **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, [44] can be read.
- **ondemand** - adjust based on utilization
- **conservative** - adjust based on utilization but be a bit more conservative by adjusting gradually

CPUFreq governor of a Linux system can be changed at any given moment by using the cpufreq-utils command. The following Linux shell commands shows 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.
```

6.3.7 Online Timing Analysis Features in A4MCAR

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 we refer to 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 43. As seen, 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 3.4). Main processing task can also read from core usage logs and inform users about the low-level module core utilization percentages $LU_{0\text{-}15}$ and high-level module core utilization percentages $HU_{0\text{-}3}$. Furthermore, 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.

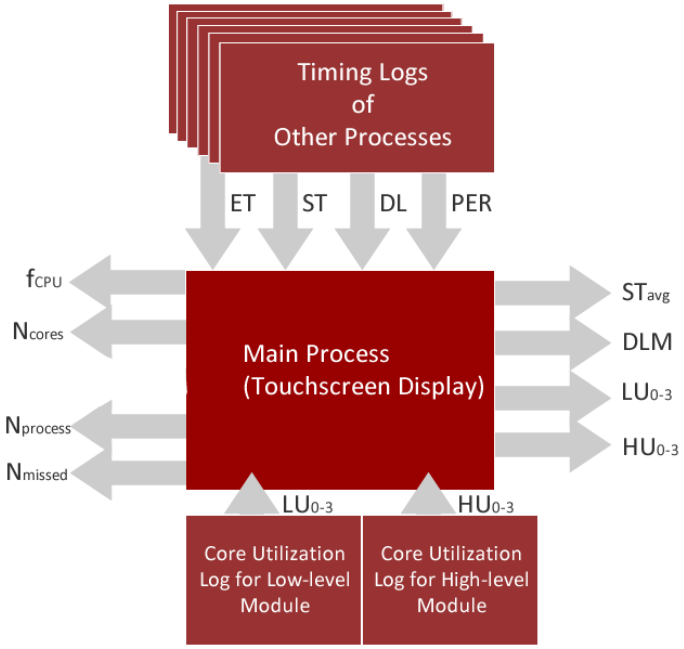


Figure 43: Online Timing Analysis explained in A4MCAR

Subsection 5.5.2 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 the Figure 30 (g). Calculation of the information that are presented to the user ST_{avg} and DLM are carried out as follows:

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

$$ST_{\text{avg}} = \frac{1}{N_{\text{process}}} \sum_{n=0}^{N_{\text{process}}} ST_{\text{n}} \tag{9}$$

- 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 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:

$$DLM = \frac{100}{N_{\text{process}}} \sum_{n=0}^{N_{\text{process}}} DF_{\text{n}} \tag{10}$$

7 Modeling and Results

7.1 System Limitations and Factors that Affect the Results

In this chapter, system modeling with APP4MC and evaluations of mapping outcomes are discussed. A4MCAR, like every embedded system does, has its limitations that are related to both hardware and software. APP4MC has primary objective of 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 the aforementioned interfaces. Furthermore, overheads and limitations are present for custom 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, due to easement, some trivial shared resources and runnables 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 profiler**: Perf profiler uses dynamic analysis to get the granularity of the tasks [46]. 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 significantly higher instruction size than of any other process. Therefore, the goal is to achieve the best possible load balancing with the obtained granularity data.
- **Limitations of XTA**: XTA tool, as mentioned in the section 6.2.1, 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 conflict:** In the low-level module, which uses 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 [39], a combinable function can only be distributed to a core. However, due to the fact that core implementations of some library functions does 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 some tasks. For that reason, the utilization is not visualized very efficiently.

