Universiteit Antwerpen

Faculteit Toegepaste Ingenieurswetenschappen
Elektronica-ICT

Machine Learning-based Hybrid Worst-Case Resource Analysis for Embedded Software and Neural Networks

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Toegepaste Ingenieurswetenschappen
aan de Universiteit Antwerpen te verdedigen door

Thomas Huybrechts

Prof. dr. Peter Hellinckx
Dr. ing. Siegfried Mercelis

Antwerpen, 2022

Contact

Ing. Thomas Huybrechts
University of Antwerp - imec, IDLab - Faculty of Applied Engineering
Sint-Pietersvliet 7, 2000 Antwerp, Belgium
M: `thomas.huybrechts@uantwerpen.be`
T: +32 3265 1683

Version: 27th June, 2022

Machine Learning-based Hybrid Worst-Case Resource Analysis for Embedded Software and Neural Networks

Abstract

With the rise of the Internet of Things and Cyber Physical Systems with Artificial Intelligence, it is important to design computational powerful and energy efficient embedded systems. However, constraints that specifically apply to these embedded systems, such as real-time requirements, energy-aware computing, etc., remain equally important to create safe and reliable systems. These constraints are described as the usage of resources in the context of the worst-case scenario, or in other words the worst-case resource consumption. The need for more powerful and efficient systems is often tackled by introducing optimisation techniques in software and hardware, e.g. cache memory, vectorisation, multi-core processors, etc. Nevertheless, these techniques introduce a negative effect on the predictability of the system behaviour, and thus making it difficult or even impossible to predict the worst-case resource consumption of a task. Without knowledge of this value, it can never be proven that a specific implementation will meet its requirements. Different methodologies exist to determine the worst-case resource consumption, each with their advantages and disadvantages. Furthermore, these methods are only applicable on compiled code. When engineers have insight in the worst-case values during the development process, it will be possible to detect and prevent costly design flaws early-on.

In this thesis, we tackle the previously mentioned issues by introducing a new hybrid methodology that combines machine learning to acquire early predictions on execution time and energy resources of embedded software and machine learning models, without the need to deploy and measure it on the physical hardware. We start with an in-depth analysis of current analysis methodologies, and software/hardware-related influences on the worst-case execution time and energy consumption of a software task. Based on this analysis, we compose a hybrid methodology step by step that consists of a measurement-based and static analysis layer. The first step consists of dividing the code or neural network into smaller components, or blocks. By changing the size of those hybrid blocks, we are able to create a balance between accuracy and computational complexity of both the measurement-based and static analysis layer. Next, we exchange the measurements by a trained machine learning model that analyses each block and provides an estimated upper bound prediction of its worst-case resource consumption. The third step is the extraction of a feature set with relevant software (and hardware/toolchain) attributes that characterises the software task (or neural network) and target platform with their influences on the worst-case resource consumption.

The machine learning-based hybrid worst-case resource analysis has been implemented in our own developed framework, the ‘Code Behaviour Framework’ (COBRA) framework. This tool is a modular analysis framework that allow us to implement, extend and validate our analysis techniques on software code and neural networks, which we discuss in this thesis.

Samenvatting

Met de opkomst van het ‘internet der dingen’ en cyber-fysische systemen met artificiële intelligentie is het belangrijk geworden om computationeel krachtige en energie efficiënte ingebedde systemen te ontwikkelen. Daarnaast dienen deze ingebedde systemen ook te voldoen aan andere harde beperkingen, zoals real-time gedrag, energiebewuste computatie, enz., om zo veilige en betrouwbare systemen te bekomen. Deze beperkingen worden omschreven als het gebruik van systeembronnen in een ‘worst-case’ scenario, ofwel de ‘worst-case resource consumptie’ genoemd. De nood aan krachtigere en efficiëntere systemen wordt vaak aangepakt met de introductie van optimalisatie technieken op software en hardware niveau, zoals cachegeheugen, vectorisatie, multi-core processoren, enz. Desondanks houden deze technieken een negatieve impact in voor de voorspelbaarheid van het systeemgedrag en resulteren dus in een moeilijke, of zelfs het onmogelijk voorspellen van de ‘worst-case resource consumptie’ van een taak. Zonder inzicht in deze waarde is het onmogelijk om aan te tonen dat een specifieke implementatie zal voldoen aan de opgestelde systeem voorwaarden. Verschillende methodieken zijn beschikbaar voor het bepalen van de ‘worst-case resource consumptie’ met elks hun voor- en nadelen. Daarenboven focussen deze methodieken op het analyseren van gecompileerde softwarecode. Indien ingenieurs inzicht zouden krijgen in de ‘worst-case’ waarden tijdens het ontwikkelingsproces, dan zouden zij beter in staat zijn om design fouten tijdig te detecteren en voorkomen.

In deze thesis introduceren wij een nieuwe hybride methodiek die de bovenstaande genoemde problemen aanpakt. Deze methodiek maakt gebruik van ‘machine learning’ om voorspellingen te maken over de uitvoeringstijd en energieverbruik van ingebedde software en neurale netwerken zonder dat deze dienen geïmplementeerd of uitgevoerd te worden op de fysieke hardware voor metingen. We beginnen ons werk met een uitgebreide analyse van de huidige analysemethodieken, en software/hardware-gerelateerde invloeden op de ‘worst-case’ uitvoeringstijd en energieverbruik van een softwaretaak. Aan de hand van deze analyse stellen wij stapsgewijs een hybride methodiek samen die bestaat uit een dynamische (d.w.z. meting gebaseerde) en statische analyse laag. De eerste stap van deze methode bestaat uit het opdelen van de code of het neurale netwerk in kleinere componenten of ‘blokken’. Door de grootte van deze hybride blokken te wijzigen, kunnen wij de balans aanpassen tussen de accuraatheid en de computationele complexiteit van de dynamische en statische analyse lagen. Vervolgens elimineren we de nood aan fysieke metingen door deze te vervangen door een getraind ‘machine learning’ model dat elke

blok analyseert en een geschatte bovengrens voorspelling geeft van de ‘worst-case resource consumptie’. De derde stap is de extractie van een set aan attributen dat de softwaretaak (of het neurale netwerk) en het doelplatform het best typeren met hun invloeden op de ‘worst-case resource consumptie’.

De ‘machine learning’ gebaseerde hybride ‘worst-case resource’ analyse is geïmplementeerd in een eigen ontwikkelde softwaretool genaamd ‘Code Behaviour Framework’ (COBRA). Deze tool is een modulair analyse framework die we uitgebreid bespreken in deze thesis en ons toelaat om onze analysetechnieken te implementeren, uitbreiden en valideren op softwarecode en neurale netwerken.

Preface

It has been 10 years (time really flies!) since my story at the University of Antwerp started as a student in the Faculty of Applied Sciences at the 'Paardenmarkt'. An amazing adventure as told in blockbuster movies in which I was able to feed my hunger for knowledge, make friends for life, and grow as the proud person I am now.

After my studies, I was asked by my supervisor Peter Hellinckx to start a PhD. He gave me the opportunity and full support to improve myself for which I am grateful. My six years as a PhD student were definitely one of the best years yet. During this time, I taught master courses, founded the technology club *iMagineLab* and spin-off company *DigiTrans*, watched our research team grow into *IDLab*, found love, and now I am ready to earn my doctoral degree. However, this would not have been possible without the many people who supported me along the way.

First of all, I would like to thank my supervisors Peter Hellinckx and Siegfried Mercelis for their guidance, advise and support which made this thesis possible. A special thank you goes out to all my colleagues at IDLab and the faculty with whom I had the opportunity to work with and have fun with. They are a close-knit dream team that makes the impossible work possible. I would also like to thank the members of the jury Walter Daems, Maarten Weyn, Kerstin Eder and Roel Wuyts for their great advice and feedback. Their valuable and constructive comments helped improve my dissertation and are very much appreciated. To conclude, I would like to thank my parents for their support they have given me and for the opportunity to follow my dreams and ambitions. And last but not least, I am very grateful to all my friends, family and my partner Maxim who have given me the support, energy and motivation to keep going even through the though moments. Thank you!

As my PhD adventure has come to an end now, a new chapter of my story is just around the corner, ready to be written...

Antwerp, Belgium
June 2022

Thomas Huybrechts

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

DSP Digital Signal Processor. 5

DUST Distributed Uniform Streaming framework. 145–152

DUT Device Under Test. 53, 56, 57, 101, 115

ECU Electronic Control Unit. 2, 12

FFNN Feed-Forward Neural Network. 76, 77, 84, 85

FIFO First-In First-out. 32

FLOP Floating-Point Operation. 97

FPGA Field-Programmable Gate Array. 55–57

FPU Floating-Point Unit. 50, 51

GLPK GNU Linear Programming Kit. 64, 138

GPIO General Purpose Input/Output. 57, 58, 60

GPU Graphics Processing Unit. 5, 50, 95, 96

GUI Graphical User Interface. 139

HDD Hard Disk Drive. 30

HPA Hybrid Program Analyser. 55, 61, 63–65, 68, 73, 93, 127–129, 131–136

I/O Input/Output. 9, 16, 27, 29, 55, 59, 60, 62

IDE Integrated Development Environment. 38

ILP Integer Linear Programming. 15, 17, 63, 64

IoT Internet of Things. 1, 3, 5, 59, 65, 68, 86, 87, 93, 95–97, 100, 121, 142, 145

IPET Implicit Path Enumeration Technique. 15, 63, 93, 138

ISA Instruction Set Architecture. 17, 27, 96

ISR Interrupt Service Routine. 30

JSON JavaScript Object Notation. 93, 147

KPI Key Performance Indicator. 1, 6, 9, 22, 98, 108, 141

L1 Level 1. 38

L2 Level 2. 30, 31, 33, 38

L3 Level 3. 30, 150

LED Light-Emitting Diode. 60

LiDAR Light Detection And Ranging. 144, 145

LLVM IR LLVM Intermediate Representation. 27

LoRa Long Range. 149

LPDDR2 Low-Power Double Data Rate generation 2. 38

LRU Least-Recently Used. 32

MAPE Mean Absolute Percentage Error. 78–80, 82, 84, 87, 89, 92, 101, 103, 104, 106, 108, 109, 111, 112, 116, 117

MCU Microcontroller Unit. 5, 10, 12, 33, 38, 50, 55–57, 96, 98

MISRA Motor Industry Software Reliability Association. 40, 46, 52

ML Machine Learning. 4–8, 24, 52, 69–71, 73–75, 77, 78, 82, 83, 86, 88, 93, 95–101, 120–125, 134, 135, 137, 138, 142–144

MNIST Modified National Institute of Standards and Technology. 100, 101, 108, 112, 115

MRPE Mean Relative Percentage Error. 79, 80, 90, 117, 119, 124

MRU Most-Recently Used. 32

NAS Neural Architecture Search. 98, 142

ONNX Open Neural Network Exchange. 99

OS Operating System. 10, 25, 28–30, 38, 60, 96, 97, 101, 152

PC Personal Computer. 56

PMC Performance Monitoring Counters. 70

QoE Quality of Experience. 144

RAAI Resource-Aware Artificial Intelligence. 5, 7, 8, 141, 142

RAM Random Access Memory. 30, 38, 51, 62

RBF Radial Basis Function. 75, 90, 103, 104, 109, 116, 117

RF Radio Frequency. 3

RISC Reduced Instruction Set Computer. 26

RMSE Root-Mean Square Error. 83–85

RQ Research Question. 6, 7, 68, 93, 120–122

SA Simulated Annealing. 24

SI International System of Units. 88

SIMD Single Instruction Multiple Data. 44, 47, 49

SoC System-on-Chip. 38, 86, 100, 110, 115

SOTA State-of-the-Art. 2, 6, 7, 12, 14, 17, 42, 70, 71, 97, 98, 142–145

SPDP Simple Participant Discovery Protocol. 150

SSE Streaming SIMD Extensions. 47

SVR Support Vector Regression. 75, 79, 80, 82, 88–90, 92, 103, 104, 109, 116, 117

TCP Transmission Control Protocol. 151

TDM Time Division Multiplexing. 33

TPOT Tree-Based Pipeline Optimization Tool. 88, 124, 137

TPU Tensor Processing Unit. 5, 97, 98

TRNN Tree Recursive Neural Network. 77, 84–86

UART Universal Asynchronous Receiver-Transmitter. 56

UDP User Datagram Protocol. 151

UID Unique Identifier. 148

USB Universal Serial Bus. 56, 59, 60, 62

WCEC Worst-Case Energy Consumption. xii, 4, 6, 11, 16, 17, 19, 21, 26–28, 51, 60–63, 65, 68–70, 76, 86, 87, 89–93, 100, 115–122, 127, 133

WCET Worst-Case Execution Time. xii, 3, 6, 10–15, 19–21, 24, 27–29, 31–36, 42, 54, 56, 63, 65–71, 76–84, 86, 93, 100–114, 117, 120–122, 127, 133, 139, 144, 147

WCRC Worst-Case Resource Consumption. 2, 5–11, 17–24, 26, 36, 37, 42, 45, 52–55, 63, 68–74, 76, 77, 79, 86, 93, 95–101, 117, 120–125, 127, 128, 132, 134, 135, 137, 138, 141, 142, 144

XML Extensible Markup Language. 129, 139

Chapter 1

Introduction

1.1 Problem Statement

Embedded systems are hardware systems with electronics that are controlled by software on the device to control or perform specialised tasks, such as programmable appliances at home, in your car, or in the printer that has plotted each character on paper of the book you are reading now. These systems have grown more prominent in our environment in the last decades. For instance, it is estimated that the number of Internet of Things (IoT) devices connected to the internet was 22 billion units in 2018, and it is expected that these numbers will grow to 50 billion by 2030 [1] with a still growing market [2]. With the advent of new technology, new constraints to size, cost price, energy consumption and real-time behaviour (i.e. execution time) are imposed on these systems to create affordable, reliable and safe systems on a massive scale. The code running on these devices is affected by these constraints. Therefore, systems designers need insight in the behaviour of the embedded software with respect to these system constraints.

The constraints on a system depend on the context that system will operate in. This context is translatable into a collection of Key Performance Indicators (KPIs) that represent device and metric-specific weights for objective-based optimisation [3]. These KPIs entail mostly system resource-related constraints, such as energy availability, execution time, latency, storage capacity, etc. In the context of this thesis, we are focusing on two significant markets with hard constraints on specific system resources:

- Real-time systems with hard constraints on **execution time** for task scheduling;
- Intermittent computing devices with very limited **energy** budgets available.

Based on the two domains mentioned above, we have selected two system resources that we will focus on in this thesis, namely execution time and energy consumption. These two types of resources are most prominent in these areas because of the hard constraints imposed on these systems compared to other resources. Furthermore, both resources have no direct correlations with each other [4], why it is important to examine the behaviour of both independently.

The resource consumption of a given software task running on such system is not a single value, but a distribution. When designing a software-controlled system with hard constraints, the worst-case scenario or Worst-Case Resource Consumption (WCRC) of a task should always lie within the imposed boundaries of that system. Determining the WCRC is an active research field with different approaches. Nonetheless, obtaining these worst-case boundaries of a given system is not trivial problem to solve, due to the complexity of software and hardware interactions, and enormous state spaces in complex systems. Furthermore, most of these State-of-the-Art (SOTA) techniques are applied on 'classic' written software code in the C programming language for embedded systems. With the recent advances in Artificial Intelligence (AI) technology getting more performant, it has become feasible to run these models on embedded systems to perform operational decisions. Nevertheless, the WCRC of these models is barely researched in the SOTA.

1.2 Application Domains

This section provides an overview of three application domains that rely on WCRC analysis. The first two application domains are selected to target respectively **execution time** and **energy consumption** as constrained resources. The basis of this thesis is built on top of these two domains that are focused on general logic systems (i.e. source code). The last domain on AI-controlled systems applies both execution time and energy consumption as resource constraints on neural networks. These systems are discussed in the final Chapter 7 and conclude this thesis.

1.2.1 Cyber-Physical Systems

Cyber-Physical Systems (CPSs) are physical machines controlled by software logic in a network of computing devices and sensors that are capable of sensing and interacting with their environment in real-time. These machines play an important role in modern society, e.g. assembly robots, cars, avionics, etc. Unlike general purpose computers, CPS applications require real-time behaviour (i.e. execution time) of the controller units. This guarantees the creation of reliable and safe systems.

There are three classes of real-time systems, i.e. soft, firm and hard real-time systems. The difference between these classes is the importance of meeting the timing constraints and the consequences or impact it will have on the system when they are not met. Missing a deadline on a soft real-time system will result in a quality decrease over time of the service that the system delivers, such as latency in audio and video streams, consistent frame rate in video games, sensor data from weather stations, etc. Firm real-time systems will render its produced results useless after the deadline is passed, i.e. the usefulness of the data is instantaneously equal to zero. Examples of such systems are satellite positioning systems and videoconferencing. Nonetheless, there are no direct fatal consequences for both of these systems.

An (autonomous) car is a perfect example of a CPS with hard real-time constraints. It is from the uppermost importance that these systems not only have correct behaviour, but also are responsive. For instance, a car contains dozens of Electronic Control Units (ECUs) that control specific (critical) systems. The ECU of the braking system should respond to the breaking pedal before a strict deadline in order to prevent catastrophic consequences. In order to design a real-time system, the software tasks need to be scheduled on the hardware platform, so they meet their corresponding deadlines. However, to determine the schedulability of each task we need to know the longest possible time required to execute each task [5]. This value is the WCET. However, determining this value cannot always been taken for granted, as different optimisation techniques used in embedded systems and compilers influence the deterministic behaviour of the software, such as pipelining, branch prediction, pre-emption, parallelisation, etc. As a result, the WCET analysis becomes complex to perform. In the state-of-practice, these influences are often simplified or neglected and compensated with a safety margin resulting in a less tight or underestimated upper bound [6].

In order to determine the schedulability of software tasks, a timing analysis is required to calculate the WCET. For instance, this value is required by the scheduler of operating systems and hypervisors to schedule all tasks within specified time frames on the system [5]. Therefore, a timing analysis is performed to calculate the WCET of each task. In practice, margins are added to the calculated upper and lower bounds of the timing distribution because of the complexity of the analysis. For example, when taking the influences of the cache into account, a cache miss will always be assumed when calculating the upper bound. This pessimistic attitude is not realistic and will eventually lead to over-proportioned systems [7].