7.2 Modeling the A4MCAR using APP4MC

In the Section **??**, motivations and design techniques of using APP4MC has been introduced. Recall from the aforementioned section that user has to start designing the parallel application by making use of the modeling functionality of APP4MC. In order to evaluate how APP4MC performs using A4MCAR, the design starts by identifying several attributes that are related to software and hardware. With the help of the modeling functionality of the 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, depicted 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, containment 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 44:

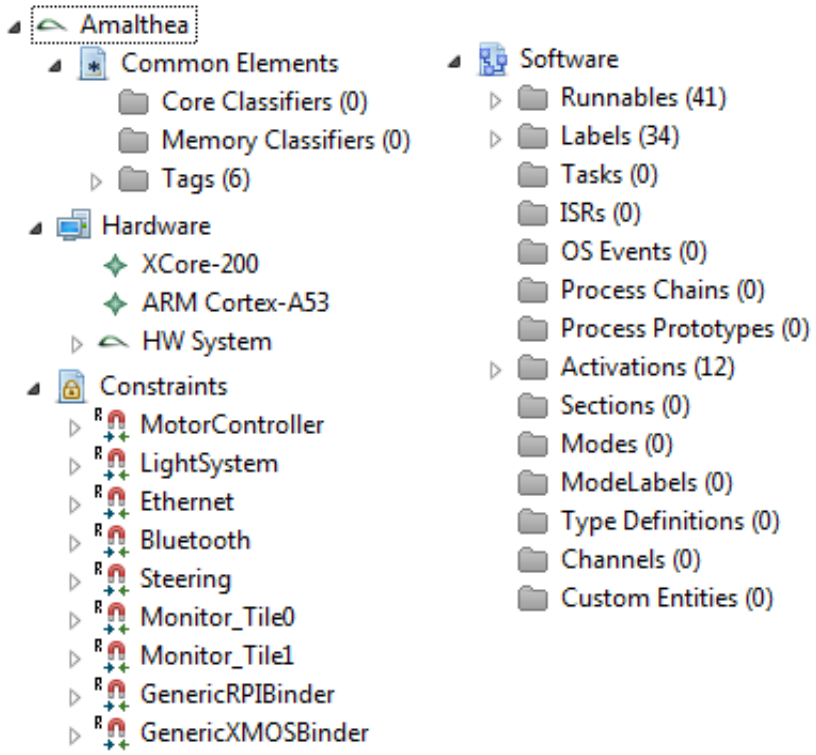


Figure 44: AMALTHEA Contents tree window for the created model for A4MCAR

- **Hardware Model:** Hardware model consists of two processor types: *xCORE-200* for low-level module (XMOS board), and *ARM Cortex-A53* for high-level module (RPI3 board). The *HW System* element contains ECUs (*XMOS* and *RPI3*) with each of ECUs having their respective microcontrollers defined under them. In A4MCAR, *XMOS* ECU has two tiles defined as microcontrollers (*Tile0* and *Tile1*) , whereas *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, 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 when mapping of software processes to the hardware cores are performed.

- **Software Model:** 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 44.

  Recall that in the chapter **??**, the information tracing of multi-tasked systems are discussed. By making use of the aforementioned techniques, runnables, labels, and activations should be defined. Even in a distributed architecture such as A4MCAR, all the runnables from each ECU are listed under the *Runnables* element. Runnables usually are the independent smallest functional parts of a task. However, in A4MCAR some processes, events, and tasks are modeled as runnables due to the ease of modeling. In A4MCAR, an initial analysis led up to a model with 41 runnables. Each runnable listed under *Runnables* element has their granularity (i.e number of instructions) defined. Furthermore, labels are used to define shared variables and inter-process communication. After labels are defined (either by defining bit size or memory size), each runnable are configured depending on their read and write accesses to labels. One should take a look at the Figures 14 and 16 in order to better understand label accesses that are modeled in A4MCAR's

low-level module and high-level module. Finally, activations of each runnable are listed under *Activations* element. Activation 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:** Contraints model define target core dependencies and pairings in the generated partitions. In A4MCAR's model, runnables that functionally belong together are paired using constraints model. As an example, in the Figure 44 it is seen 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 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 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 common elements model. Tags define binders for runnables that are considered in the pre-partitioning phase. Thus, 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. That is, 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.

7.3 Partitioning and Mapping

One important thing to consider when partitioning, 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. To solve this issue, we considered two solutions: (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 is performed by using APP4MC to obtain the distribution results. Obtained results will be shown in the following sections.

For the sake of completeness, how partitioning and mapping outcomes look in APP4MC should be briefly discussed. After the model is complete, by using APP4MC multicore sections drop-down menu, one can perform several operations. After partitioning is complete on the initally created model, new model that involves the process prototypes is generated in the `output` folder in the APP4MC project. Under Process Prototypes, partitions could be seen which shows all the runnables that are in a partition. The Figure 45 depicts an example partitioning outcome.

After the partitioning, one should generate tasks using the same drop-down menu. On the created new model, one can perform mapping. Mapping will generate the utilization information on console for the given model as well as the mapping output on the model as shown in Figure 46. By looking at the model given in the figure, one can see how created partitions are allocated to the cores. If one desires, using these mapping outcomes and using a custom scheduler, the visualization of tasks on cores are also possible by using the APP4MC's multicore feature "Visualize Task Execution". Since in this work real traces are used for visualizing the task execution, this feature is not used.

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 it is said that with a 2-step or n-step algorithm, the load on cores could be balanced more optimally [**?**]. APP4MC's

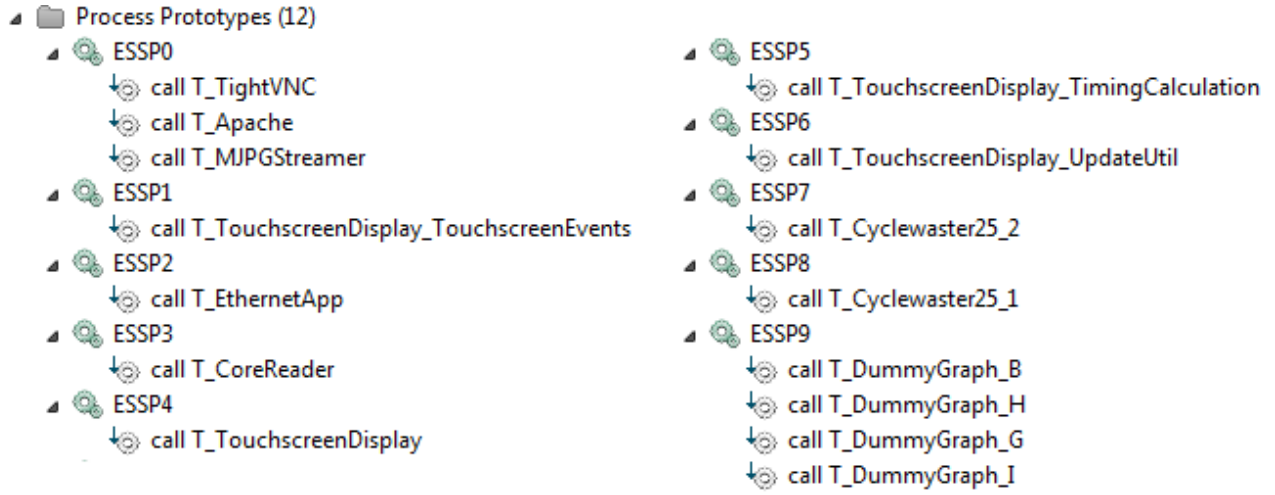


Figure 45: Example partitioning output from APP4MC

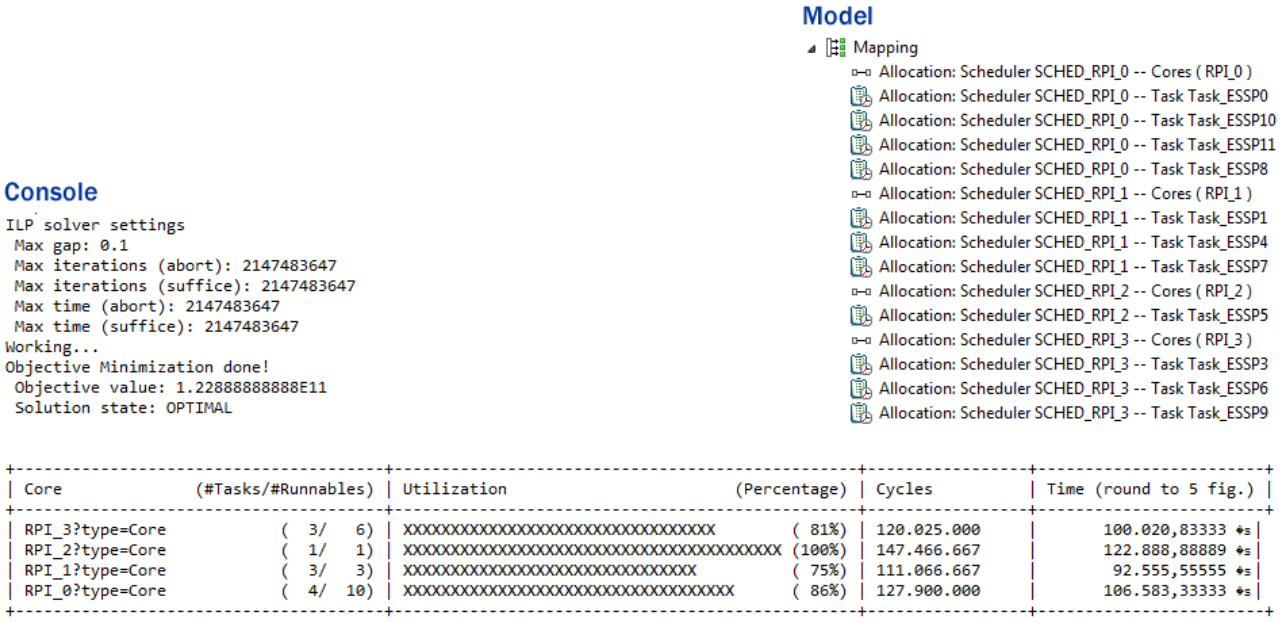


Figure 46: Example mapping outputs from APP4MC

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.