1.2.2 Internet of Things Devices

Many IoT devices are small or mobile nodes that are operated by batteries, as providing outlet power to the vast number of devices on different and/or remote locations is not feasible. Nevertheless, these battery-operated devices come with a high cost. Batteries have a large impact on the environment because they contain heavy metals that are bad for your health and contribute to the greenhouse emissions during the exploitation of the required minerals and the actual production [8], [9]. Furthermore, the limited lifespan and energy capacity of batteries require regular servicing of these devices on site which comes with high operational costs. When the number of IoT devices reach 50 billion, we will need up to 1 trillion batteries to power those devices. With an optimistic lifespan of 10 years, a total of over 274 million batteries needs to be replaced every day that end up in toxic waste [10]. As a result, energy is a hard-constrained resource for these devices.

An alternative to the relative short-lasting and environmentally harmful batteries is the evolution to batteryless devices [11]. The batteries in these devices are replaced by an energy harvesting circuit harnessing environmental sources, such as solar-, kinetic- or Radio Frequency (RF)-energy [12], and a (super)capacitor storage element. A capacitor has a longer lifespan compared to batteries due to the larger number of recharge cycles it is able to handle [13]. Supercapacitors in particular have higher power density (W/kg) characteristics and charge much faster than batteries. However, the energy density (Wh/kg) of these storage elements is significantly lower [14]. Therefore, the available energy is even more scarce for batteryless devices. In order to operate within these constraints, the software needs to schedule its tasks while taking the available produced and stored energy into account.

In the state-of-practice, most schedulers in batteryless devices have no knowledge of the available energy and take a best effort approach to schedule all tasks [11], [15] as illustrated in Figure 1.1. Therefore, tasks may fail when the voltage has dropped below the cut-off voltage before they are completed. Failed tasks need to be resumed later when the capacitors are back at full charge. In the example of Figure 1.1, all tasks are scheduled one after the other. The execution of Task 3 is interrupted, because the

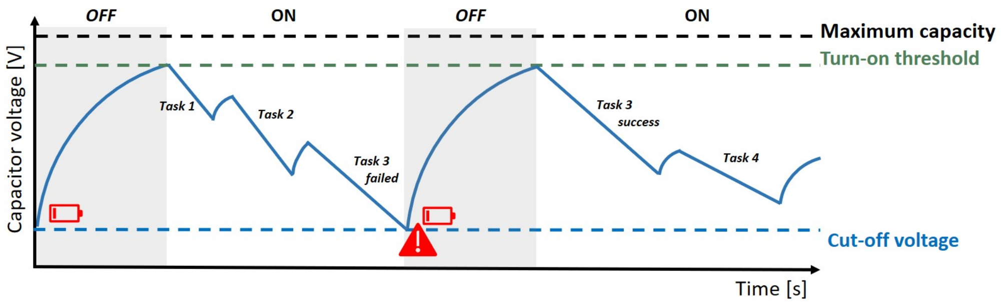


Figure 1.1: Charge cycle of the capacitor in a batteryless device without an energy-aware scheduler. The first try of Task 3 fails as the capacitor voltage drops below the cut-off voltage. After a full recharge of the capacitor, the task is successfully rescheduled.

capacitor has dropped below the cut-off voltage. After the capacitor is above the turn-on voltage, the failed Task 3 is retried. As a result, valuable energy is lost during the failed task execution that could have been used for another task, or the task could have been finished successfully if the device had waited until enough energy was available in the capacitor. Furthermore, failing tasks are prone to miss their execution deadline resulting in corrupt, incomplete or obsolete data and actions. Current research is being conducted to create schedulers that are aware of energy availability [16], [17]. This will allow the scheduler to turn into a low-power mode to avoid rescheduling failed tasks due to power loss, and eventually lowering the chances of missing execution deadlines. Nonetheless, the scheduler requires insight in the WCEC of each task in order to schedule them, which are not trivial to obtain.

1.2.3 Machine Learning on Embedded Devices

The classic approach of developing control logic for embedded systems entails writing the software algorithm that describes the problem and/or solution to calculate the output based on the input and a set of rules. This approach works excellent for simple predictable behaviour of the system, such as the basic laws of physics, or a simple decision tree. Nevertheless, systems with complex interactions require in-depth knowledge in the data to compile a list of rules which the control logic needs to comply with. For example, the detection of anomalies in the operational behaviour of a production assembly machine. Data streams of dozens of sensors are fed to the controller for processing. The developer needs to understand which sets of datapoints indicate an upcoming problem, i.e. an anomaly. These sets will consist of intricate combination of different factors, such as temperature, production rate and vibrations. Finding these sets is a tedious process that is error-prone and difficult to maintain. A more feasible approach is to have the algorithm learn these patterns in the data and the corresponding rules by itself. This AI concept for which a prediction model is trained on data is called *Machine Learning (ML)*. The final model is then able to provide a prediction for a given input vector based on previous datasets seen during the training step. This process is called inference.

While the concept of AI was first introduced by McCarthy et al. [18] in 1955, it only has a significant breakthrough in the last decades due to the availability of more powerful computational hardware, large datasets, and new architectures, such as deep learning. The growth in AI has resulted in the emergence of new research and application domains. The innovative approach to create applications based on data without fully understanding the system's complexity opens the question to integrate these prediction models in different domains, such as CPS and IoT. These integrations have resulted in revolutionary technologies that were deemed impossible with classic programming, e.g. autonomous driving, anomaly detection, predictive maintenance, etc. Nevertheless, CPS (Section 1.2.1) and IoT (Section 1.2.2) systems still need to comply with the hard resource constraints that are imposed on them. Therefore, we are actively exploring the behaviour of AI models on the resource consumption. A summary of our research within the Resource-Aware Artificial Intelligence (RAAI) team is included in Appendix B.

The development of an ML model has some significant differences compared to traditional programming logic. The process consists of four prominent steps:

- Collecting a dataset that represents the data to which the deployed system has access to as input for the model;
- Designing a model architecture and processing pipeline that suits the best to perform the prediction (e.g. input/output vector, number of hidden layers, feature generation, pre-processing data, etc.);
- Training the model with a subset of the (annotated) dataset, and validating its performance with the remaining data;
- Deploying the model and running inference in the environment on the target platform. Additionally, keeping track of the performance of the model in production, and adapt/retrain it if the model is underperforming.

Designing an ML model is commonly developed in specialised frameworks, such as TensorFlow [19] and PyTorch [20]. These frameworks contain optimised implementations for the operators of ML architectures, and specialised tools for training and validating ML models. Using powerful server infrastructure with Graphics Processing Unit (GPU) capabilities makes it computational feasible to train and run these models. Deploying these models on edge devices, such as IoT sensors, smartphones, etc., enables us to distribute the computational load and lower the bandwidth demand of IoT applications [21]. However, this will require significant optimisation efforts in order to make the models executable on these devices, such as being small enough to fit on the constrained resources of these embedded edge devices or performing quantization on the weights and biases of the model to fit them into 8-bit integers instead of 32-bit floats. For example, Google uses small Deep Neural Networks (DNNs) to create accurate wake word detector models that are deployable on Digital Signal Processors (DSPs) of mobile devices as they are just a few KBytes in size [22], [23]. Specialised DSP chips, or Microcontroller Units (MCUs) in general enable low-power computing (i.e. within the range of milliwatts) at a relative low cost per unit. The concept of deploying optimised ML models on ultra-low-power devices ($\leq 1\,\mathrm{mW}$) is called *TinyML* [21], [23]. The TensorFlow framework already contains a '*Lite*' extension that enables embedded developers to convert a full TensorFlow model into an optimised version for MCUs.

In the domain of WCRC analysis, there is little research on the resource behaviour of neural networks on MCUs, except for some specialised hardware (e.g. Tensor Processing Unit (TPU) [24]). Techniques to determine the WCRC in the state-of-practice are mostly limited to basic logging, probing, and (shotgun) profiling [23].

1.3 Contributions

In the introduction, we discussed the importance of the WCRC as an important KPI to create safe, and reliable systems. Nevertheless, acquiring insight in these constraints, such as execution time and energy, is not a straightforward process. An overview of the SOTA in WCRC analysis is presented in Chapter 2. As there exist different methodologies to determine an upper bound of the resource consumption, each technique has its advantages and disadvantages nonetheless. Additionally, each of these approaches focus on performing analysis on compiled binaries, and therefore is only applicable in the final validation stage of the process. Furthermore, we notice that the novel field of resource-aware embedded AI and its impact on WCRC behaviour is insufficiently explored in the SOTA.

We want to tackle these problems by examining a sound and computational acceptable approach to determine the WCRC on embedded platforms and create a solution for early estimation of the WCRC during development with fast feedback on the influence of a code or model change on the upper bound. Based on these gaps and criteria, we have defined the following Research Questions (RQs):

- **RQ 1**: *How can we approximate an upper bound on the WCET/WCEC that is computational acceptable to perform and still provide a sound bound, and can we create a dynamic trade-off between those two?*
- **RQ 2**: *How can the hybrid WCRC analysis be employed to provide early predictions without performing physical measurements on the target platform?*
- **RQ 3**: *How can we predict the WCRC of neural network powered control logic by solely using descriptions of these networks?*

Each of these questions will be answered within the course of this thesis as outlined within Section 1.4. Chapters 5 - 7 will discuss our work that we extended layer upon layer. Each of these chapters addresses one of the questions. The final conclusions of our RQs are summarised in Chapter 8.

Within the scope of the problem statement in Section 1.1 and the RQs, we developed new methodologies and tools to extend upon techniques used in the SOTA. The main contributions of this thesis are as follow:

1. An overview of **challenges and solutions** to perform WCET/WCEC analysis of software/ML tasks on embedded systems with hard constraints;
2. A **hybrid analysis approach** to create a dynamic trade-off between computational complexity and precision of the WCRC upper bound estimation;
3. Extending the **hybrid analysis with ML** to estimate the resource consumption without (physical) measurements on the target platform;
4. Applying hybrid ML-based analysis techniques for **WCRC estimation on embedded AI systems**, namely neural networks;
5. Development of tools for the extension of the **Code Behaviour Framework (COBRA) framework** to apply and validate our new methodologies.

The basis of this thesis focuses on two types of resources, i.e. execution time and energy. Nonetheless, the principles of the proposed hybrid methodology could also be applied to other resources with hard constraints, such as latency, memory, throughput, etc. However, applying and validating our framework on other resources is beyond the scope of this work.

The tools we developed within the context of this research have been integrated within a multi-functional framework for code behaviour analysis as presented in Appendix A. Furthermore, these tools and methodologies are actively used within the RAAI team by fellow researchers to support their research. An overview of these interactions and dependencies is summarised in Appendix B.

1.4 Outline

Based on the RQs that are formulated in Section 1.3, we have structured the thesis in six main chapters as is illustrated in Figure 1.2. With each chapter, we add new insights and extensions to our methodology to eventually perform WCRC estimations of neural networks in Chapter 7. The content of each chapter is derived from the work of previously published articles. An overview of the related published work is available in Appendix C. The outline of this work is as follows:

- **Chapter 2** introduces the *basic concepts of WCRC* and SOTA analysis methodologies to determine timing and energy resource consumption. Content of this chapter has been published in the following publications: [25]–[29].
- **Chapter 3** discusses the most prominent *platform-related influences* that have an impact on the resource consumption, interfering with the predictability of the system. Content of this chapter has been published in the following publication: [30].
- **Chapter 4** continues with the impact on the WCRC of tasks from the *influences of the software logic's point of view.* Content of this chapter is part of a progress report.
- **Chapter 5** presents the *hybrid WCRC analysis methodology* we apply to determine the resource consumption of embedded devices, and how we integrated this methodology into an automated framework for profiling. Content of this chapter is part of a progress report and has been published in the following publications: [26], [29].
- **Chapter 6** extends on the hybrid methodology to gain early *estimations of the WCRC by employing ML techniques* to predict the resource consumption instead of physically measuring it on the system. Content of this chapter has been published in the following publications: [27]–[29].
- **Chapter 7** applies the developed hybrid methodology for *WCRC analysis on neural networks* instead of traditional programmed tasks. Content of this chapter has been submitted for publication.
- **Chapter 8** summarises the results and formulates the final *conclusions* on the research questions of this thesis.
- **Chapter 9** concludes the thesis with an overview on *future work* to further extend and improve our proposed methodology in follow-up research.

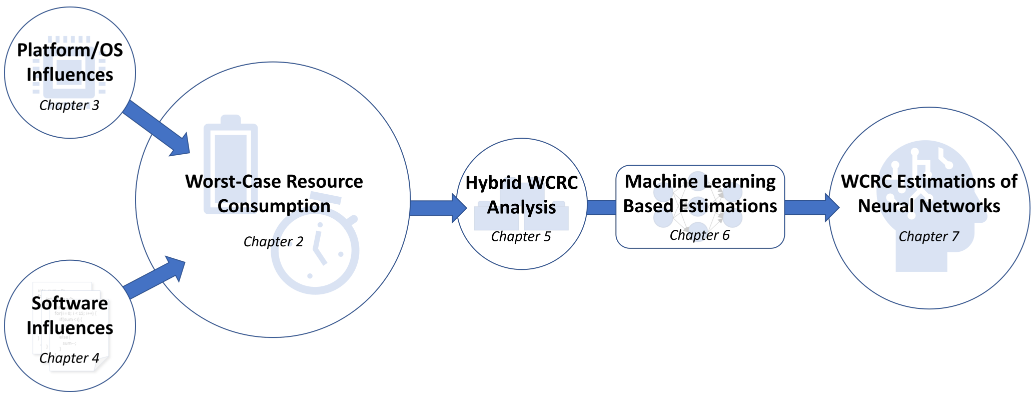


Figure 1.2: Schematic outline of the thesis' chapters: starting from the concept of WCRC (Chapter 2), discussing platform and software influences (Chapters 3 and 4), presenting a hybrid WCRC analysis approach (Chapter 5), extending the hybrid approach with ML (Chapter 6), and finally estimating the WCRC of Neural Networks (Chapter 7).

- **Appendix A** presents the *COBRA framework* that is developed within our team and summarises the work on extending the framework to implement and assist the research of this thesis. Content of this chapter has been published in the following publications: [25]–[29].
- **Appendix B** provides an overview of the work on *RAAI* performed within our team for which the WCRC analysis is an essential component in the overall implementation of the research. Content of this chapter has been published in the following publications: [31], [32].

Chapter 2

Worst-Case Resource Consumption

Designing safe and reliable CPS and other embedded systems, we need to have insight in the behaviour of software on the hardware. One of these criteria is the resources the software logic requires to operate on the target hardware. However, each physical system has only a finite available amount of resources to its disposal, due to physical or budgetary reasons. Nevertheless, if the system requires more resources than there are available it will have a detrimental impact on the functional operation of the system. This results in unintended behaviour that could lead to catastrophic accidents with CPS, such as a car accident or a pacemaker failing. Therefore, we need to know the boundaries of the required resources the system needs in order to guarantee deterministic behaviour in all scenarios. To achieve this compliance in all situations, we are in most cases interested in the worst cases, or simply put the WCRC as the upper limit the system would possibly need to function under operational conditions.

In this chapter, we will look at two different resources, i.e. execution time and energy, and how to determine their respective worst-case consumption. Next, we discuss different methodologies to get insight in the WCRC, and finally the advantages and disadvantages these techniques entail.

2.1 Resource Consumption

System resources of an embedded system are physical or virtual constrained elements due to their finite or limited availability. These resources apply to different subcomponents of a system, such as memory, network, Central Processing Unit (CPU), Input/Output (I/O), energy, etc. The availability of resources is (mostly) determined by the hardware of the system. In theory, we could scale the system's hardware infinitely to prevent any shortage. However, this would include an enormous financial cost that is practically unfeasible to justify. Therefore, the resources of commercial embedded devices are scaled down to a minimum to conserve costs from an economic standpoint.

Limiting the available resources of a system is an important KPI for a system engineer to minimise the unit price. Nonetheless, failing to provide sufficient resources during the operation of the device could lead to unintended behaviour of the system with potential lethal consequences by CPS, as discussed in Section 1.2. As a result, we need to determine if the resource requirements, or consumption, for those systems with (critical) functional requirements are guaranteed during operation to obtain safe systems. In the context of this research, we limit us to those systems that are controlled with software on MCUs.

An embedded system consists of a hardware controller with software logic that operates the functionality/intelligence of the device. The operation of an embedded system consists of executing a sequence of software tasks. The behaviour of the software code is also depending on the resources of the system. For example, a continuous task could run on a higher periodicity when the system's clock frequency is increased (within limits), or a task will fail when no sufficient memory or energy is available. As the system resources are bounded, a balance should be found between decreasing the software task's resource requirements (e.g. optimising the software), or increasing the available resources (e.g. adding hardware, and thus increasing costs).

The resource consumption of a software task is generally not just one deterministic value, but a distribution of possible values that is determined by the behaviour of the code. When a single task is executed in multiple consecutive invocations, the observed resource consumption will fluctuate depending on different factors, such as which code traces are executed, provided input parameters, state of the hardware, etc. As a result, we are able to plot a distribution as the example used in literature that is shown in Figure 2.1. We obtain such a graph if a given task is executed with every possible input and internal state. The horizontal axis indicates the resource consumption (e.g. power consumption, latency, clock cycles, etc.). The vertical axis is the number of times a certain consumption value is observed for different inputs/states of the task. As a result, we note three important indicators: the best-case, average-case and worst-case consumption. These boundaries give a good indication on the variation of the resource consumption. The shape of the distribution in Figure 2.1 has an atypical pattern with the average of the curve that is mostly shifted to the best-case (i.e. left-hand side). This behaviour is easily explained by the design of most systems that employ techniques and hardware optimisations to improve the most frequent occurrences, or in other words the average-case scenarios. Nevertheless, the optimisations have little or even a negative impact on the less common worst-case scenarios resulting in a long tail with outliers. In our research of creating safe CPS and resource-aware systems, we are mostly interested in the worst-case or WCRC. This value indicates the resource requirements/consumption of the task in the worst possible scenario. When the system is able to schedule the task within the boundaries of the available resources for the worst-case scenario, we have certainty that the task will execute successfully. With the example in Figure 2.1, we see that the worst-case scenarios are less common during execution, and that the worst-case tail is significantly longer compared to the best-case tail. Therefore, it is less trivial to trigger the worst-case, but on the contrary no less important to determine, as missing the major outlier in the worst-case scenario will compromise the safe and sound operation of critical resource-aware systems, such as CPS and batteryless devices.

Obtaining the real best- and worst-case boundaries of a task is hard or even impossible to achieve on complex hardware. These values cannot be precisely determined, due to unknown external interference and limited knowledge of the system [33]. For example, the system state and non-determinism of the hardware could result in local WCET measurements of a (sub)task that are not part of the actual global WCET of the application. This phenomenon is called *timing anomalies.* These anomalies are introduced by hardware optimisations with non-deterministic behaviour that are designed to improve the average-case performance, but impact the WCRC (e.g. cache memory, speculative execution, greedy scheduling, etc.) [34]. We discuss different platform/Operating System

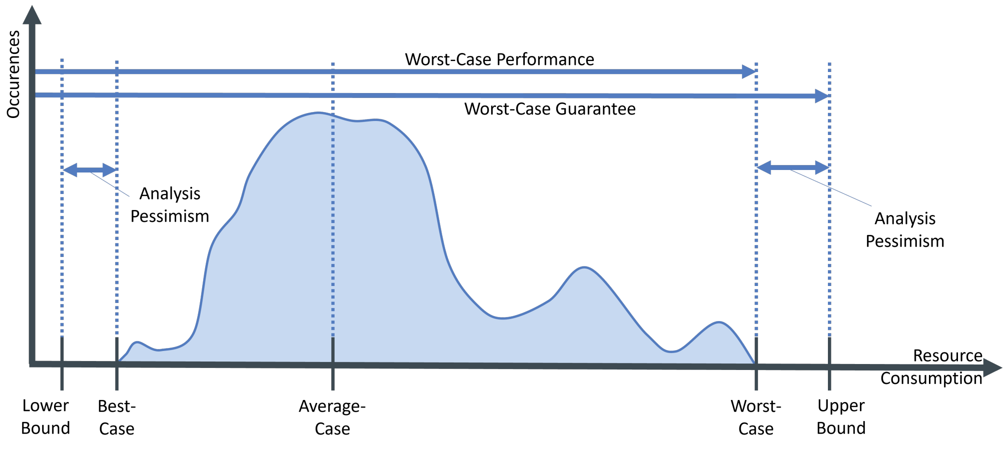


Figure 2.1: Distribution of resource usage of a given task. Indicating the best-, average- and worst-case resource consumption. The lower and upper bounds indicate the approximations of respectively the best- and worst-case consumption.

(OS) (Chapter 3) and software (Chapter 4) influences on the WCRC in the following chapters. In order to tackle this uncertainty, we determine a lower and upper bound for respectively the best- and worst-case values. The difference between the real values and the predicted bounds indicates the *analysis pessimism*, and thus the predictability of the system. During an upper bound analysis, we want the upper bound as close to the real worst-case value, as overestimating the upper bound results in redundant resources, and so over dimensioning the hardware of the product. The opposite situation for which the upper bound is smaller than the worst-case is problematic, as underestimating the WCRC could lead to detrimental behaviour of the system. The upper bound is therefore sound when it is greater or equal than the real worst-case value.

The predictability of a system is not the only important factor in the design process. CPS and other hard-constrained systems require guarantees on the WCRC. The worst-case guarantee is linked to the upper bound. Another indicator is the worst-case performance of the system. This indicator is equal to the actual WCRC value. The worst-case performance and guarantee are just as important as the predictability of the system. For instance, a system can be designed in such a way that it is perfectly predictable (i.e. no analysis pessimism) but has a poor worst-case performance (i.e. high WCRC). This would result in a poor performing system (e.g. slow, high energy draw, etc.). In this thesis, we will focus on two types of resources that are significant within the established application domains of Section 1.2, namely execution time (i.e. WCET) and energy consumption (i.e. WCEC).

2.1.1 Worst-Case Execution Time

The execution time of a task is the time interval from fetching the first instruction until the completion of the final instruction of that task, i.e. end-to-end. The time needed to execute these tasks depends on the path through the control flow and the time required for each instruction to run on the target hardware. In order to find the WCET, we need to determine the input and initial state of the system that would lead to the worst-case path. However, these variables are unknown and mostly impossible to determine [35].

We can express the execution time measurement as (micro-)seconds in the physical world. However, the MCUs are synchronous systems that use a central clock to synchronise all operations. A clock cycle is the period between two pulses of this clock. Therefore, we are able to use these pulses as measurement unit for easy representation and calculations. Obtaining the real execution time is simply multiplying the number of clock cycles times the clock period (i.e. inverse of clock frequency) of the CPU clock, with an error margin $|\delta| \leq \epsilon$, and ϵ equal to the accuracy of the hardware clock [36].

In order to improve the execution time of a program, a variety of optimisation techniques are used to improve the throughput of the processor, such as pipelining, caches, multi-core on hardware level; and pre-emption, parallelisation on software level. These techniques are often used in multi-purpose computer systems, e.g. desktops, laptops and servers. It is not a life-threatening issue for these systems that a calculation takes more time to complete in certain situations. In these systems, the average execution time of the code is optimised. However, the WCET is a very important factor for time-critical systems, such as CPS. For instance, the ECU that controls the airbag system of a car should respond to a collision within a strict time frame. Therefore, it is important that these systems are deterministic. Optimisations are necessary in common embedded systems, but most compromise the deterministic behaviour of the software running on those processors. In the SOTA these effects are often neglected, or the hardware is overdimensioned to minimise the impact, due to the complexity of the predictability analysis. As a result, powerful and expensive processors are used in systems in which they are not necessary. The designer wants to make sure that the real-time constraints are met. Therefore, he/she will overestimate the WCET by underestimating the benefits of the optimisation techniques mentioned above. It is important in real-time embedded systems that the tasks are handled within the specified time frame. The execution time of a task in a system is not ambiguous but follows a time distribution as shown in Figure 2.1.

Hard real-time systems need to manage their tasks in such a way that they are all executed within the given time frame, i.e. before a specific deadline. When using an operating system, this is done by the scheduler. It schedules the tasks according to a particular scheduling algorithm. The art of analysing whether a set of tasks is schedulable using a specific scheduling algorithm is called schedulability analysis. In the 70's, Liu and Layland [37] considered the schedulability of a set of tasks with a lot of assumptions. However, these constraints do not always hold in the design of real-time embedded systems, because tasks are rarely independent and often reactive instead of periodic. Later work focuses on extending the work of Liu and Layland to relax the outlined constraints. More recently, Bini et al. [38] improved the bound given by Liu and Layland for the monotonic algorithm.

In multi-core processor environments, schedulability analysis is significantly more complex. A wealth of research has been performed on scheduling and schedulability analysis for multi-core processors, as presented by the survey of Davis et al. [39]. However, the same assumptions were made for the analysis as Liu and Layland did on single-core processors [37]. In [40], Palencia proposes a solution to the problems of task suspension and multiprocessor environments if task offsets could be dynamic, i.e. if they could change from one activation to the next. For example, in parallel programming a task may be released when a previous task completes its execution, and a message is received; this release time may vary from one period to the next.

Acquiring a precise estimation of a task's WCET during analysis has different challenges to tackle. Firstly, the control flow of a task is *data dependent*. The flow of a program consists of a sequential execution of instructions. Nonetheless, the control flow contains dynamic jumps/calls in the instruction sequences to execute different paths based on conditional and jump instructions. This allows us to have dynamic behaviour of the program at runtime. If we combine all paths of the control flow in a graph, we will obtain the Control Flow Graph (CFG). Each dynamic call or jump operation that 'manipulates' the flow of the application is an edge in the graph. To determine the WCET of the task, we need to find that path through the CFG that would result in the WCET. The analysis is difficult to perform as we need for example to exclude infeasible paths (i.e. path through the CFG that will never be executed at runtime), and find loop bounds of iteration and recursion statements. The latter is important to limit the number of iterations. As iteration conditions are dependent on variables and/or external input, it requires insight in the runtime environment to obtain precise and feasible results.

Secondly, the execution time of instructions are *context dependent*. In a simple architecture, we are able to make an abstraction of tasks and instructions as their behaviour is independent of each other. However, these assumptions are unfortunately not valid anymore with more advanced processor architectures. The behaviour is influenced by the state of the hardware, and thus by the context of the previously executed tasks and instructions [35]. These platform-related influences, such as caches, pipelines, branch prediction, etc. are discussed in Chapter 3.

Lastly, hardware optimisations for execution time improvements could result in *timing anomalies*. These anomalies are influences of local execution times of instructions that impact the global execution time of the entire task in a non-intuitive way [34], [41]. Therefore, it is a false assumption that a local worst-case transition with conditional branches will resolve into the global WCET [42]. A timing anomaly takes mostly place during speculative and scheduling processes. Anomalies that manifest from speculation are found in cache memory.

Cache memories are small memory components with short access times close to the CPU that are introduced to improve the average execution time and reduce the energy consumption. When the processor requests an instruction or data, it will first check within respectively the instruction or data cache if it is present. A cache hit occurs when the data is found. In contrast to the slow main memory, the data is rapidly returned to the processor in a few clock cycles. The maximum cache size is limited to a few MBytes to keep the cost affordable due to the high cost per bit of cache memory. As the entire program never fits in the cache, a cache miss occurs when the requested data is not available. A time penalty is associated with this event as the total time to fetch the instruction or data block is equal to the lookup time in the cache to detect the miss, plus the time to retrieve the data from the main memory. The limited cache size will result that another cached element will be evicted from the cache to create space to store the new element. Different eviction strategies exist to select which element to remove from the cache memory first, as described in [43]. For example, when the speculative execution of a conditional statement has prefetched the wrong branch of instructions, the processor has to undo all steps, and switch to the other branch. This impacts the state of the hardware, such as the cache memory; and results in a high execution time penalty. When we examine the example in Figure 2.2, we have two instructions I_1 and I_2. In the first scenario we observe that I_1 is available in cache memory. While the processor waits for the conditional statement to resolve, it starts prefetching one of the branches.

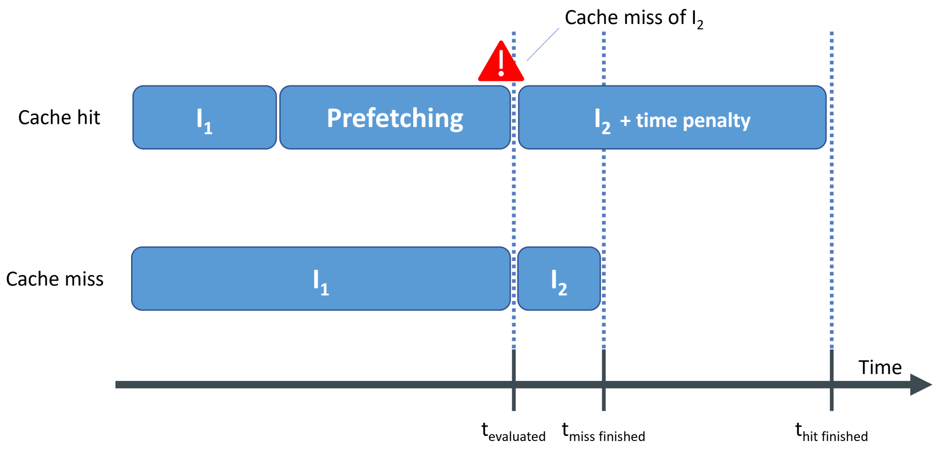


Figure 2.2: Timing anomaly caused by speculative execution. The scenario with a cache hit has a higher global execution time compared to the scenario with the initial cache miss.

After evaluation of the condition at $t_{evaluated}$, the speculative execution has taken the wrong branch. This adds a significant time penalty to instruction I_2, as it has to undo all prefetched calculations and I_2 needs to be fetched from main memory again because it was evicted from cache. In the other scenario, we notice that a cache miss of I_1 will last until $t_{evaluated}$, and thus no speculative execution has been taken place. The execution time of instruction I_2 is significant shorter, as the instruction was not evicted from cache. What we notice is that the local execution time of a cache hit of instruction I_1, that is shorter than a cache miss; will result counter-intuitively in a worse global execution time.

Timing anomalies are also possible during scheduling, as reported by Graham [44]. When different sequences of instructions or tasks are scheduled on concurrent resources (e.g. processor cores, pipeline units, etc.), it could result in scheduling patterns of instructions/tasks that have high local worst-case performance, but on global level have a better worst-case performance. The example of Figure 2.3 shows two scenarios of two tasks (i.e. A and B) that need to be scheduled on the resources α and β. While instruction $A_{\alpha 1}$ has a longer execution time in the first scenario, we see that the global scheduling time of both tasks is shorter compared to scenario 2.

In industry, many developers employ traditional physical measurements techniques to gain insight in the WCET behaviour of their systems, using probing software and/or external measurement equipment [45], [46]. Nevertheless, a measurement-based approach is a tedious and error-prone process that has no guarantees to find a tight upper bound.

WCET analysis is a research domain that has been explored for many years. In the SOTA, a lot of focus is put on adopting static methodologies to obtain safe and tight upper bound predictions. Therefore, correct approximations from the platform (and all of its components that have an impact) needs to be acquired. These interactions are so complex that entire research projects are dedicated on analysing the impact of single components, such as cache analysis [47]–[52], pipelines [53]–[55], and shared buses [56], [57]. Each of these analysis techniques creates an abstract interpretation of the hardware interactions

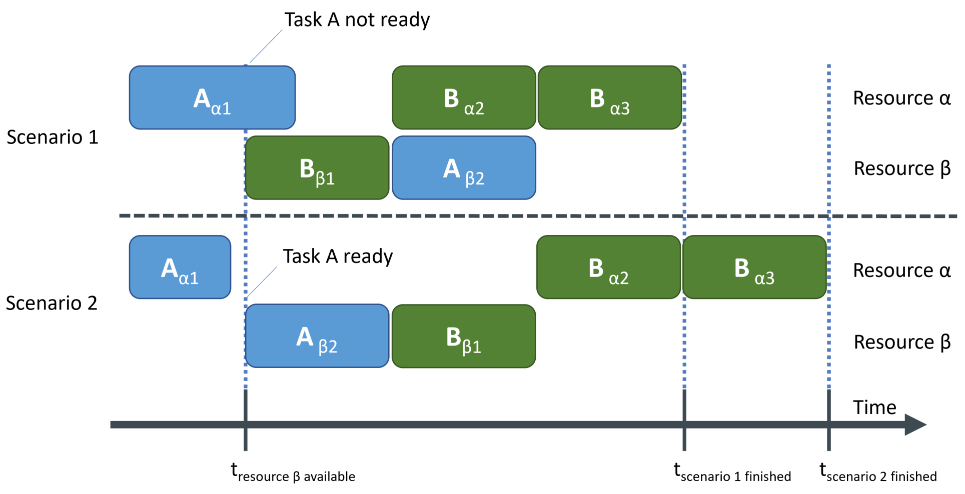


Figure 2.3: Timing anomaly caused by resource scheduling. Scenario 1 has a shorter global scheduling time compared to scenario 2 with shorter local execution times for both tasks A and B individually.

and their state, as they are context dependent (i.e. execution history). Eventually, this results in an occupancy state of the component for each path in the code flow. To estimate the upper bound, we need to calculate the invariant between the occupancy states of the different component analyses [35]. This low-level analysis provides insight in the resource consumption of each instruction based on the processor behaviour model. Nevertheless, this model needs input from the high-level, dynamic behaviour of the software task that is running.

Gaining insight in the dynamic behaviour of the program is performed through control flow analysis of the code, such as which branches are taken, which functions are called, or how many times a loop is iterated. Different approaches exist to obtain this information through (manual) annotations [58]–[60], or automatic flow and value analysis [50], [52], [56], [61]–[63]. The final result of this analysis is an annotated syntax tree that contains the different code paths and the derived flow facts, that is used to calculate the upper bound [35].

The final step is calculating the upper bound estimation of the WCET. Therefore, all timing- and flow-related output of the high- and low-level analysis are combined. Different methodologies exist to estimate the bound on the resource consumption. In literature, we distinguish three types of approaches: path-based [64], tree/structure-based [65] and Implicit Path Enumeration Technique (IPET) [66], [67]. The latter one was first proposed by Li et al. as an Integer Linear Programming (ILP)-based methodology to determine the WCET [66]. This technique is one of the more prominent used by many researchers. The goal of the IPET is to combine the timing information with the flow facts to obtain sets of arithmetic constraints that can be resolved, for example with ILP.

A different approach to estimate the WCET is through running a simulation of the program on an abstract processor model. The simulator will execute the code but without any input. The semantics of data types are extended with the state ‘unknown’, and therefore it needs to operate with partially unknown execution states. All static analysis stages are performed during the symbolic simulation [68], [69]. A major drawback of this methodology is the computational complexity that is proportional to the actual execution time of the task. A simulation is significantly slower compared to native code execution, and thus requires much time to perform [35].

2.1.2 Worst-Case Energy Consumption

The base formula to calculate the electrical energy (Joule: J) is the integral of the power draw over a time interval: $E = \int_{t=0}^{T} V(t)I(t)dt$. For the WCEC, we are interested in finding the maximum amount of energy a given piece of code will consume on the target platform. Therefore, we need to find the path in the code which would result in the highest energy consumption. One could think that the WCEC is equal to the longest execution path, as energy is the power draw over time. However, this naive conclusion is definitely not sound. In the context of execution time, the longest code trace, for example, does not guarantee the worst-case scenario. This is caused by timing anomalies that are introduced by the hardware [34], such as cache misses, out-of-order execution, etc. [30]. These anomalies also apply for the energy consumption. The total energy consumption is determined by instruction-specific silicon that is activated during the different stages of the instruction pipeline and performance enhancing hardware, such as cache memory and branch prediction. There is no guarantee that the longest execution path results in the highest energy consumption or vice versa. Experimental research has showed that no direct correlation exists between execution time and energy consumption [4].

The power draw of the hardware consists of two parts: static and dynamic power dissipation [4], [70]. The static component is the power that the hardware dissipates or 'leaks' when it is turned on in a stable environment, e.g., constant voltage, clock frequency and temperature, without considering the state or in- and outputs of the system. The energy consumption of the static power component is therefore directly proportional to the execution time of the program in an ideal environment. The dynamic component comprises the power dissipation of the activity on the hardware due to the code, changing states and I/O on the system. The dynamic power usage in the processor is the result of the switching and retaining the state of the circuit logic, such as registers, data buses, clock tree, Arithmetic Logic Units (ALU), etc.