7.4 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 discussed previously, and involves average slack time, gross execution time, computation time, speedup factor, utilization percentages in cores, deadline misses, current used by the processor (in relation to power consumption). Furthermore, tracing results are be visualised using Eclipse TraceCompass for every distribution.

For the sake of clarity, this section introduces all the implemented and modeled runnables with their granularities and activations in the Table 2. Because of the aforementioned limitations, only the dependency graph for the Dummy Graph process is modeled, which is shown in the Figure 24.

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 will be presented to depict how distribution briefly looks and how much of CPU time is fully utilized.

Three mapping results are compared for every high-level distribution at the end of this section: OS distribution where core affinities are not constrained (that is, processes and threads are not 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 abilities.

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. With partitions that have same activation period, placement of runnables to partitions are carried out to reach load balancing. For the mapping stage, for all distributions we select GA-based load balancing technique from APP4MC which is currently aimed towards reducing the overall computation time, as mentioned.

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

7.4.1 APP4MC Results for the Distribution HL_Distr_wStream

Initial distribution for 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 how mapping will result when the camera is used by Camera Stream. That is to say, image processing application is not running but will be given with another distribution.

In the HL_Distr_wStream distribution, following processes are considered active (running, contributes to the processing, and therefore modeled) from the process and thread list given in Table 2: `Camera Stream`, `Web Server`, `Core Recorder`, `Ethernet Client`, `VNC Server`, `Touchscreen`, `Dummy Graph`.

Partitioning and mapping of this distribution using APP4MC resulted in the partitions that are shown in the Table 3.

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 3: Partitioning and mapping results of HL_Distr_wStream using APP4MC

7.4.2 APP4MC Results for the Distribution HL_Distr_wImageProc

The process list when the camera stream is not active but the image processing is active is also modeled and evaluated as mentioned. In the distribution HL_Distr_wImageProc, following processes are considered active from the process and thread list given in Table 2: `Web Server`, `Core Recorder`, `Ethernet Client`, `VNC Server`, `Touchscreen`, `ImageProcess`, `Dummy Graph`.

Partitioning and mapping of this distribution using APP4MC resulted in the partitions that are shown in the Table 4.

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 4: Partitioning and mapping results of HL_Distr_ImageProc using APP4MC

7.4.3 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 2: `Web Server`, `Core Recorder`, `Ethernet Client`, `VNC Server`, `Dummy Graph`, `Touchscreen`, `Camera Stream`, `dummy_load_25_1`, `dummy_load_25_2`.

Partitioning and mapping of this distribution using APP4MC resulted in the partitions that are shown in the Table 5.

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: Partitioning and mapping results of HL_Distr_AvgStress using APP4MC

7.4.4 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 2: `Web Server`, `Core Recorder`, `Ethernet 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 this distribution using APP4MC resulted in the partitions that are shown in the Table 6.

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 6: Partitioning and mapping results of HL_Distr_FullStress using APP4MC

7.4.5 Comparison of High-level Module Distributions and Results

APP4MC is able to display theoretical load utilization among modeled cores of a system after it's done optimizing. The experiments done using GA-based mapping approach with 10-partitions partitioned using ESSP algorithm resulted in the utilization given in the Figure 47.

It should be kept in mind 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

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)	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	(97%)	143.733.734	119.778,11111 ms
RPI_3?type=Core	(3/ 3)	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	(75%)	111.258.334	92.715,27778 ms
RPI_0?type=Core	(1/ 1)	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	(100%)	147.466.667	122.888,88889 ms
RPI_2?type=Core	(2/ 4)	XXXXXXXXXXXXXXXXXXXXXXXXXXXX	(70%)	104.000.000	86.666,66667 ms

FullStress

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

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 ms
RPI_1?type=Core	(4/ 9)	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	(89%)	132.025.400	110.021,16667 ms
RPI_3?type=Core	(1/ 1)	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	(100%)	147.466.667	122.888,88889 ms
RPI_2?type=Core	(1/ 1)	XXXXXXXXXXXXXXXXXXXXXXXXXX	(66%)	97.500.001	81.250,00000 ms

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 ms
RPI_0?type=Core	(1/ 1)	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	(100%)	147.466.667	122.888,88889 ms
RPI_1?type=Core	(6/ 7)	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	(83%)	122.667.067	102.222,55555 ms
RPI_3?type=Core	(1/ 4)	XXXXXX	(13%)	19.500.000	16.250,00000 ms

Figure 47: Resulted Mapping Utilizations from Distributions

core running at 100% everytime, and using a Linux scheduler on top of it would alter the utilizations obtained in actual case.

- Partitioning considers load-balancing, but with only for partitions that have the same type of activation.
- Partitioning gives higher priority to sequences, since considering synchronization between runnables will result in reduced computation time.
- Mapping approach in APP4MC is focused on reducing overall computation time rather than achieving pure load balance.
- Modeled runnables are not very exactly fine-grained as it is in the theoretical or industrial applications. This could also lead to load imbalance between cores.

By applying APP4MC-based, OS-based, and Sequential core affinities to the processes and threads in A4MCAR, the Table 7 is obtained. In the given table, 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, overall execution time (GET), slack time average (ST_{avg}), deadline miss percentage (DLM), utilization in each core ($U_{0\text{-}3}$), and current going into the board (I_{DD}) as an indicator of power consumption. Evaluating the high-level distributions, several remarks could be made, which are given in the following list:

- 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 the times, 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.
- As seen from 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 affinity is not constrained.
- It is also seen from table that changing the clock frequency alone improved the current consumption, but very slightly. One can see this improvement from table which is about 0.01-0.05A. It should be commented that using a lower chip voltage would make this power improvement a lot better. Due to safety concerns, this approach has not been used in this work.

No	Distr.Name	Distr.Type	f_{clk}	GET	ST_{avg}	U_{0-3} (%)	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 7: Distributions compared in High-level module

- One should also note that achieving significantly reduced power consumption is achieved 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 should 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, GA-based results are used.