According to Morse et al. [70], determining the dynamic power consumption is not a trivial problem. For execution time analysis, the time can be quantized to the number of clock cycles it takes to execute. This approach results in a set of discrete execution time possibilities. When we shift to energy, the power consumption is added to the equation. Power cannot directly be quantized as execution time to the clock period but is correlated to the number of transistors and their probability to change depending on the data for each operation. The complexity to statically calculate the energy consumption rises tremendously as the consumption for a single instruction varies on the Hamming distance, i.e., number of flipped bits, of each register during operation, and thus is dependent on the data input and its state. The state space of possible switches in hardware that we need to keep track of during analysis is equal to the powerset 2^S with $|S|$ equal to the number of transistors in the processor. Therefore, Morse et al. [70] concluded that this 'Circuit Switching Problem' is an NP-hard problem intractable to solve. While some research has claimed that the dynamic component of circuit switching does not contribute a significant part in the total energy consumption [4], [71], other researchers have noticed that the switching due to data input has a substantial impact of the observed consumption [70], [72] when solely looking at the energy consumption of the processor. Nonetheless, the energy consumption of the processing unit is sometimes only a fraction of the total energy consumption of the embedded systems depending on its peripherals [73], such as sensors, and most definitely wireless transceivers. The fluctuations in the energy usage of instructions due to data will become so small that it could be perceived as noise.

In the SOTA, we see lots of research on creating energy models of embedded processors, such as the ARM Cortex-M, XMOS xCORE, AVR, LEON3, etc. A first approach by Tiwari et al. [74] proposed a measurement-based analysis technique to quantify the power usage per instruction and the cost of switching between instructions. The model is then able to estimate the energy consumption of a code trace. This model was further extended by Steinke et al. [75] by including the switching activity on the data buses of the processor as the toggling bits take a major role in the energy consumption of the processor. In the EU ENTRA project [76], they investigated an energy transparent methodology for energy-aware development by using static analysis techniques for energy consumption estimations at compile time. In a first stage, they applied the static approach on the instruction (Instruction Set Architecture (ISA)) level [77], and later extended it on the block level of the LLVM compiler [78]. All these methodologies focus on estimating the energy consumption but are not looking at WCEC.

In the work of Jayaseelan et al. [4], they pointed out the need of taking energy constraints into account for mission critical battery-operated systems. In order to provide these guarantees, we need insight into the WCEC behaviour of computing tasks on the system. Compared to the work mentioned in the previous paragraph that focuses on average-case energy consumption, the research of [4] presents a static analysis approach that estimates the WCEC. Their methodology starts with estimating an upper bound for each basic block in the code. This upper bound is the sum of static leak power and the dynamic energy consumption of each instruction. Next, an ILP solver is used to find the path with the highest energy consumption based on the CFG of the program.

Research by Wägemann et al. [79] uses the ILP technique combined with symbolic execution and genetic algorithms to create a hybrid tool named 0g for WCEC analysis. Based on requirements, the tool selects the best approach to limit the analysis duration, and if it should under- or over-approximate the upper bound. The static analysis tools are able to use absolute energy models that provide exact numbers on the energy cost of an instruction. However, creating such a model is difficult to achieve due to non-deterministic behaviour of the hardware, or not having access to the implementation details of the architecture. On the other hand, the 0g tool is able to estimate WCEC based on relative energy models with genetic algorithms to determine valid input sets that trigger the worst-case path, but this will not provide absolute WCEC. These input sets are used to perform physical measurements on the device to obtain a WCEC upper bound.

In Table 2.1, we present a summary of the related research we discussed in this section and how our proposed work is novel in comparison.

2.2 Analysis Methodologies

In order to determine the resource consumption of software tasks, an analysis strategy is required to obtain an upper bound on the WCRC. In the state of the art, there are three

Table 2.1: Overview of different methodologies from related research in comparison to our work. R1: Altenbernd et al. [80], Bonenfant et al. [81], Huybrechts et al. [27] and Bachard et al. [82]; R2: Tiwari et al. [74]; R3: Steinke et al. [75]; R4: EU ENTRA project [76]; R5: Jayaseelan et al. [4]; R6: Wägemann et al. [79]; R7: our work; ^{1} Hardware model is required; ^{2} Average-Case Energy Consumption; ^{3} LLVM's intermediate representation; ^{4} Fallback strategy if the hardware model is not available; ^{5} ARTIST2 Language for WCET Flow Analysis.

	R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**
Resource	WCET	WCEC	WCEC	ACEC^2	WCEC	WCEC	WCEC
Approach	Static/ Hybrid	Measur.	Hybrid	Static	Static	Hybrid	Hybrid
Target Platform	All	All	$+/-^1$	$+/-^1$	$+/-^1$	All^4	All
Code Level	ALF^5/ ISA/ Source	ISA	ISA	ISA/ LLVM^3	ISA	LLVM^3	Source
Machine Learn.	Yes	No	No	No	No	No	Yes
Early-Stage Predictions	Yes	No	No	No	No	No	Yes

main WCRC analysis methodologies, namely the static, measurement-based and hybrid approach [83].

2.2.1 Static Analysis

The static analysis uses models of the software and the target hardware to obtain an accurate upper bound of the WCRC [4], [47], [79], [83]. This approach is able to obtain these bounds without actually executing the code itself on the target platform. The concept of solving recursive equations to determine a cost function for the resource consumption of an application dates back to the work of Wegbreit [84] in 1975. In his research, he proposed a methodology to derive the application behaviour with closed-form expressions based on those properties of the (input) data that are most relevant to the performance of the system, such as input size. His work has been extended upon by other research [85]–[88].

In general, the analysis consists out of four steps [35]. Firstly, a value analysis determines the value ranges and memory addresses of the data in the CPU registers for each point during operation. This data provides valuable insights during the second step, i.e. control-flow or high-level analysis. The different flows of the application code are represented by the CFG, or tree-based representation [89]. Infeasible paths can be excluded using the data from the value analysis. The next step is to determine the behaviour of the processor itself with a low-level analysis. The resource consumption of each instruction and hardware interaction is calculated based on the hardware model. Lastly, a bound analysis uses the control-flow analysis and the resource consumption from the processor-behaviour model to estimate the final WCRC, i.e. upper bound.

Detailed models of the application and the hardware are required to obtain sound results with a low upper bound. However, these hardware models are in most cases not publicly available by the manufacturer to protect their intellectual property [35], and therefore require enormous engineering efforts to create. Moreover, the introduction of optimisation techniques, such as caches, branch prediction, speculative execution, hyper-threading, etc., further complicates the predictability of the platform, which is explained in more details in Chapter 3.

Academic research in the domain of WCET static analysis has resulted in the creation of various tools, e.g. SWEET [90], OTAWA [91], BOUND-T [92], Heptane [93] and Chronos [94]. This methodology got eventually adopted by industry in commercial tools to validate the timing behaviour, such as AbsInt's aiT WCET Analyzer [95]. In the domain of WCEC, the available static analysers are mostly academic tools, e.g. SysWCEC [96], 0g [79] and platform specific tools [4], [76]–[78]. A commercial static energy analyser tool from AbsInt called 'EnergyAnalyser' is in development and currently supports two embedded architectures, i.e. ARM Cortex-M0 and LEON3 [97].

2.2.2 Measurement-Based Analysis

The measurement-based approach, also referred to as dynamic analysis, consists of performing physical measurements on the system to obtain estimates on the bounds of the resource consumption. In this method, a limited number of different input sets are provided to the system. The measurements will follow a certain distribution as shown in Figure 2.4. The distribution is used to acquire an estimate of the WCRC, which we call the upper bound. This solution is computationally acceptable as the number of measurements is decided by the researcher. Nonetheless, it often becomes infeasible to exhaustively test complex systems as the number of inputs and state space are too large to fully cover. By measuring an arbitrary limited number of inputs, it is therefore never guaranteed that the real WCRC is triggered resulting in an underestimated upper bound! The precision will drop dramatically when the number of measurements decreases as not all system states are covered by the given input sets. Therefore, safety margins need to be taken into account when deducing the upper bound from the measurements to ensure safe boundaries for the resource consumption [35], as illustrated in Figure 2.4.

An estimated boundary is *safe* only if that bound is more pessimistic than the actual corresponding outer limit, i.e. upper bound needs to be equal or larger than the WCRC. If the estimated upper bound would be smaller than the worst-case scenario, it could result in unintended behaviour of the system, such as losing power during operation for batteryless devices or life-threatening accidents in CPS. Nevertheless, the upper bound should also be *precise* as overestimating the WCRC with lots of pessimism would result in over dimensioned hardware that increases the costs of the system.

2.2.3 Hybrid Analysis

The hybrid analysis combines both the static and measurement-based technique into one methodology [26], [98], [99]. The hybrid model splits the source code of the task into a set of smaller components which we call *basic blocks*. Each block resembles a trace of consecutive instructions which has exactly one entry and one exit point in the flow [83]. In our work [26], [27], we extended on this principle by making the size of these blocks variable. We call them *hybrid blocks*. These blocks can contain a single instruction up to entire functions or programs. Nonetheless, each hybrid block still needs to have exactly one entry and one exit point. A balance between the static and dynamic analysis layer is obtained by changing the block sizes. The first step in this analysis is to measure the execution time of each block. The second step is to statically combine all results

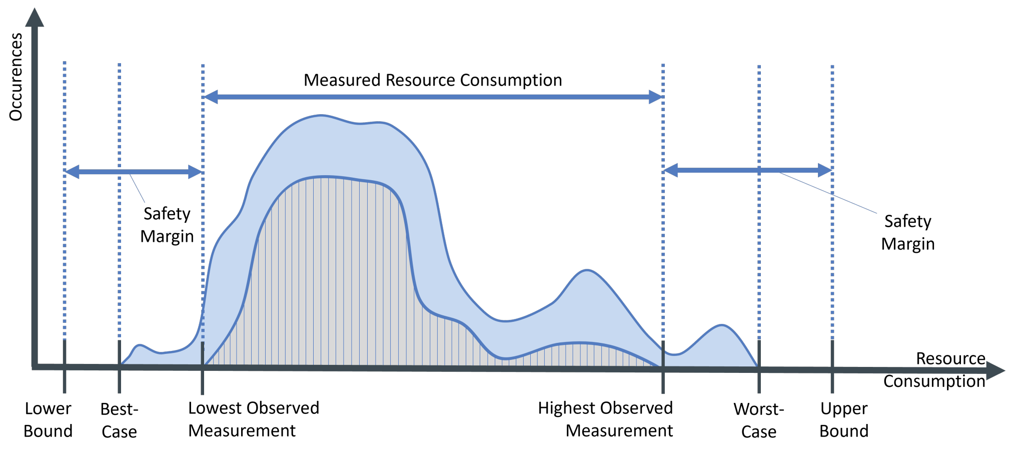


Figure 2.4: Measured resource consumption distribution of resource usage for a given task. The lower curve shows the distribution of the observed resource consumption during measurement with the lowest and highest observed measurement. The upper curve illustrates the actual resource consumption of the task. A safety margin is used to guarantee safe boundaries.

to acquire an estimate of the WCRC. The challenge is to find a balance between the computational complexity of the static layer and the precision of the measurement-based layer.

Research on hybrid WCET analysis has resulted in different tools, such as TimeWeaver [100] and Rapita Systems' RapiTime [101]. Based on this methodology, we have created an extension on the COBRA tool to perform hybrid analysis resource analysis [26]. The implementation of this extension is explained in Appendix A.

2.3 Benchmark

In order to validate and compare the accuracy of tools and methodologies with others we need a baseline to compare to. Therefore, we make use of benchmarks. The accuracy A of an analysis tool is the division between the reported bound of the tool divided by the actual WCRC. This result indicates the closeness of the reported bound with the actual baseline of the analysed task [73]. An accuracy with $A > 1$ shows an overestimation of the upper bound as a result of pessimistic assumptions during analysis. The opposite when $A < 1$ indicates an underestimation of the upper bound which is detrimental in the context of WCRC analysis.

Benchmarks used for computing systems are collections of test applications (i.e. source code) in order to compare the relative performance of systems/components with each other. A wide variety of benchmark suites are available on the market. These benchmarks are collections of domain specific workloads, such as EEMBC SecureMark testbench for testing the performance of encryption algorithms or the PassMark CPU Mark to test the overall speed of CPUs.

In the domain of WCET, specialised benchmark suites exist. The Mälardalen WCET benchmarks is one of the first suites to benchmark the performance of WCET analysis tools [60], [102]. The focus of this benchmark lies on the performance of the program flow analysis. The MiBench is another WCET centred benchmark that targets the embedded controllers [103].

A benchmark we use in our research is the TACLeBench. The TACLeBench is a benchmark project of the TACLe community to evaluate timing analysis tools and techniques (i.e. WCET) in order to compare their performances [60]. This benchmark is an open-source collection of over 50 benchmark programs composed from different research groups and tool vendors [60]. The advantage of this benchmark suite is the collection of benchmark applications that are fully self-contained in only one source file without any system specific dependencies. Therefore, each program is easily deployable on different hardware architectures. Additionally, all code is annotated with different flow facts [60]. These flow facts provide hints on the entry point of the application and the min-max number of iterations for each loop, i.e. loop bound. The TACLeBench suite is divided into five classes [60]:

- **Kernel**: synthetic benchmarks with small kernel functions;
- **Sequential**: large function blocks, such as encoders/decoders, graph search, cryptographic algorithms, etc.;
- **Application**: benchmarks derived from real life applications (e.g. lift controller, powered car window);
- **Test**: artificial test application to stress test WCET analysis tools;
- **Parallel**: applications with multiple tasks that can be run in parallel.

Within the context of WCEC, academics have created benchmark suites to evaluate energy consumption on embedded devices. The BEEBS benchmark developed by the University of Bristol and Embecosm is a collection of programs that are selected for energy consumption analysis of embedded platforms [104]. Another tool used in research is a benchmark generation tool called GenEE [105]. This generator creates custom programs by selecting pseudo-random generated code of patterns to evaluate static analysis for energy-constrained CPS. By carefully selecting these code pieces, the tool wants to retain knowledge of the system's state in order to obtain the worst-case flows to derive the WCEC of each generated set. To achieve this, they assume that the WCEC path is equal to the WCET path by applying branch overweighting [105]. This means that instructions that could lead to timing anomalies, e.g. cache misses, pipeline halts, etc., are compensated with sufficient additional instructions in the other branch.

2.3.1 WCRC Performance Comparison through Benchmarks

As an easy to implement alternative to WCRC analysis methodologies, we propose a practical benchmark-based methodology to assist engineers in taking decisions during the hardware exploration phase of the design. The basic principle is to benchmark each hardware platform of interest according to a set of selected test applications that closely resemble the characteristics of the target code base that needs to be deployed on the hardware. The resulting metric provides a relative indicator of the performance for each tested platform according to the composed test set.

Current benchmark suites consist of a wide collection of test applications covering a wide range of different coding paradigms (e.g. control loops with lots of branching, large stacks with recursion, memory intensive access to provoke cache misses, etc.), and assign one number on the performance of the platform to relatively compare the systems with one another. However, each hardware optimisation feature has its strengths and weaknesses depending on the actual code that will run on the platform. For example, caches greatly improve the average code execution time, but when lots of cache misses occur, the penalties will add up and result in a worse worst-case performance compared to systems without cache memory. In order to compare different platforms, we must consider performing comparative studies with a collection of tests that closely represent the actual code features instead of generic tests covering irrelevant cases.

A wide variety of benchmark suites exists on the market to evaluate the hardware performance. Each of these suites are a collection of tests composed to evaluate a specific application domain, such as power efficiency, multimedia, networking, multi-core, floating point operations, generic purpose computing, etc. These tests provide good insights if the target application is related to the benchmark domain, e.g. EEMBC SecureMark testbench for testing encryption algorithms. As most of these benchmark suites are free to use, they are not always open source, and thus we cannot observe the actual code that was used as workload during testing.

For open-source benchmark suites, we have direct access to the source code of the workloads, such as EEMBC Coremark, TACLeBench, Dhrystone, etc. Based on this source code, the characteristics of each workload can be described as a set of features. When the performance of each test is determined for each platform in the comparison, we are able to compose a performance function based of those tests with a feature set that closely resemble the characteristics of the actual code base. At this point, the engineer is responsible to select those tests that would resemble the actual load on the target platform. The selection can be performed with a visual inspection based on the expertise of the developer. However, a set of 'guidelines' could be provided that assist the engineer in the decision-making process. This list of guidelines is preferably composed based on hardware and software KPIs that influence the resource consumption of interest. These metrics can be based on software characteristics that impact the resource consumption due to platform related features, e.g. branch prediction, caches, etc. We are able to formulate these metrics into questions that the engineer can use to classify its software to select a set of appropriate test applications. These questions could be as follows, "Does the control algorithm contains many nested conditional statements?", or "Does the code uses dynamic memory allocation?". Another approach is to refine these questions by quantifying the metrics. This will assist the engineer in objectively selecting the most fitting benchmark sets instead of relying on subjective opinions on those questions. Examples of quantifiable metrics are, the level of the deepest nested conditional statement, number of variables/arrays, array sizes, loop bounds, etc. These metrics are also easily obtainable by automated parsing tools, such as the COBRA framework presented in Appendix A. In the following chapters (Chapters 3 and 4), we take a closer look into the influences of the platform, and the coded software logic of the application.

2.4 Early-Stage WCRC Predictions

Many existing WCRC tools perform the analysis on compiled binaries for specific hardware platforms. This approach requires the developers to have a compilable version of the application and the physical hardware platform available before any estimate can be

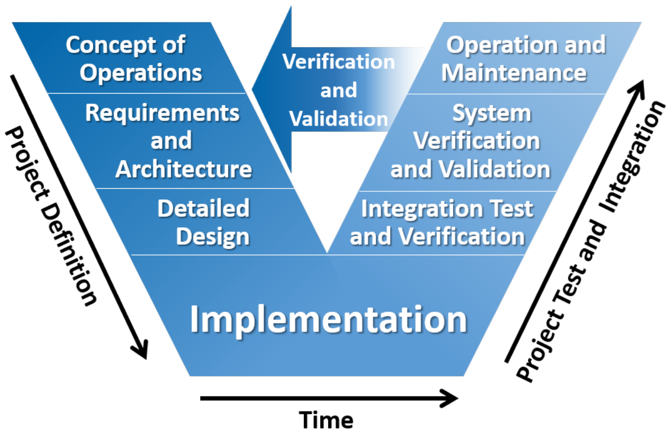

Concept of Operations
Requirements and Architecture
Detailed Design
Implementation
Verification and Validation
Operation and Maintenance
System Verification and Validation
Integration Test and Verification
Project Definition
Project Test and Integration
Time

Figure 2.5: V-model - Development and verification process model

provided [35], [106], [107]. As a result, it becomes difficult to acquire early insight of the WCRC during development. Additionally, systems, such as hard real-time systems, have strict requirements on these resources as failing these requirements could lead to failing behaviour with potentially disastrous consequences. To ensure that all requirements are met during development, two scenarios are possible.