- It can be seen that power consumption is higher when core usage is high. However, having high core usage makes the system better in 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 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-sequal 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 in Raspberry Pi, and only thread and process can be distributed among cores, threads and processes are modeled as runnables. 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 is concerned about load-balancing only when activation periods of runnables are the same. Since in our particular application, periods are runnables are quite different, partitioning gave results that are not aimed toward load-balancing.
- Furthermore, APP4MC's mapping algorithm is also not focused on load balancing. The main optimization goal used in APP4MC's mapping algorithm is reducing the overall computation time. And 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 7).

To make sure distributed processes (and not threads) affected the results, a second experiment has been done with only threads, Dummy Graph (given in Fig 24), 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.

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, converted to CTF format, and visualized using Eclipse TraceCompass. The Figure 48 demonstrates how OS-based, APP4MC-based, and sequential distribution in 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 49.

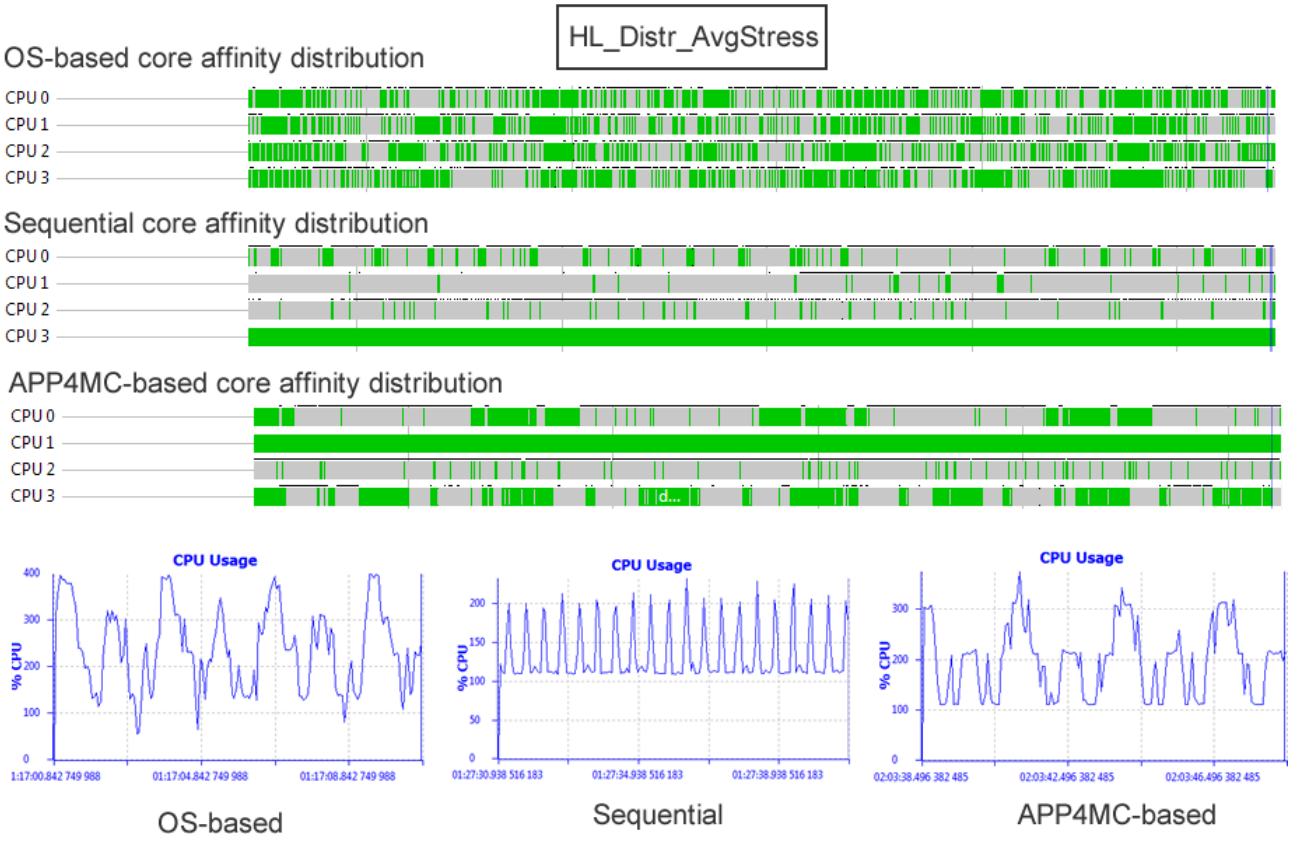


Figure 48: System trace showing how processes and threads are distributed and how CPU performed in HL_Distr_AvgStress

In the aforementioned figures, it is seen 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 seen 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 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 [50] will be used in APP4MC for producing results for applications running in complex OS-based systems.

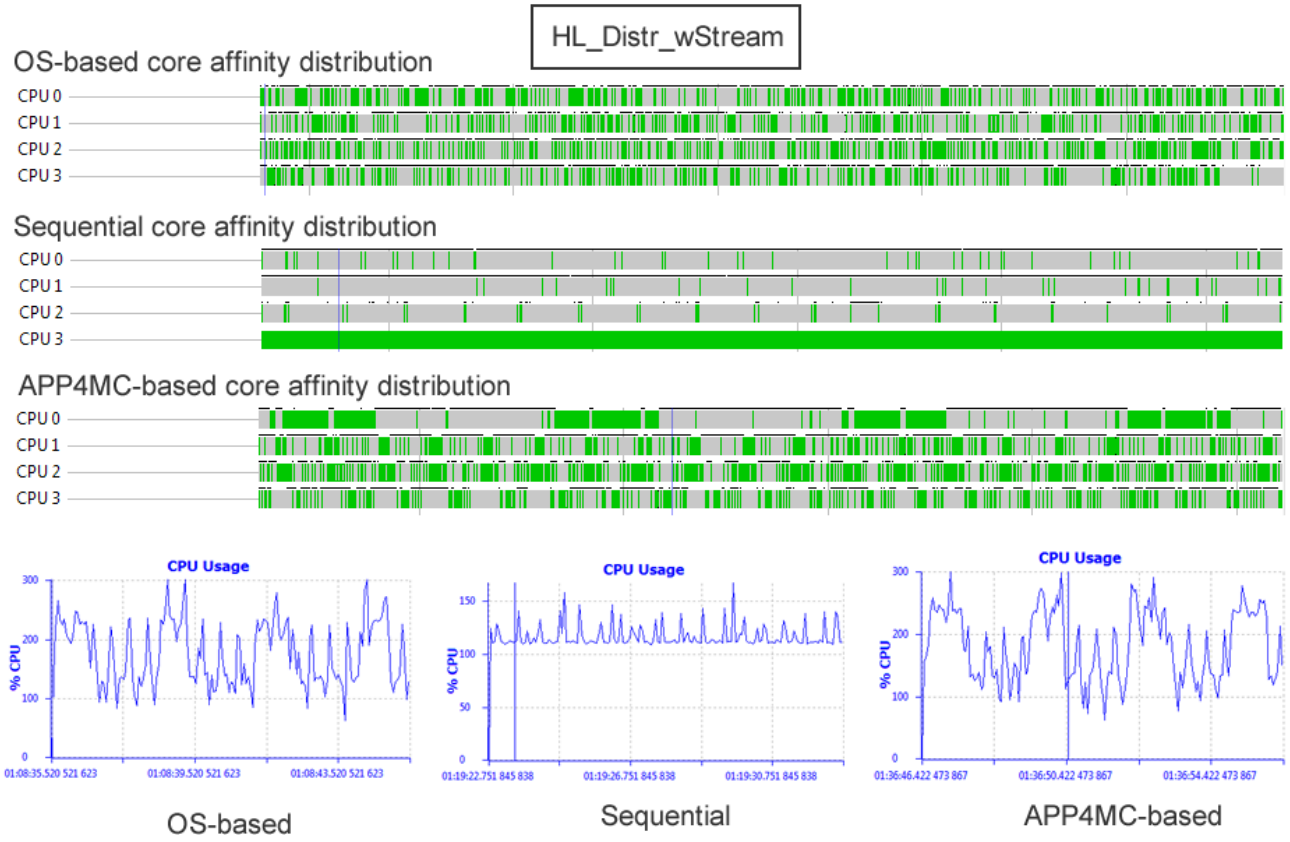


Figure 49: System trace showing how processes and threads are distributed and how CPU performed in HL_Distr_wStream

8 About Using the Software

8.1 Preliminaries

In order to use the applications created for Low-level Module (XMOS) and High-level Module (RPI) folder, the complete hardware of A4MCAR is needed. However, the scripts in the High-level module, Android application, and AMALTHEA models could be re-used in any project without modification.

8.2 Re-using the Software

The software that is developed during the evolution of A4MCAR is distributed under Eclipse Public License (EPL). A4MCAR's repository is located at APP4MC's examples repository `http://git.eclipse.org/c/app4mc/org.eclipse.app4mc.examples.git/` which is downloadable from the git address `git://git.eclipse.org/gitroot/app4mc/org.eclipse.app4mc.examples.git` along with several other demonstrators. Thus it can be downloaded using the following command with Git:

```
1 git clone git://git.eclipse.org/gitroot/app4mc/org.eclipse.app4mc.examples.git
```

The repository contains several folders inside the a4mcar folder, each of which contain the files for distinct development environments along with a readme file. The mentioned folders involve web_interface, high_level_applications, models, android_application, low_level_applications, documentation. These folders and their content scope could be explained as follows:

- **web_interface:** This folder contains the files of the web interface that is developed for A4MCAR project which is used to control A4MCAR over remote Wi-Fi connection.

  In order to set up web_interface and install the dependent software, one should run the setup script: `web_interface/setup_web_interface.sh`

  In order to run the web_interface correctly, the high-level modules core_reader and ethernet_client should be ready and working. To run the web_interface one should connect to the access point of Raspberry Pi from a client computer web browser and visit `http://IP_Address/jqueryControl2.php` or `http://IP_Address/jqueryControl.php`.

- **high_level_applications:** This module consists of several high-level applications that are developed for A4MCAR's high-level module (Raspberry Pi). These applications involve: touchscreen_display, core_recorder, dummy_loads, ethernet_client, and image_processing.

  In order to run the applications, respective Python files could be run or C/C++ binaries could be executed. Also the scripts that are located under scripts folder could be used to initialize some of the applications.

  In order to set up high_level_applications module dependencies, one should run the setup script and follow the instructions: `high_level_applications/setup_high_level_applications.sh`

- **models:** A4MCAR's hardware and software model with Eclipse APP4MC is located in this directory.

- **android_application:** This directory consists of the source files that belong to the A4MCAR's bluetooth based driving application. The source and design files could be used in an Android IDE in order to make tweaks to the application and to generate new .apk files.

- **low_level_applications:** low_level_applications module involves the source code for the low-level module that are run using a multi-core microcontroller XMOS xCORE-200 eXplorerKIT. The low level applications are responsible for tasks such as sensor driving, actuation, communication, and core monitoring of the A4MCAR. The low_level_applications module could be imported into xTIMEcomposer to make tweaks to the tasks.

- **documentation:** This is the directory that involves created documentations for A4MCAR in PDF format.

8.3 Re-using the Scripts

Created scripts for the x86/Linux platform is located under `a4mcar/high_level_applications/scripts/`. In the folder many Bash scripts could be found. The purposes of these scripts involve starting and killing the applications in a4mcar, environment configuration, process manipulation, and tracing. The scripts that are related to process manipulation and tracing is made in a reusable format, thus they can be used in any x86/Linux system regardless of its hardware. The created reusable scripts address tackling the challenges mentioned: placing a process or a thread to a core, monitoring apps, killing processes by their names, tracing and monitoring a Linux system, profiling (getting instruction details dynamically) of the threads of a process, profiling a process, monitoring the threads of a process, and listing the threads of a process.

References

[1] git://git.kernel.org/pub/scm/linux/kernel/git/acme/linux.git.

[2] http://ipm-hpc.sourceforge.net/profilingvstracing.html.

[3] http://lttng.org/.