On the one hand, system designers assume really pessimistic results of the upper bound to compensate for all errors and assumptions that were made during the analysis. This results in the use of overqualified hardware which will increase the cost of the final product. Whereas small cost savings on better dimensioned hardware will result in huge savings in mass production.

On the other hand, underestimating the final WCRC during development leads to financial losses when custom developed hardware does not appear to be sufficient enough to schedule the software tasks [108]. The costs associated with making changes to the design rises tremendously during the product development phase. In [109], it is stated that the cost of making changes during the conceptualisation phase of the product equals to a factor $10\times$, while making the same changes could increase the cost to a factor $50,000\times$ in the production phase.

The main reason that causes the previous scenarios is the lack of insight on the WCRC in the early stages of the development cycle. When we look at the V-model development process, we start with defining the project requirements and add more details to the design with each step, as shown in Figure 2.5. In this model, the code development will only start in the implementation phase at the bottom. As a result, the first opportunity to gain insight in the WCRC is after the project details are already fixed. In the case of an underestimation of the upper bound, the development process has to move up again in the V-model to adapt the design.

Measurements on the physical device require long and high effort to accomplish and can only be performed in the validation stage of the design. A better approach would be to gain insight early on in the development process without tedious measurements to 'fail fast', so that the developer has the opportunity to fix the design much faster and reduce costs. Previous research has proposed new methodologies to gain early insight in the domain of execution time. Altenbernd et al. [80] used a linear time model with Simulated Annealing (SA) algorithms that translates code into basic instructions for which the timing is determined. This technique is extended in other research by using ML. Bonenfant et al. [81] approximated the WCET by training a neural network using worst-case event counts which are matched to a WCET estimate for the target architecture. The research of Bachard et al. [82] has extended on the hybrid analysis principle with ML on a lower level, i.e., machine code, to incorporate cache behaviour and tackle the training data issue by automatically generating code samples.

2.5 Conclusion

In this chapter, we discussed the importance of resources in constrained embedded systems. Systems with hard constraints require insight in the WCRC to guarantee correct deterministic behaviour. In this thesis, we take a closer look at two types of resources: execution time and energy. Three main types of analysis methodologies exist in order to predict the WCRC of a software task on a given target system. The choice of analysis technique is a trade-off between precision and computational complexity. To determine the performance of these techniques, or the predictability of a system, we are able to employ benchmarks to validate and compare them. We proposed a practical benchmark-based methodology that is able to assist engineers in evaluating hardware platforms based on relevant test cases. Finally, we position the WCRC analysis in the development process of such system and look at the possibility to gain insights of the resource consumption in the early stages of the development.

Chapter 3

Platform and OS Influences on WCRC

The resource consumption of a software task is determined by characteristics of the system. Different inputs will result in triggering different traces in the program. However, consecutive equal input sets could have distinctive results as the state of the entire system differ, due to anomalies introduced by the hardware [34]. In this chapter, we will discuss the following prominent platform- and OS-related system components that influence the resource consumption of software tasks running on an embedded controller:

1. Compilers and instruction set architectures
2. Clock speed
3. Memory access
4. Scheduling and pre-emption
5. Caches
6. Shared resources
7. Pipelines

3.1 Compilers and Instruction Set Architectures

The compiler is a computer program which translates software code of a higher-level programming language to a lower-level language. In this context, we are interested in the translation to binary code that the processing unit on an embedded controller is able to interpret. This step is essential as computers are only able to understand a limited set of predefined instructions, such as read register, add two values, write register, etc. Based on this instruction set, the compiler has to translate each abstract code instruction in the higher-level language to a combination of basic low-level instructions. Depending on the available instructions in the set of the CPU, one instruction in your source code could require dozens or hundreds of basic instructions to perform.

In the end, the final execution time is (partially) determined by the number of actions (basic instructions) that the CPU performs. Each machine instruction takes one or more compute steps or clock cycles to execute. The number of cycles required to perform an instruction depends on the hardware implementation of the instruction set. Some specialised hardware has an optimised ALU to perform certain calculation in just one clock cycle. For example, vector operations are commonly used in computer graphics. By reducing the number of clock cycles needed for an operation, the higher the throughput becomes, which results in less stalling of the processor.

The implementation of these instruction sets depends on the architecture of the processor. The most well-known instruction set architecture classifications are the Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) processors. A CISC architecture has an extensive list of specialised instructions. The main concept behind this approach is to provide one instruction to perform a certain (complex) operation. As the instructions more closely matches high-level programming constructs, the code density of the binaries increases as fewer instructions are needed to be encoded [110]. Therefore, more instructions can be stored in the limited high-performance caches. The hardware could be optimised to execute complex operations in few clock cycles with specialised hardware. However, this increases the architecture complexity dramatically. An example of a CISC architecture is the popular x86 processor family.

The instruction set of a RISC architecture focuses on simplifying the overall design and implementing instructions that are optimised in instruction cycle times (and not in the number of available instructions!). The average cycle time of a RISC instruction is near one cycle. To accomplish this, the 'complex' CISC instructions are split in separate fetch, compute and store operations. As each instruction is optimised in cycle time, it becomes feasible to introduce a pipeline that chains the different stages of consecutive instructions to maximise the throughput. Many processor architectures are based on the RISC approach, such as ARM, PowerPC, Atmel AVR and MIPS processors.

Each architecture will require individual binaries as the instruction syntax, available instructions (opcodes) and registers, etc. differ between them. As a result, specialised compilers for each architecture are used to generate binaries of source code. The execution time is highly dependent on how the code instructions are translated into machine operations. Therefore, different compiler toolchains (of different vendors) for the same architecture can have various results in the final binary due to their implementation and applied optimisations.

Next to implementation differences, most compilers have extensive configuration parameters. These configurations mostly have impact on the optimisations the compiler will perform on the code to increase the overall performance (throughput, code density, etc.). However, while these optimisations improve the average case, they may induce adverse consequences for the worst-case scenarios. The most prominent optimisation configuration is the 'optimisation level' option (-O). For example, the GCC compiler has five optimisation levels from which the programmer may choose. These levels range from -O0 (zero – no optimisation) to -O3 (full optimisation) and a fifth option (-Os) to optimise to on-board size (instructions and data). Each optimisation level will induce a list of optimisation techniques. Switching between these levels will have major impact on the execution distribution of the program! Increasing the optimisation level of the compiler will in general reduce the number of instructions, and therefore lower the average energy consumption in most cases [111]. Nevertheless, there are no guarantees that the WCEC will not be negatively impacted by these optimisations.

Another popular compiler toolchain is LLVM, which is used in other research for WCRC analysis [75], [76], [78], [79], [105], [112]. LLVM is an open-source project with a collection of modular compilers and toolchains for code compilation [113]. One of the strong points of this framework is the LLVM Intermediate Representation (LLVM IR) that enables to have a higher-level abstraction of the actual code instead of working with ISA. Creating optimisation for LLVM IR makes it more generic to apply on a wide set of target hardware. In other work [76], [112], they created LLVM optimisers with the notion of energy consumption in order to gain insight in the WCEC and introduce energy-aware optimisations to code.

3.2 Clock Speed

In a synchronous CPU, all operations are synchronised with each other by a central clock indicating the pace. Each operation has a deterministic timing scheme (which is determined by the state of the system – e.g. cache hit requires X cycles, cache miss requires X + Y cycles). A clock cycle is the period between two pulses of a clock generated signal. At the start of a clock signal, all components start their operations. Before the next period starts, all components should finish their operation and be in a stable state ready for the next cycle. As a result, the maximum clock speed is limited to the slowest component in the chain. However, delays and/or pipeline stalls can be implemented for slower non-continuous operations, e.g. fetching data from slow main memory.

Until now, we have used clock cycles as a 'virtual' measurement of time as they are integer numbers that are easy to perform calculations on and comparisons with. However, CPS have hard real-time deadline in the physical world expressed in (micro-)seconds. The actual execution time (s) is obtained by multiplying the clock period (1/Hz) times the number of clock cycles. It is important to note that the clock frequency in some processors are adjustable, even dynamic (CPU throttling/boosting). By increasing the clock speed (overclocking), the execution time can be reduced. However, overclocking is limited by the physical characteristics of the electrical components (transistors, etc.) of the processor and will result in higher power consumption, increased heat production and decreased lifespan of the components [114]. Additionally, overclocking will not always improve the performance when the application heavily depends on data of slower peripherals, such as main memory, I/O devices, etc., that are not influenced by the changing clock speed, as the pipeline needs to stall its operation. To determine the WCET, it is safe to assume to have no speed boost (i.e. CPU boost disabled during benchmarking), as throttling heavily depends on the CPU temperature. Unless you have guarantees that the environment and CPU temperatures are within certain ranges and the thermal space of the system has enough margin to spike.

On the other hand, throttling (lowering) the clock speed is a commonly used technique in modern processors to improve the power consumption of the processor. It is important to note when measuring execution times in clock cycles (C) with a built-in clock register based on a non-constant clock frequency ($f(x)$) that the actual execution time (t) equals:

$$\int_0^t f(x)dx = C \tag{3.1}$$

$$F(t) = C \tag{3.2}$$

$$t = F^{-1}(C) \tag{3.3}$$

In low-power applications, energy efficient hardware has embedded sleep capabilities in which certain features of the embedded controller are deactivated up to a deep sleep mode where the CPU clock is turned off.

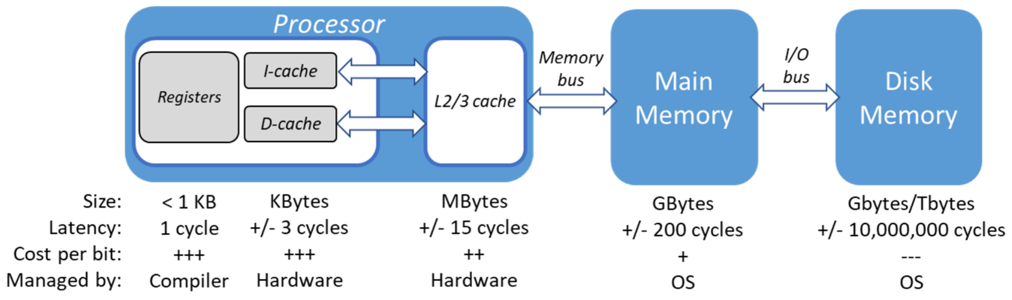


Figure 3.1: Layered memory hierarchy from main memory to registers annotated with the average size, access latency and cost per bit.

3.3 Memory Access

The structure of a program consists of a sequence of instructions and associated data on which operations are performed in line with the instructions. This information is stored in the main memory of the controller. This memory is slow relative to the high clock frequency of the CPU. Valuable clock cycles and energy are lost while the CPU is waiting for the requested data. Therefore, multiple memory layers are integrated into the hardware to improve the overall performance. The layers closest to the processor are the registries and the instruction/data-caches. These memory elements are extremely fast resulting in minimal latency for retrieving instructions and data. However, the size of this memory is very limited, due to the high costs of fast memory and the technical limitations of physical space to embed it in the proximity of the processing unit to guarantee low latency [47]. The next layers of memory become larger and cheaper to implement. Nevertheless, each layer comes with higher latency as every additional layer is slower to access. Figure 3.1 illustrates the memory hierarchy in a modern computer.

This multi-level hierarchy provides a good trade-off between speed, power consumption and storage capacity. During first start-up, as all caches and main memory are clear, data and instructions need to be retrieved from the persistent storage at the outermost layer. The stored content of each layer is managed by the compiler, hardware logic or the OS. The register utilisation in the processing unit is determined at compile time by the compiler. It determines which registers are used to store variables. The outer storage, such as main memory and disk memory, are managed at runtime by the OS. On both the registers and the OS managed memory, the software is able to acquire insight in the memory hierarchy. However, the different cache memory layers are managed by hardware itself and its existence/functionality is transparent for the software. In order to determine the WCET, the latency of retrieving data plays a major part as it could potentially stall the execution pipeline of the CPU.

In [115], they compared optimisation strategies for WCET and WCEC on instruction fetching and cache memory. During experiments they found that optimising the WCET performance does not guarantee any optimisation for the WCEC but could even aggravate performances. A better strategy is to use a combined optimisation approach with a WCET-WCEC trade-off analysis.

3.4 Scheduling and Pre-emption

Systems that are able to run multiple tasks or applications (with or without OS) need a controlling mechanism to plan all tasks on the available system resources. This mechanism that controls the execution of tasks is called the scheduler [116]. The scheduler determines which tasks are executed on the CPU and when. Each task is placed on a queue when it is ready for execution. The order in which these tasks get executed is based on a given priority to the tasks.

The scheduling algorithm can be categorized into pre-emptive or non-pre-emptive algorithms. Pre-emptive schedulers are able to switch between tasks even when the current one is not finished [116], for example round-robin scheduling and shortest remaining time first. A non-pre-emptive only switches tasks when the previous one is finished, such as shortest job first scheduling and first-come-first-served. By using a pre-emptive scheduling approach, the throughput of the processor can be increased. If a program is waiting for I/O operations, the processor can execute another task in order to avoid idling during the time waiting for the I/O operation. However, pre-empting a task can lead to misbehaviour when it happens during a critical section of the code. A critical section is a segment in the code which contain operations that share resources. During pre-emption, another task could manipulate the state of the shared resource leading to unexpected false results. These phenomena are called race-conditions.

In CPS where real-time behaviour is an important constraint of the system, a real-time scheduler is used. Each task in these systems has a timing constraint. Each task must respond within the given deadline imposed by the constraint. Therefore, the functional behaviour as the temporal behaviour needs to be guaranteed [117]. Depending on the type of real-time system, the consequences of missing deadlines have a different impact on the system as a whole:

- **Soft real-time**: systems where missing deadlines will result in a degradation of quality of the task's outcome. The system performance will decrease when too many deadlines are missed. Nevertheless, the system will not fail in the end. An example of a soft real-time system is a weather station with multiple sensors that is comparing the different inputs, such as temperature, humidity, etc.;

- **Firm real-time**: systems where missing deadlines will result in losing significant quality (up to zero) of the task's outcome. Nonetheless, missing deadline does not result in the system failing, but the performance drops when too many deadlines are missed. In the case of video streaming, missing deadlines of frames will cause skipping frames, and thus a jittering image. A frame loses its significance (quality equals zero) when it is over its deadline;

- **Hard real-time**: systems where missing deadlines will result in task outcomes with no value. Not being able to perform certain tasks in time will eventually lead to system failure or even catastrophic events, e.g. airbag system, Anti-lock Braking System (ABS) controller of a car, pacemaker, etc.

In order to schedule tasks in a real-time environment, the scheduler needs to know the deadline of each task and the WCET as it must try to find an optimal scheduling to run each task to completion before its respective deadline. Therefore, the WCET of a task needs to be known in advance. However, the scheduler has an impact/overhead of itself on the system, because the scheduler needs computational resources to evaluate and schedule tasks. Additionally, pre-empting a running task requires to copy the context (i.e. registers, program counter) onto the stack and recover another task's context. This is referred to as context switching [118]. The real-time scheduler should take notice of these phenomena when scheduling the tasks. Important to note is the scheduler itself needs processing time to perform the scheduling, and therefore trigger context switches.

The scheduling of tasks can be performed as periodic, sporadic or aperiodic. Periodic tasks are repeated every T time units, with T as a fixed time interval. Sporadic tasks are reoccurring tasks with an interval T that indicates the minimal time between two executions. Aperiodic tasks have no minimal reoccurring time but are able to be released at any given time based on an event or interrupt generated in the system. Events generated by the OS (software) and/or peripherals (hardware) needs to be handled by the scheduler. An interrupt is an alerting system that notifies the CPU of an event that requires immediate attention. When the interrupts get accepted, the context of the running task gets switched by the interrupt handler to trigger the appropriate Interrupt Service Routine (ISR). After handling the interrupt, the scheduler is back responsible for scheduling the next task. Based on the implementation of the scheduler, the scheduling process is driven by interrupts that are triggered by events and/or quanta. Event-driven scheduling reacts on events such as task releases, task completions and priority changes. This allows the scheduler to directly select a new task to perform. For quantum-driven scheduling, the scheduling process is triggered on fixed time intervals or time quanta (Q) [116]. These quanta are time slices in which tasks are scheduled. As a result, the latency of scheduling new tasks is higher compared to event-driven, however this approach is easier to implement.

3.5 Caches

Caches are small memory elements which are integrated close to the processor unit. The fast cache memory will fetch data and instructions of the main memory. As a result, it provides a tremendous improvement of the average execution time compared to the slower Random Access Memory (RAM) or storage devices. However, the fast cache memory has a high cost per bit and the space close to the processor is limited. Therefore, the maximum cache size is limited to a few MBytes or KBytes for embedded systems to keep the cost affordable. This allows for a limited number of instructions of a program that can be loaded into the cache. Each cache line contains a block of multiple data elements which includes the data or instruction of interest. These cached blocks are very efficient due to the sequential order of instructions in the program flow. This concept is referred to as the locality principle [47], [50].

When the processor requests an instruction, it will first check within the cache if the instruction is present. A cache hit occurs when the instruction is found. In contrast to the slow main memory, the data can now be rapidly returned to the processor in just a few clock cycles. In the opposite case, the cache will miss, and thus the instruction needs to be retrieved from a higher level of the memory hierarchy. The total time to fetch an instruction which is not present in the cache is equal to the lookup time in the cache to detect a miss plus the time to retrieve the data from a higher level. For instance, Level 2 (L2)/Level 3 (L3)-cache, RAM or Hard Disk Drive (HDD). Each cache miss adds an extra delay to the execution time. This results in a higher variance in the distribution of

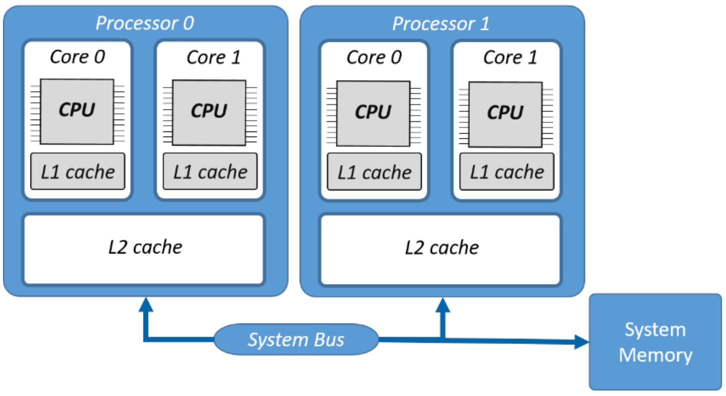


Figure 3.2: Shared caches in a dual-core, dual-processor system.

the program's execution speed [47], [119].