[4] http://opencv.org/.

[5] https://cs.umd.edu/class/spring2015/cmsc411-0201/lectures/lecture04_performance_reliability.pdf.

[6] https://developer.android.com/studio/index.html.

[7] https://docs.python.org/2/.

[8] https://docs.python.org/2/library/dis.html.

[9] https://docs.python.org/2/library/socket.html.

[10] https://docs.python.org/2/library/threading.html.

[11] https://gcc.gnu.org/onlinedocs/gcc-6.3.0/gcc/.

[12] https://github.com/controlwear/virtual-joystick-android.

[13] https://github.com/jacksonliam/mjpg-streamer.

[14] https://jquery.com/.

[15] https://linux.die.net/man/1/taskset.

[16] https://lwn.net/articles/410200/.

[17] https://open.nasa.gov/blog/what-is-nasa-doing-with-big-data-today/.

[18] https://perf.wiki.kernel.org/index.php/main_page.

[19] https://projects.eclipse.org/proposals/app4mc.

[20] https://pypi.python.org/pypi/psutil.

[21] https://sourceware.org/binutils/docs/binutils/objdump.html.

[22] https://summerofcode.withgoogle.com/projects/#5257433030066176.

[23] http://stackoverflow.com/questions/4310039/user-cpu-time-vs-system-cpu-time.

[24] http://stackoverflow.com/questions/43576997/building-perf-with-babeltrace-for-perf-to-ctf-conversion.

[25] https://ubuntu-mate.org/raspberry-pi/.

[26] https://wiki.debian.org/howto/cpufrequencyscaling.

[27] https://www.android.com/.

[28] https://www.cs.ru.nl/e.poll/ufrj/5_staticanalysisprefast.pdf.

[29] https://www.eclipse.org/app4mc/.

[30] https://www.eclipse.org/app4mc/help/app4mc-0.8.0/index.html.

[31] https://www.eclipse.org/legal/epl-v10.html.

[32] https://www.gliwa.com/downloads/timing

[33] https://www.pygame.org/.

[34] https://www.raspberrypi.org/downloads/raspbian/.

[35] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[36] https://www.tecmint.com/12-top-command-examples-in-linux/.

[37] https://www.w3schools.com/xml/ajax_intro.asp.

[38] https://www.xmos.com/download/private/an01005

[39] https://www.xmos.com/download/private/xmos-programming-guide-(documentation)(e).pdf.

[40] https://www.xmos.com/download/private/xmos-timing-analyzer-manual

[41] https://www.xmos.com/support/boards?product=18230.

[42] http://tldp.org/ldp/abs/html/.

[43] http://tracecompass.org/.

[44] http://trac.gateworks.com/wiki/dvfs.

[45] http://www.amalthea-project.org/.

[46] http://www.brendangregg.com/perf.html.

[47] http://www.clib.dauniv.ac.in/e-lecture/web

[48] http://www.gurucoding.com/en/raspberry_pi_eclipse/index.php.

[49] http://www.jqplot.com/.

[50] http://www.or.deis.unibo.it/kp/chapter8.pdf.

[51] http://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/.

[52] http://www.tldp.org/ldp/linux-filesystem-hierarchy/html/proc.html.

[53] http://www.tldp.org/ldp/sag/html/system-monitoring.html.

[54] http://www.xmos.com/download/private/xcore-architecture-flyer

[55] http://www.xmos.com/download/private/xe216-512-tq128-datasheet

[56] 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.

[57] 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.

[58] Sanford Friedenthal, Alan Moore, and Rick Steiner. Omg systems modeling language tutorial. 2006-2009.

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

[60] Robert Höottger, 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.

[61] 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.

[62] Robert Höttger, Mustafa Özcelikörs, Philipp Heisig, Lukas Krawczyk, Carsten Wolff, and Burkhard Igel. Constrained mixed-critical parallelization for distributed heterogeneous systems. In *The 9^{th} IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems, Technology and Applications*, 2017.

[63] 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.

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

[65] Lukas Krawczyk, Robert Hoettger, and Uwe Lauschner. Introduction in dps architecture, distributed and parallel systems lecture notes, esm dps ws 2015/2016.

[66] 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.

[67] Igor Ljubuncic. Apache web server complete guide. 2011.

[68] 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.

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

[70] 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.

[71] Nicholas Nethercote. Dynamic binary analysis and instrumentation. 2004.

[72] 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.

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

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

[75] 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.

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

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

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

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

[80] 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.

[81] 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.