When looking at a multi-core processor, multiple layers of cache memory are introduced as shown in Figure 3.2. A first level of cache is placed close to each core. This level has a similar functionality as the single-core caches. A second layer of cache is shared between the cores to improve the performance of a multithreaded application working on the same data (cached memory blocks) or the communication between the different cores using these caches.

The complexity to calculate the influence of caches on multi-core systems increases tremendously in comparison to single-core systems. The different cores can fetch data simultaneously from the memory. A standard L2-cache shares the memory between the multiple cores of the processor. Each core has access to the entire cache memory. In order to calculate the WCET, each possible state of the cache must be known at each given point in time for all possible program traces. Since multiple programs with different context can be executed concurrently on different cores, all the influences have to be taken into account to obtain a sound approximation of the execution time bounds.

Storing data in the cache memory always is performed on block level. When access to data is requested that is not present in the cache memory, a fixed sized block that contains the requested data is copied. Copying data blocks into cache will improve the performance due to the principle of locality of reference. This principle is built upon two assumptions of temporal and spatial locality of references. Temporal locality defines that recently referenced resources are more likely to been referenced again soon, and therefore we want to cache those resources that have been referenced recently. In the case of spatial locality, the chance to reference a resource that is closely located to a recently referenced resource is higher. For example, the next instruction in the program has a higher probability to be executed compared to the instruction 50 places down the trace as there may occur jump statements modifying the program flow before arriving to that instruction. To exploit this principle, the referenced resource with its immediate neighbourhood is copied into cache as a cache block.

When a cache miss occurs, the required data must be transferred from the main memory to the cache memory. Since the size of the cache memory is limited, a previously loaded block has to be evicted. The selection of this block is done according to a replacement strategy. The choice of this strategy has an impact on the performance of the system. The most often used replacement strategies in modern architectures according to Paun et al. [43] are:

- **Least-Recently Used (LRU)**: The cache lines are ordered from most recently used to least recently used. When a cache hit occurs, the corresponding element will be placed at the front of the queue. On the other hand, when a cache miss occurs, the least recently used element will be evicted from the cache and the new fetched one is placed in front of the queue;
- **Pseudo-LRU**: This policy has similar concept as LRU. The selection procedure is performed with a binary tree structure. The advantage of this implementation is the smaller quantity of required status bits. Each cache line is provided on one of the ends of the binary tree. Each node of the tree has a binary status (0 or 1), which refers back to one of its two branches, respectively left or right. The path, that is constructed from the root to an end node, points to the next cache line that will be evicted when a cache miss occurs. When the cache hits, all nodes which are on the path to the accessed element will change their status bit, so that each node will point away from the last accessed element. As a result, the recently used element is protected from eviction at the next cache miss;
- **First-In First-out (FIFO)**: A FIFO policy (or round-robin) places each cache line in a queue. With every cache miss, the oldest added element in the queue will be evicted from the cache regardless of any recent access to that element;
- **Most-Recently Used (MRU)**: Despite the name this policy does not replace the most recently used cache line. Each element in the cache has a status bit. This bit is set to 1 when the element is retrieved. When the status bit of the last element different from 0 is set to 1, all other status bits are reset to 0. A cache miss will evict the first element with a status bit equal to 0 from the cache. Thus making place for the new cached element.

In order to determine the WCET, we need to know if the data/instruction access will result in a cache hit or miss. A cache analysis will determine the behaviour of the cache for a program on a set of inputs with a possibly unknown initial cache state [47]. Therefore, it calculates the probability if a certain resource is located within the cache.

As these replacement policies are implemented in hardware, it becomes difficult or even impossible to model the state of the cache at any given point in time. For example, the FIFO and MRU replacement policies are hard to predict as each hit and miss is not treated in a uniform way. In the case of FIFO, an element in the last position in the queue may be evicted right after being accessed when a cache miss occurs. The MRU on the other hand is heavily influenced by the 'last' bit that will be set to 1 and all others are reset back to 0. This asymmetry in the last bit makes MRU unpredictable. Nonetheless, the LRU policy has desirable properties as it treats the cache hits and misses in a similarly way, and therefore the cache state can be reconstructed rather quickly with every memory access. As the timing behaviour of the code depends on the state of the cache, it is important to note that starting from an empty cache will not result in the worst-case behaviour [120]! In order to incorporate the influence of the initial cache state, a technique called 'cache pollution' can be applied [121]. This approach will run arbitrary code before each measurement to acquire a non-empty cache state. As a result, more realistic execution states of the benchmarked software are achieved during measurement.

Different studies have shown that cache memory could take up to 50% of a MCU power consumption [122], [123], because of the high off-chip capacitance and memory storage size. The energy consumption of memory has been described by [123] as a combination of a static and dynamic component. The static component contains the static energy consumption the memory unit needs per cycle. The dynamic energy component is the sum of the energy consumption while retrieving a cache element, and the energy penalty from each cache miss. Tuning of the energy consumption of cache memory could be performed at design time (i.e. offline static cache tuning) by selecting the most optimal cache configuration using simulation; or by including optimisation mechanism to optimise energy consumption at runtime (i.e. online dynamic cache tuning) [124].

An alternative to ‘classic’ cache memory that is gaining popularity in embedded platforms is Scratchpad memory. This memory array maps memory objects in the last stage of the compiler, and thus is determined by the user or compiler during compilation [125], [126]. As the content (memory blocks) of the Scratchpad are known upfront, there is no need to check the availability of the data or instruction in its memory. This results in less circuitry needed to determine the cache hit/miss [125].

The predictability of Scratchpad memory for WCET analysis is more feasible compared to cache memory as the behaviour of Scratchpad memory is under control by software instead of the transparent operation of hardware-controlled memory. In addition, this memory type proves to be a low-power alternative with average energy reductions up to 40% [125].

3.6 Shared Resources

When introducing multi-core processors in real-time systems, all available resources need to be shared among multiple processes running on different cores. Therefore, the longest trace in software does not implicitly imply the worst-case path in the code [34], [127]. In order to get access to different resources, the cores are connected to those resources through shared buses. As these buses are shared, only one data stream can be sent at any time. In order to provide every resource access to the bus, an arbitration policy is put on the bus. This policy allocates time slots for each resource in order to communicate on the bus. Examples of arbitration policies are Time Division Multiplexing (TDM) or Round Robin, Lottery-Based Arbitration and Fuzzy Logic Arbitration [128].

The complexity of calculating the WCET for multi-core systems increases tremendously compared to single-core systems. The synchronisation mechanic used has a significant impact on the performance [129]. The different cores can fetch data simultaneously from memory. A standard L2-cache shares its memory between the different cores of the processor. Each core has access to the entire cache. As a result, the content needs to be protected for concurrent access to critical code sections to avoid race conditions. Locking the access to and from a resource is achieved by applying a mutex on a resource. This mutex will guard the access such that only one thread is able to read from or write to the shared resource. Locking any resource requires multiple steps on different levels, such as hardware context switching, the ring levels on the CPU (permission levels) and the locking instruction itself [129]. The overhead for locking resources could go up to a factor 50 when inter-process sharing is used that requires switching to kernel level, compared to having only one process for each thread, which is the fastest as no context switching is necessary [129].

In order to calculate the WCET, each possible state of the cache must be known for each point in time of all the possible program traces. Since multiple applications with different program context may run concurrently on different cores, all influences (e.g. preemption, resource sharing) need to be taken into account to obtain a sound approximation of the execution time bounds. In case of static cache analysis when no bounds or rules are applied on cache access, the methodology becomes infeasible to perform as an immense amount of states need to be evaluated.

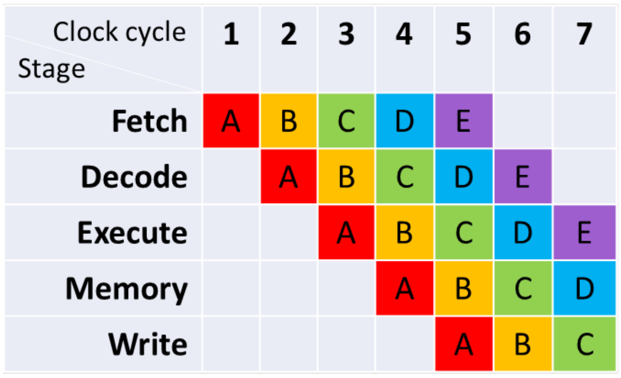
In [119], Hahn et al. propose a methodology called timing compositionality. This concept suggests that the processor architecture is decomposed in minor hardware and software systems, whose timing contributions are calculated independently from other systems. The upper bound of the execution time is then determined by all the timing contributions of each subsystem. In order to achieve compositionality in multi-core systems, Hahn et al. [119] present two different approaches. A first approach is to find a compositional analysis for existing architectures. The development of a sound compositional timing analysis becomes very complex when the available resources (e.g. multi-level caches) are shared among different cores. The analysis cannot be further decomposed through the dependencies that influence the timing distribution. Therefore, it is not possible to perform a compositional decomposition for every architecture. However, modifications can be applied to existing hardware to support decomposition. In [48], Chisholm et al. propose to partition the shared cache such that each core has access to an assigned part of the cache. This allows the shared cache to be decomposable, and thus making it possible to establish a timing distribution of the cache.

A second approach is to develop new hardware architecture which supports timing compositional analysis [130], [131]. These timing compositional architectures are decomposable into different subsystems whose timing distribution can be calculated straightforward. Nevertheless, it limits the benefits of having multiple cores, i.e. shared context between the cores [49]. The challenge is to design an architecture so that the performance of these systems is similar to those of current systems. However, no methodology has been found yet to perform a compositional analysis.

3.7 Pipelines

Each program instruction is a combination of different steps or stages that are performed. These stages include fetching instructions, decoding the instruction, executing the instruction, accessing memory and writing the results back to memory [132]. As each instruction involves the same stages to perform on dedicated parts of processor logic, it is possible to run these stages for different instructions in parallel. This results in instruction-level parallelism and provides higher efficiency of the processor as it tries to utilise each stage of the processor constantly. An example of multiple instructions being executed by the instruction pipeline is shown in Figure 3.3. This pipeline principle is comparable to a factory belt where each step of the process is performed on one dedicated stage on the belt and each unit passes each stage consecutively after one another.

By splitting the instruction execution into several stages, and by allowing to execute these stages in parallel, the overall throughput (average) increases tremendously. Additionally, some processors are able to start multiple instructions per cycle (e.g. superscalar



Clock cycle / Stage	1	2	3	4	5	6	7
Fetch	A	B	C	D	E		
Decode		A	B	C	D	E	
Execute			A	B	C	D	E
Memory				A	B	C	D
Write					A	B	C

Figure 3.3: 5-Stage pipeline with five instructions (A-E) going through.

processors) and some may even execute instructions out-of-order to further improve the throughput based on the availability of data [132]. However, these optimisations introduce pipeline hazards that introduces large penalties for WCET. We distinguish four types of pipeline hazards:

- **Data hazards**: when instructions depend on data that is not yet available to perform action on;
- **Resource hazards**: when consecutive instructions require access to the same resource;
- **Control hazards**: the use of conditional branches in code as the pipeline does not know with certainty which branch to take;
- **Instruction-cache hazards**: when fetching an instruction causes a cache miss that stalls the pipeline as it needs to wait for retrieving the instruction from main memory.

As these hazards occur, it could result in (1) stalling, or (2) flushing the pipeline. When data or resources are not available, because of a dependency or cache miss, the execution has to temporarily halt until the data or resource becomes available. During the pause of a stage, a delay or 'bubble' is introduced in the pipeline [132]. This bubble is an empty slot that travels along the remaining stages and produces no value for that stage it is located in. Each pipeline stall will introduce a new bubble that delays the entire chain of following instructions with an additional clock cycle.

To further improve the performance of instruction pipelines, new techniques are introduced to lower the chances of pipeline stalls or minimising the impact. In the case of conditional branches, the pipeline needs to wait for the result of the condition in order to determine which trace to continue. This technique is called branch prediction, a form of speculative execution [132]. The processor will try to predict which branch to take when a conditional statement is presented. If the guess was correct, the pipeline benefits from speed gain as it could continue ahead. However, when the guess was incorrect, the pipeline has to 'go back' and continue with the other branch. Therefore, the pipeline gets flushed (i.e. cleared) as all work performed for the wrong branch was irrelevant.

Another technique to minimize the impact of pipeline stalls is out-of-order execution [132]. The pipeline will decide in which order to execute all instructions by taking their dependencies into account. If data would be unavailable, due to a cache miss, etc., the pipeline could use another instruction to fill in the 'bubble', and thus maximising the throughput. Each instruction is queued for each different ALU until the target unit is free for execution. As each logical CPU unit has multiple execution units, the instruction pipeline is able to process two instruction streams in parallel that eventual merge in one set of execution units.

When performing a WCET analysis, all previously mentioned techniques add additional layers of complexity to the mix. For static analysis approaches, all these layers need to be predictable during their interactions with each other. By isolating all these subsystems first and taking their most pessimistic assumptions before combining them will result in a solution that is too pessimistic and non-feasible to create [53]. As for hybrid approaches, the sizes of the hybrid blocks play an important role in acquiring sound results [26], [27]. The interactions and states of each layer in the pipeline is determined by the previous and next instructions in the program trace. Just profiling atomic hybrid blocks in isolation does not guarantee the worst-case performance, as the state of the pipeline could be 'worse' during real program execution [53], [54]. This principal is referred to as 'timing anomalies' where local WCETs are not necessary part of the actual global worst-case scenario [34].

3.8 Conclusion

In this chapter, we discussed some prominent platform-related influences on the WCRC of software tasks. The composition of the hardware and software features of the system does not only impact the distribution of the resource consumption, but also impact the predictability of the worst-case analysis on the system itself. This is caused due to anomalies introduced by performance enhancing features, such as caches and pipeline optimisations. These features are focused on improving average resource consumption, but are detrimental for the WCRC, and sometimes even its predictability because of their non-deterministic behaviour. In the next chapter, we look at software-related influences on the WCRC.

Chapter 4

Software Influences on WCRC

The resource consumption of a software task is determined by the instructions that are executed on the embedded controller. In this chapter, we will discuss prominent software-related features that determine the actual resource consumption and look at the impact they have. Finally, we conclude each section with a concise list of defined qualitative questions that are based on software characteristics that would influence execution time and/or energy consumption with respect to platform-related features. These questions fit within the benchmark-based performance comparison methodology for WCRC as proposed in Section 2.3.1. The list of prominent software-related features that we discuss in this chapter is as follows:

1. Mathematical operations
2. Recursion
3. Loop bounds and nesting
4. Conditional statements
5. Jump operations
6. Vector operations
7. Data types and their size
8. Variable scope and storage class

4.1 Mathematical Operations

The basic operations in each algorithm are mostly mathematical (scalar) operations, e.g. additions, multiplications, etc. These binary operations consist of taking the values of two registers (operands), performing the mathematical operation, and writing the result into a third register. Unary operations take the same approach, but only with one input register. However, different algorithmic approaches exist to calculate these mathematical operations in hardware. The time it would take to perform an operation depends on its computational complexity, or the time it would take to run the algorithm. As the time needed for computation depends on the size of the input, we can study its behaviour as a function to infinite, i.e. limiting behaviour. We are able to classify each algorithm according to the growth rate of the function. This results in the 'Big O notation' or $O(X)$-notation, where X is a function of n with n the size of the input. The Big O notation is an asymptotic approximation of the real timing complexity of the algorithm. For example, an algorithm with $O(n)$ has a complexity that grows linear with the input size. The best performing algorithms have a complexity that is constant independent of input size, i.e. $O(1)$. However, most well performing algorithms will achieve a logarithmic complexity or $O(log(n))$. The feasibility of using algorithms with quadratic complexity $O(n^2)$ or higher, drops significantly when working with large input values/sets.

Depending on the available hardware (or simulated in software), the execution time of different mathematical operations can vary. Addition/subtraction operations are mostly $O(n)$ or $O(log(n))$, while the multiplication and divide operations are $O(n^2)$ for classical operations down to specialised algorithms with $O(n\ log(n))$ [133].

In a small experiment, we compared the performance of running arithmetic operations on two different platforms. The first platform was a 32-bit ARM Cortex-M3 MCU with a clock speed up to 48MHz on the EFM32 Giant Gecko board of Silicon Labs. The model used has 1MB of flash memory and 128kB of RAM [134]. The code on this embedded platform is deployed bare metal on the system with the use of the complementary system libraries in the Simplicity Studio Integrated Development Environment (IDE). The second platform was the Raspberry Pi 3B. This system consists of a custom Broadcom BCM2837 System-on-Chip (SoC) which includes a Quadcore ARM Cortex-A53 CPU with a clock of 1.2GHz and Level 1 (L1) (32kB) + L2 (512kB) caches. The chip is connected to 1GB of Low-Power Double Data Rate generation 2 (LPDDR2) RAM memory [135]. On this platform, a custom Raspbian OS was deployed with an updated kernel containing the Litmus real-time OS patch [136]. The Litmus$^{\text{RT}}$ patch is a testbench created for Linux to perform real-time multi-core scheduler analysis. It allows us to change the default task scheduler of Linux with different policies. For this experiment, we used a partitioned fixed-priority scheduling profile to run the process in isolation on a single-core with highest priority to minimise the interference of other processes running in the background of the OS.

A small testbench was created that generated 100 random pairs of numbers and performed one specific scalar operation on each pair. The elapsed time to perform the calculations was recorded for multiple runs of the testbench. The measurements were registered in the number of CPU clock cycles. By measuring the time in clock cycles, we ignore the difference in clock speed in order to compare the operator efficiency between the target platforms. Based on the recorded datasets, the average and worst-case execution times were acquired.

Table 4.1 shows the relative percentage difference of the average and worst execution time for four different scalar operations. We notice that the Cortex-A53 of the Raspberry Pi performs better for the addition, subtraction and multiplication operations compared to the Cortex-M3. However, the performance of the divide operation is worse for the Cortex-A53. This significant discrepancy in performance is easily explained as the Cortex-A53 has no specialised hardware divide on chip [137], while the Cortex-M3 has it [138]. As a result, the Cortex-A53 chip needs to simulate the divide operation in software which is slower to perform. Therefore, implementing an algorithm requires careful usage of operators as the performance of each operator between platforms is not always proportional.

Table 4.1: Relative percentage difference of the execution time for scalar operations between the Raspberry Pi 3B and Giant Gecko developer board. A positive percentage ($> 0\%$) indicates that the Raspberry Pi performs the testbench task in less clock cycles than the Giant Gecko board, and vice versa.

	Addition (+)	**Subtraction (−)**	**Multiplication (*)**	**Division (/)**
Average	25.62%	25.62%	23.64%	−23.95%
WCET	22.88%	24.78%	20.41%	−32.81%

The energy consumption for a single-threaded execution model was described by Tiwari et al. [74] as the sum of the base cost of each instruction, the interinstruction overhead from the circuit state of switching between instructions and the external effects on the hardware, such as cache misses, pipeline stalls, etc. The base cost for each instruction is dependent on the specific silicon that needs to be activated on the processing unit, e.g. ALU. Moreover, the energy cost is also dependent on the actual input data that is used as the power dissipation depends on the number of transistor switches that are happening [76], [77]. The order of instructions will also have an impact on the energy consumption and the shape of its distribution [139].

Based on the qualitative benchmark approach described in Section 2.3.1, we defined the following questions which an embedded software engineer can use to classify benchmark applications to compose an equivalent test set to evaluate the potential code behaviour of the target application.

- *Which types of mathematical operators are used in the code?*
- *Are certain operands more prominent in the code base? Unroll loops to get insight into the size order of operands.*

4.2 Recursion

A recursive algorithm in software is a procedure that is able to invoke itself at least once in the procedure for a finite number of times during execution until a termination condition is met. Recursive functions are a powerful tool in programming to solve problems with recurrence relations with a small finite computer program. However, the solution should be finite in order for the recursion to finish.

The main disadvantage of recursive algorithms is large and possibly unbound memory requirements to run. In order to operate and store information during a function call, a part of the memory gets allocated for that function call. This region of the memory that get allocated is called the stack. It stores local variables, arguments, and return information when the function ends. For each new invocation, a new portion in the stack gets allocated, which is called the stack frame. During a new invocation of a function, the stack gets updated with the return address after completing the call. Then, a new stack frame is created on which the state of the caller function is saved unto the stack, i.e. registers and local variables. When the execution of the function is done, the top of the stack is cleaned up and restored to the state before the call. After the stack frame is restored, and the instruction pointer is back on the return address, the program execution continues. Each method invocation generates an overhead in memory usage on the stack during creation of stack frames and processing time during creation and clean-up of those frames.

If a recursive method is not well bounded, the stack may run out-of-memory resulting in a stack overflow and termination of the program through exception. It is therefore of the upmost importance to check the maximum depth of recursion to verify that no stack overflow may happen during execution. Because of the possible overflow hazard, the official Motor Industry Software Reliability Association (MISRA) standard (Rule 16.2 – 2004) [140], for programming embedded controllers in safety-critical applications, states not to use recursion!

Due to the invocation overhead with each recursion, transforming it to an iterative loop will result in better or equal (e.g. tail recursion) resulting performance. However, the recursive solution will have smaller code size and be assumed more 'elegant' compared to iterations.

When a recursive call is the last thing to execute in the function, then the function is tail recursive. As a result, the compiler is able to optimise this recursion with tail call elimination. Since the recursive call is the last statement within the function, there is no need to save the current function's stack frame as there is nothing left to execute when the recursion ends. Therefore, the invocation overhead of saving/restoring the context and possible stack overflow are not applicable in this case.

The performance of recursion is mainly influenced by the invocation overhead, and thus the number of recursive invocations. The computational complexity of the divide-and-conquer recursion strategy is determined by [141]:

- The size of the problem ($n \geq 1$);
- The number of subproblems in the recursion ($a \geq 1$);
- The size of each subproblem ($\frac{n}{b}, b > 1$);
- The computational complexity of the code within the function except for the recursive calls themselves ($f(n)$).

Based on these terms, we are able to compose an approximation of the computational complexity for recursion based on the master theorem [141], given the recurrent relation:

$$T(n) = \begin{cases} c & n = 1 \text{ and } c > 0 \text{ is a constant} \\ aT(\frac{n}{b}) + f(n) & n > 1 \end{cases} \tag{4.1}$$

As $f(n)$ is the computational complexity of the function, we may apply:

$$f(n) = O(n^d) \quad \text{with } d \geq 1 \tag{4.2}$$

This provides us the formal definitions of the computational complexity for a given recursive function:

$$T(n) = \begin{cases} O(n^d) & a < b^d \\ O(n^d \; log(n)) & a = b^d \\ O(n^{log_b(a)}) & a > b^d \end{cases} \tag{4.3}$$

If we apply the master theorem on the example in Listing 4.1, we find that the recursion is called four times within the *for*-loop. This gives us $a = 4$. The size of each sub problem is stated in the recursive call ($\frac{n}{2}$), for which $b = 2$. The complexity of the function itself is determined by the first loop that is executed for n times. Therefore, the complexity of the function equals to: $f(n) = n$, and thus $d = 1$. The recursive notation of the recursive function in Listing 4.1 equals to:

$$T(n) = 4T(\frac{n}{2}) + n \tag{4.4}$$

Given the relation of a and b^d, we know which definition applies to the example:

$$a \ ? \ b^d \Rightarrow 4 \ ? \ 2^1 \Rightarrow 4 > 2 \Rightarrow a > b^d \tag{4.5}$$

The computational complexity of the recursive function in Listing 4.1 is equal to:

$$T(n) = O(n^{log_b a}) = O(n^{log_2 4}) \tag{4.6}$$

```c
void recursion(int n) {
    if(n <= 1)
        return;

    for(int i = 1; i < n; i++)
        calculate(n);

    for(int i = 0; i < 5; i++)
        recursion(n/2);
}
```

Listing 4.1: Example of a recursive function

The order of the computational complexity function gives an indicator on how fast the cost to perform the recursion will increase depending on the problem size n. A more accurate approximation, for which the sub problems of the recursion could have different sizes, is obtained by applying the Akra-Bazzi method [142]. This methodology generalises the master theorem with the recurrent relation of the form:

$$T(n) = \sum_{i=1}^{k} a_i T(b_i n) + f(n) \tag{4.7}$$

Based on the qualitative benchmark approach described in Section 2.3.1, we defined the following questions which an embedded software engineer can use to classify benchmark applications to compose an equivalent test set to evaluate the potential code behaviour of the target application.

- *Are there recursive calls inside the code base that are no tail recursions?*
- *What is the worst-case recursion depth during operation?*
- *How many recursive calls are within the recursive function itself?*

4.3 Loop Bounds and Nesting

Program loops are a programming paradigm that enables us to repeat a code section for an arbitrarily number of times. A loop consists of three main components: a conditional statement, iterated inner block with instructions and a jump operation. The last component manipulates the control flow of the program that will result in repeating the inner block section of the loop. However, the order of these components is not fixed. For example, a while-loop will first perform a conditional check to decide if the loop should be entered, whereas a do ... while-loop always enters the inner block section before evaluating the loop condition. The conditional statement of a loop is a special conditional branch that decides if the execution pointer should continue, or a jump operation is required to reiterate or exit the loop. This conditional statement of the loop is a Boolean condition (true/false) that is re-evaluated upon each iteration, and therefore determines the number of iterations.

In order to determine the WCRC of a loop, we need to know the number of iterations that loop will enter. The number of iterations is defined by a distribution as it is determined by the conditional statement of the loop that contains any Boolean expression, such as simple loops with linear arithmetic updates (e.g. iterating over a fixed size array), non-deterministic initialisation of the evaluated variables and multi-path loops with abrupt termination (e.g. use of the break statement) and/or monotonic conditional updates (i.e. updating the conditional evaluated variables within the inner block section) [143]. The minimum/maximum of this distribution are the bounds of the loop. For the WCRC, we are interested in the maximum loop bound that could occur during operation as each iteration will add up to the worst-case total.

When loops reside within the inner block section of other loops, then these loops are nested. As a result, the number of iterations of a nested loop gets multiplied by the loop bounds of each encapsulating loop, and thus grows exponentially with the number of iterations of its parent loop(s). For example, three nested loops, each iterating over a fixed array of 100 elements, will result in iterating the inner block section of the deepest loop by 1 000 000 times.

Automated loop bound extraction, that is sound and closely bound, is not a trivial problem to solve. SOTA WCET analysis tools use different techniques to determine these bounds, such as static flow analysis [144], and symbolic computation [143]. These techniques are able to generate safe upper bounds for different types of scenarios (e.g. loop interrupts, loop conditions, increment types, nested loops, etc. [144]). Nevertheless, some loops require more context by the developer. For example, when a conditional statement is dependent on input data that is not known at compile time for which the developer needs to provide hints on the value range of the input variable. This information is mostly provided through manual annotation of the software code. For instance, all code in the TACLeBench is annotated with flow facts using '*Pragma*'-annotations that indicate the minimum and maximum loop bound [60]. It allows the developer to provide insight of the entire system, e.g. system input, code dependencies, etc. This is useful when the loop condition is ambiguous or depended on system input. Additionally, it allows to strictly delineate the boundaries of the loop to avoid having too pessimistic estimations.

The computational complexity of a loop operation can be divided into five different categories. In the first category, the loop has a constant iteration bound, As the number of iterations is predetermined, the complexity of these loops is equal to $O(1)$. As the exit condition of the loop is predetermined, we are able to explicitly count the loop bound.

However, if the exit statement is variable, e.g. iterating over all elements of a variable-sized array, then the computational complexity is a function of n which indicates the size of the problem. The order of the function depends on the step size of the iterator:

- **Linear incrementation**: iterator is incremented/decremented by a fixed amount. $T(n) = O(n)$;
- **Multiplicated incrementation**: iterator is multiplied/divided by a fixed amount. $T(n) = O(log(n))$;
- **Exponential incrementation**: iterator is increased (e.g. power of x)/reduced (e.g. square root) exponentially by a fixed amount. $T(n) = O(log(log(n))$.

The last category are the nested loops. The computational complexity increases exponentially with each additional nested level. The instructions in the innermost loop will be executed the most, as the total number of iterations is equal to the multiplication of all loop bounds of its parent loops. As a result, the complexity of nested loops is equal to: $O(n^c)$, where c equals the number of times the innermost loop is executed.

In order to improve the performance of loops, the developer and/or compiler are able to perform several optimisations. These optimisations are oriented to increase the overall throughput by minimising delays in the hardware. As a result, certain optimisations will have a higher impact on the execution time depending on the available hardware capabilities and configuration. The effects of these optimisations are interdependent [51]. The performance may be better or worse when optimisations are applied independently or together. The compiler needs to search for the most optimal transformation for the loop while taking cache behaviour (Section 3.5), pipelines (Section 3.7), register allocation and loop overhead into account.

Changing the order and/or the structure of a loop operation, the compiler may improve the cache performance by utilising the 'locality of reference' [47], [51]. Examples of transformations focusing on the locality of reference are:

- **Loop fission**: splitting the loop in multiple smaller loops each iterating over a subset of the original index range;
- **Loop fusion**: combining multiple loops that iterate over the same index range;
- **Loop interchange**: switching the order of nested loops;
- **Loop tiling**: reordering the loop into blocks whose data size fit the cache in order to reduce cache fetches.

Loops have significant impact on the pipeline throughput, due to the integrated jump operation that is required to reiterate over the inner block section. More information on the impact of these operations is provided in Section 4.4 and 4.5. In order to minimise pipeline stalls and/or flushes, the following transformation techniques can be applied:

- **Loop unrolling**: duplicating the inner block section to reduce the number of iterations, and thus jump operations needed;

- **Loop inversion**: replacing while-loops with the do . . . while variant that is wrapped in an if-block in order to decrease the number of pipeline stalls caused by jump operations;
- **Software pipelining**: out-of-order execution of instructions between iterations without dependencies. The joining operations of the out-of-order execution is performed by the compiler (i.e. in software).

A last optimisation approach is to introduce parallelisation and vectorisation (Section 4.6) of the loop. This improves the processor throughput on respectively multiprocessor and Single Instruction Multiple Data (SIMD) vectorised systems.

Based on the qualitative benchmark approach described in Section 2.3.1, we defined the following questions which an embedded software engineer can use to classify benchmark applications to compose an equivalent test set to evaluate the potential code behaviour of the target application.

- *What is the size of the loop bound?*
- *Are there nested loops iterating over (multi-dimensional) arrays?*
- *What is the depth of the nested loops?*
- *What type of loop statements are used (e.g. while, do . . . while, for, etc.)?*
- *Are there consecutive loop sections iterating over the same data?*
- *Are the operations between iterations dependent or independent of each other?*
- *What is the size (i.e. number of operations) of the loop body?*

4.4 Conditional Statements

Conditional statements alter the program flow based on a condition. The result of the evaluated condition will update the instruction pointer to a specific ‘branch’ in the code. As a result, the instructions that are executed can be chosen at runtime. An example of a well-known conditional statement is the if . . . else construct. The conditional statement in the if-clause needs to return a Boolean response (i.e. true/false). Depending on the returned value, the inner block beneath the if-clause will be executed when the value equals ‘true’. If the value is ‘false’, the inner block below the else-clause will be executed instead.

Selecting a different branch is achieved by manipulating the instruction pointer to the first instruction of the desired inner block that needs to be executed. When a conditional statement is reached, there are two or more possible traces the control flow is able to follow. The instruction pipeline (Section 3.7) would need to stall operation each time a conditional statement is reached until the condition is evaluated to know which branch to continue on. Stalling the pipeline will result in significant performance penalty. In order to mitigate the stalling issue, pipelines have hardware built-in to guess which branch to take [132], i.e. continue execution or jump to the other clause. After taking the binary decision, the pipeline will continue execution. However, these instructions are now speculatively executed. This means that the output is not final yet. If the prediction was incorrect after the evaluation of the condition, then the speculatively executed instructions are reverted and dropped from the pipeline. The instructions of the intended branch will then be fetched and executed instead. The time penalty of a branch misprediction is equal to the number of stages between the fetch and the execution stage included, as shown in Figure 3.3. In modern architectures, the pipeline length has been increasing in size, resulting in higher losses when failing to predict the correct branch. Therefore, more advanced branch prediction mechanics are implemented in the hardware.

A wide variety of branch prediction strategies exist to have more advanced predictions at the cost of more complex hardware. The most significant prediction strategies are [145]:

- **Static branch prediction**: all predictions are statically determined at compile time. These decisions are based on rules, such as always taking backward branches, not taking forward branches, etc., or statically determined with the use of code annotation;
- **Dynamic branch prediction**: the decision for the prediction is influenced by the runtime history of the application itself. Depending on the strategy, a table of branches is stored that contains how many times each branch is taken. Based on this history, the logic will determine which branch to take when encountered;
- **Random branch prediction**: the branch taken is randomly chosen by a pseudorandom bit to guarantee a 50% success rate. However, the timing behaviour of the branch prediction becomes non-deterministic as we do not know which branch will be guessed at runtime.

The implemented strategy for branch prediction will have a significant influence on the best and average performance. The worst-case execution of branches triggers when the prediction is false and is similar to the performance without having branch prediction. The success rate of the branch prediction is key in getting accurate WCRC estimates.

Based on the qualitative benchmark approach described in Section 2.3.1, we defined the following questions which an embedded software engineer can use to classify benchmark applications to compose an equivalent test set to evaluate the potential code behaviour of the target application.

- *What is the order of magnitude of conditional selection statements in the code?*
- *What is the maximum depth of nested conditional statements?*
- *How many consecutive conditional branches (e.g. else if or cases) are there?*
- *Are the selection conditions statically predetermined?*
- *Does the selection statement contain logical and/or bitwise combined conditions?*
- *Is the selection statement a simple expression or complex combinatory evaluation/-function call?*
- *Does the conditional statement have dependencies on variables that get updated in close proximity right before the selection statement?*

4.5 Jump Operations

Jump operations are statements which manipulate the sequential program flow by changing the instruction pointer. The iteration (Section 4.3) and selection (Section 4.4) instructions are special cases of jump operators for which the actual jump is determined by the result of a binary condition, i.e. true/false. In this section, we will discuss the unconditional jump statements.

As the unconditional jump on itself is deterministic in execution, there are no hazards within the instruction pipeline compared with the iteration and selection statements, as explained in Section 3.7. Different type of jump instructions exists to adapt the execution flow. We group these instructions in three main categories: global, methods and block manipulation. On the global level, we have the go to (goto) statement. This instruction allows to jump to an arbitrarily line of code indicated by a label to continue execution. This code construct provides full control over the instruction pointer but is considered as bad practice and should not be used by the MISRA standard (Rule 14.4 – 2004) [140].

On the second level, the code is structured in methods that are invocated. By invocating methods, a new set of instructions is executed in its own context on the stack as explained in Section 4.2. When the method call has reached the end of its invocation, the stack is restored to its previous state and an optional value is returned on top of the stack to be used by the invoking method. This return action can be called explicitly within the code at any given point during execution with the return statement or gets implicitly called when the last instruction of the method, that is not a return statement, is executed. The action of invocating a method (e.g. method call), contains a jump operation as well after the stack has been updated.

The last level jump instructions are performed on block sections. These instructions will interrupt the execution and continue with the next iteration or block section. For example, the continue keyword will stop the inner block section of an iteration statement and jump to the conditional statement. A break keyword in an iteration will completely interrupt and exit the loop instead.

Jump statements have no direct impact on the pipeline, as there are no branches. However, changing the instruction pointer will impact the performance of caches as they rely on the locality principal [47], [50]. A detailed discussion on cache performance is given in Section 3.5. For method calls, we need to take the overhead of invocation (i.e. stack) into account.

Based on the qualitative benchmark approach described in Section 2.3.1, we defined the following questions which an embedded software engineer can use to classify benchmark applications to compose an equivalent test set to evaluate the potential code behaviour of the target application.

- *Does the code contain goto jump operators?*
- *What is the ratio between the invocated method's sizes and quantity (e.g. many small methods or few larger ones)?*
- *What is the number of method invocations?*
- *Does the code use break/continue statements within iterations?*
- *Are there multiple break/continue statements nested within selection statements of the inner body of the loop?*

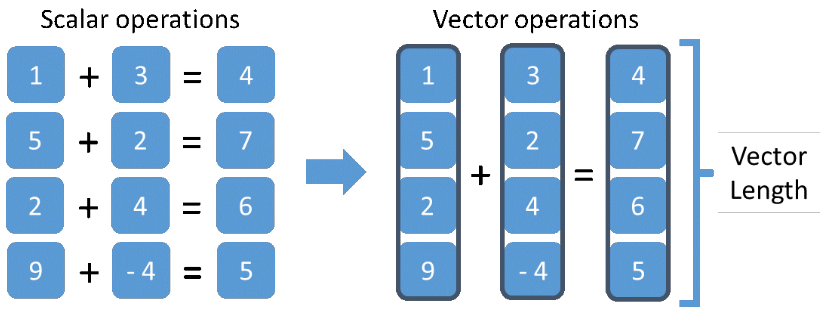


Figure 4.1: Example of SIMD parallelism.

4.6 Vector Operations

In simple computing architectures, all scalar instructions are performed sequentially one at a time. In order to improve the performance of the processor, new advances in the hardware have been achieved to increase the processor's speed. In 1965, the director of Intel, Gordon E. Moore, made the statement that every 18 months the overall processing power of processors will double [146]. For over 50 years, we have seen the number of transistors in the silicon rising linear with every new generation. The same characteristics were true for the clock frequency and the power consumption of the processor. However, since a decade, we notice a breaking point in the curve reaching an upper limit. This plateau is caused by the physical limits of cooling the CPU in the dense package, as higher clocking frequencies require more power which results in higher heat development that needs to be dissipated. Nevertheless, the performance of CPUs is still increasing with the introduction of parallelism, such as multi-core architectures and vector instruction support [147].

Operator vectorisation is the ability of the processor to group operands of scalar instructions in vectors and perform a single operation on those vectors. This approach is called SIMD parallelism which is illustrated in Figure 4.1. Data parallelism is common in modern high-end architectures. For example, intel implemented the first vectorisation in their processors with the MMX-instruction set to improve multimedia processing, followed by the Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) instruction sets [148]. For the ARM processors, the Cortex-A and R series implement the SIMD architecture extension with their NEON technology for multimedia processing acceleration [149].

In order to perform SIMD operations, the processor needs a Vector Unit. This module runs a single stream of instructions on the separate data elements of a small vector in parallel. In the case of 16 consecutive summation operations, we would need two load operations (one for each operand), an addition operation and finally, a store operation for each individual summation. This would result in $4 \times 16 = 64$ operations to perform. When vectorisation is applied, all the load, store and addition operations are done only once. These units are driven by specific instructions of the instruction set. Therefore, the software needs to be adapted to control the data parallelisation. The vectorisation of instructions can be performed manually in assembly code or automatically by the compiler.

Including vectorisation will introduce a certain overhead on itself to take into account. According to Mercelis [129], these costs are most commonly related to:

- **Vectorisation latency**: the latency introduced for performing vectorised instructions is higher compared to the scalar equivalents. This additional latency limits the actual upper bound of the theoretical speedup that can be achieved. By increasing the number of consecutive vectorised instructions, the throughput will improve as the vector units are used more efficiently and eventually, result in a lower impact of the vectorisation latency;
- **Packing and unpacking overhead**: transfer cost of data to and from the vector registers. Converting non-vector data, such as scalar arrays, to a vector array will introduce a significant overhead;
- **Branch unfolding**: as vectorisation cannot contain any branches, each path needs to be executed. At the end of the branch, the results for each vector element of the desired trace are selected. A code segment with many branches will introduce a considerable high cost;
- **Data alignment penalty**: the alignment of data in memory plays a major role in the access speed of the data during read/write operations. When data is not aligned with the cache lines (i.e. a data element is split over multiple cache lines), a time penalty will be introduced as read/write instructions will take more time to access the data. This alignment penalty is easily avoided by always aligning the data. However, a trade-off between speed and efficient usage of the cache memory will occur, as data padding (i.e. keeping memory cells empty) could occur to have all elements be aligned in memory. Most compilers will try to align the data by default to improve the overall performance;
- **Refactoring cost**: transforming scalar oriented code to a vectorised approach is not always possible, as not all architectures have an equivalent vectorised instruction available for each scalar one. Therefore, it would be required to rewrite the code base to support vectorisation. In the current intel architectures for example, all scalar operations in the instruction set have a vectorised counterpart.

In order to perform automatic vectorisation, the compiler needs to assure that no dependencies exist between the data elements of the vector. The vectorisation happens mostly on loops with scalar instructions that get split into chunks which fit in the Vector Unit of the processor. The compiler will determine if data parallelism is applicable to the scalar instructions, but certain limitations apply [150]:

- **No jump instructions**: no branches or jumps should be present in the loop body. However, conditional branches may be used to perform masked assignments;
- **Predetermined iterations count**: the number of iterations needs to be known before the loop is entered and should not depend on or change in the loop body. As a result, while-loops will not be vectorised as the loop bound is not known before entering the loop;

- **Applied on the inner loop**: the data parallelisation is only applied on the inner most loop when nested. However, if the compiler is able to unroll the loop or switch the loop order (i.e. inner with outer loops), then the new inner loop may be vectorised;
- **Only SIMD-enabled (or inlined) function calls**: pragma enabled functions that force the compiler to produce multiple implementations of the function, including a vectorised one. Using non-SIMD functions or subroutines in the loop will prevent any parallelisation attempts of the compiler;
- **No vector dependencies or unambiguity**: no dependencies between the vector elements may exist. For example, using the previous element of the array as an operand. Vectorising this dependency will result in false results (i.e. no error or exception will be thrown). This behaviour is due to the fact that all vector elements are calculating simultaneously, and thus the dependency on the previous element is not yet computed. Additionally, all references should be unambiguous for the compiler to guarantee data independence. The use of pointers could result in a parallelisation hazard, if the pointers are aliases of each other.

Based on the qualitative benchmark approach described in Section 2.3.1, we defined the following questions which an embedded software engineer can use to classify benchmark applications to compose an equivalent test set to evaluate the potential code behaviour of the target application.

- *Are there for-loops in the code (i.e. vectorisation does not apply on while loops)?*
- *Is the loop bound known before loop execution (i.e. the bound does not change or is dependent on the loop body)?*
- *Do the loops contain non-inlined function calls that are not SIMD-enabled?*
- *Do the loops contain jump statements (e.g. break, continue, return, switch, if-else-then, etc.)?*
- *Are the loops nested?*
- *Are there data dependencies between elements between iteration steps? If so, how many consecutive instructions can be batched in group without dependencies (when loop unrolling)?*
- *Do the loop body contain pointers? If so, is the usage unambiguous (no pointer aliasing)?*

4.7 Data Types and their Size

Computing units perform operations on data which are streams of bits (ones and zeros) that represent numbers. In programming, we create algorithms by performing operations on these numbers. In order to create a useful application, we need to define variables and what they represent with a data type. These data types tell the compiler what data is stored in the registers which the variable points to and how each operation should be applied on the variable. The most commonly used data types used in programming are:

- **Integer**: a discrete range of countable numbers which are equal to mathematical integers. Depending on the type, they are purely non-negative natural numbers (unsigned) or may contain negative integers (signed) as well;
- **Floating-point number**: represents an approximation of a mathematical real number where a trade-off is made between its range and precision. Floating-point numbers consist of significant digits and an exponent that is applied to a fixed base. The number of the significant digits determine the precision, while the order of the exponent digits determines the range of the number;
- **Character**: contains the smallest unit of information used in the system. Examples of characters are single digit numbers, letters, punctuations, whitespaces and control characters (e.g. tab and carriage return). However, the numeric value of a character can also be used as a byte sized variable for a whole number;
- **String**: a sequence of characters which length is determined by the number of elements (i.e. characters) it contains. Note: in native C programming, this type does not exist and is achieved by instantiating an array of characters;
- **Boolean**: a variable that has two logic states (i.e. true or false) that is used in Boolean algebra.

For the representation of the data, the compiler needs to know the size of the variable besides the data type, as each type can have different byte lengths, such as 16-bit Integer (e.g. short) or 64-bit Floating-point (e.g. long). Based on the data type and size, the compiler is now able to create symbols for each variable that are assigned to physical addresses in registers during the linking process [151].

It is the task of the compiler to translate the code into optimised instructions based on the available hardware and instruction sets of the computational unit, as discussed in Section 3.1. A basic component of the CPU is the ALU which is a digital circuit that is optimised to perform arithmetic and logic operations on integer numbers. This unit has two input ports for one integer operand each. The opcode of the instruction will indicate to the ALU which operation needs to be performed on the input. The result will then be provided on the output bus to write back in a register. Depending on the functionality of the CPU, other math coprocessors may be included. The Floating-Point Unit (FPU) is a computing unit that performs operations on float point operands, whereas a GPU is very efficient for vector operations, such as image processing and computer graphics. These computational units are each optimised to handle specific data types with a maximum size. In the event that an instruction cannot be directly mapped onto the available hardware, the compiler needs to simulate the operation in software by splitting it in multiple steps that the hardware is able to handle. For example, using floating-point numbers on hardware without FPU requires to convert floating-point numbers into integers to perform operations on the ALU. This conversion results in a high penalty in precision and execution time [152]. The same conversion overhead occurs when larger data types are used than the size of the MCU registers, such as defining 32-bit integer variables on an 8-bit controller.

The dynamic power consumption of the processor is also determined by the number of transistors that are toggling in the silicon [70], [72], as discussed in Section 2.1.2. The data types used will therefore have an impact on the amount of active silicon, such as the usage of specialised hardware (e.g. ALU, FPU, etc.) on the processor, and thus impact the WCEC of the task.

Based on the qualitative benchmark approach described in Section 2.3.1, we defined the following questions which an embedded software engineer can use to classify benchmark applications to compose an equivalent test set to evaluate the potential code behaviour of the target application.

- *Which data types are used?*
- *Are there hard(ware) and/or soft(ware) floating-point operations?*
- *What is the bit size of the data types used?*
- *Is there type casting of variables present in the code?*

4.8 Variable Scope and Storage Class

The location of a variable definition in code is an important factor in the physical location where it will be stored in memory. This is implied by the default storage class of a variable. By default, a variable that is defined inside a function is called a local variable and only exists inside the block (and child blocks) in which it was defined. These variables are created on the data stack. Therefore, the address of these variables is relative to the stack pointer.

When a variable is defined in the global scope or defined with the extern keyword (only within the same source file), the variable will be accessible throughout the program. The memory address, and thus its value, is shared among all methods. These variables are stored in the data segment of the application. The data segment is an allocated portion of the application RAM where predetermined (i.e. at-compile time) fixed-sized variables are stored. As a result, the addresses of those variables are absolute as they were assigned during the linking of the compilation process.

The storage class of a variable is overwritable by adding fixed keywords to the definition of the variable. For example, the static keyword will keep the variable in the data segment of the program, so that the value is retained between function calls. Variables that are immutable during the entire application lifetime may be declared as constant. A constant variable is embedded in the application code and resides within the code segment where all instructions are stored.

In the C programming language, the programmer is responsible to manage the memory during the application lifecycle. This feature requires careful implementation by the developer but provides a powerful tool to optimise performance and resource requirements. An example of such optimisation is the usage of the register storage class. This class hints the compiler that the variable in question will be frequently accessed and suggests storing it in the CPU's register instead of the RAM to improve access time and conserve energy. Nevertheless, this requires that the variable must fit inside a register, and it will not have a memory address, as it is stored solely in registers. Furthermore, there is no guarantee that the variable can and/or will be allocated to a register.

Until now, we have discussed variables of which their memory location is defined at compile-time and are directly addressed. However, the concept of pointers allows us to address a memory location in order to indirectly access a variable. A pointer is a numeric value that contains a physical location in memory. To get the value of the variable, we need to access the memory address the pointer is pointing to. The integration of pointers provides a powerful mechanic to pass the reference to the variables around instead of copying large data structures. Additionally, this opens the functionality to address dynamic allocated memory.

The memory allocation allows the developer to reserve a chunk of memory for variables at runtime. The advantage of using dynamic memory allocation is that the size of the allocated memory is not defined at compile time. All allocated instances are placed on the heap which is a reserved part of the system's memory that is not managed by the system itself. The developer is responsible to clear allocated memory when it is not needed anymore. If no memory gets released, the heap will grow resulting in a memory leak until no space is available. Notwithstanding, the MISRA standard discourages every use of dynamic memory allocation, as it could result in out-of-storage runtime exceptions and potential undefined behaviour (Rule 20.4 – 2004) [140].

As discussed in Section 4.3 and 4.6, the location of data and instructions have a significant impact on the performance due to long fetch times from higher-level memory layers. Allocating dynamic memory comes with a high-performance cost to manage the heap.

Based on the qualitative benchmark approach described in Section 2.3.1, we defined the following questions which an embedded software engineer can use to classify benchmark applications to compose an equivalent test set to evaluate the potential code behaviour of the target application.

- *What storage classes are used (global, local, static, register)?*
- *Are there dynamic allocations on the heap?*
- *Are data structs stored in fixed size direct access arrays or dynamic lists (i.e. pointers)?*
- *Are data structs/arrays copied or passed by reference?*

4.9 Conclusion

In this chapter, we provided a list of prominent software logic that impact the WCRC behaviour of software tasks. Additionally, we composed a list of questions for each section that would aid engineers in compiling a relevant set of benchmark tests to compare the platform's performance and behaviour according to the proposed methodology in Section 2.3.1. Finally, the influences of these code constructs are presented with their computational complexity and interactions with the target platform. These software influences provide valuable insights in the creation of optimal feature sets to train high performing ML models as discussed in Chapter 6.

Chapter 5

Hybrid Worst-Case Resource Consumption Analysis

The hybrid WCRC analysis combines the static and measurement-based methodology in a two-layered approach. With this approach, we want to find a balance between computational complexity and the precision of both methodologies. In this chapter, we discuss the implementation of the technique, and how we have integrated it in the COBRA framework for the automation of the analysis process.

5.1 Implementing the Hybrid Methodology

In the domain of WCRC analysis, the two major techniques on the opposite sides of the spectrum are the static analysis, i.e. a mathematical approach to model the behaviour; and the measurement-based approach, i.e. physical probing the Device Under Test (DUT) to gain insights. Both approaches are used by researchers and adopted by industry to tackle the upper bound analysis problem. Nevertheless, both techniques have their drawbacks. On the one hand, the static code analysis requires abstract hardware models, that are difficult to obtain, and complex interaction models of the control flow on the hardware which makes this approach intractable for complex systems. On the other hand, the measurement-based analysis does not require hardware models or complex analysis of the control flow. However, physical measurements need access to the final product that are prone to the probing effect [153] and require large amount of time and effort to test as many system states as possible which does not guarantee that the real WCRC is captured. In order to overcome these shortcomings, we believe a hybrid approach to calculate the WCRC of a task will provide the solution. The hybrid approach will combine a time measurement-based approach at machine level with a static analysis approach at algorithm level, as shown in the schematic overview of Figure 5.1.

The hybrid methodology starts with splitting the code of a set of tasks into smaller components, or so-called 'hybrid blocks'. A hybrid block resembles a trace of instructions that only has one entry point and on exit point in its control flow. These blocks have a similar functionality to the 'basic blocks' as described in the literature [83], however the size of a hybrid block may vary from a single instruction up to entire code sections or functions. By dividing the code into variable sized hybrid blocks, we are able to create

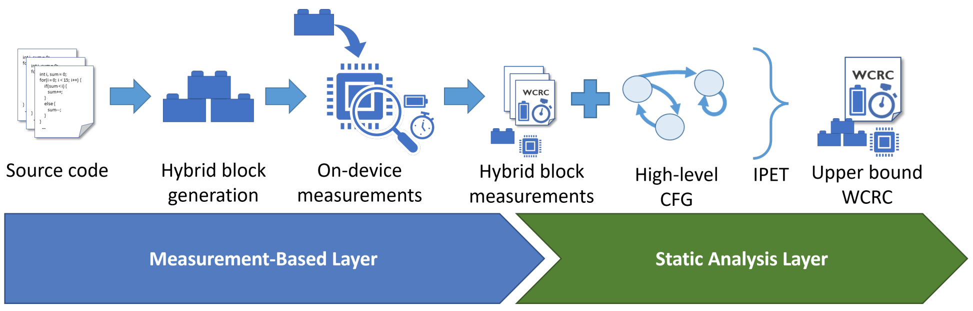


Figure 5.1: Schematic overview of the hybrid WCRC methodology.

a balance between the computational complexity of the static analysis layer and the precision of the measurement-based layer. The process of generating these hybrid blocks is discussed in more details in Appendix A.2.

Based on the resource consumption of interest, we have created two measurement platforms to perform on-board measurement for execution time (Section 5.2) and energy (Section 5.3). Those results are then combined within the static analysis layer based on the high-level CFG of the hybrid blocks as discussed in Section 5.4.

The challenge of this two-layer hybrid approach is tackling the computational complexity problems within the static analysis layer and the accuracy within the measurement-based layer. In other words, a proper balance needs to be found between the two layers of the hybrid model. This goal introduces four main challenges:

1. Finding the right block size in order to keep the complexity of the static analysis contained and to keep the accuracy of the conducted measurements, i.e. more blocks mean more complex static analysis and fewer blocks result in less accuracy in the measurement-based approach;

2. How to reduce the required number of measurements while keeping a high accuracy based on the information from the static analysis layer. This can be achieved by reducing the measurement space using the static analysis information, e.g. excluding non-existing concurrencies;

3. Combining the information of the static and measurement-based layer to reduce the predicted upper bound by excluding irrelevant results. This will be accomplished by adding additional information to the measurements in order to use only the relevant subset of measurements. Only those results fitting the situation in the state of the static analysis will be used, e.g. not using correlation with start-up procedures in a shutdown phase;

4. Exploring the influence of the number of measurements conducted and their errors on the static analysis and the total WCRC analysis. As a result, we will make the right trade-off between the measurement cost and the required precision depending on the specific application domain.

Research on hybrid WCET methodologies has resulted in different analysis tools, such as TimeWeaver [100] and RapiTime [101]. These tools perform low level analysis on compiled code. As a result, they will only support a specific group of predefined target hardware and compilers which are examined by the tool developers. Our goal is to create an analysis tool which performs and automates the analysis process based on a higher level, i.e. the code base. This tool we created is called the COBRA-Hybrid Program Analyser (HPA) framework, which is presented in Appendix A. With this approach, we want to gain insight and find correlations between source code and the resulting WCRC, so that embedded programmers can have continuous feedback on code changes during development and interpret the results on code level instead of binary code.

5.2 Measuring Execution Time

Depending on the available features on the hardware, a different measurement approach will be taken. For example, dedicated debug pins on the hardware platform provide valuable inside into the processor inner workings which are mostly found on development boards. These pins allow us to perform accurate measurements. Nevertheless, if the debugging facilities are not available, other techniques will be applied to acquire an approximation.

We try to select the most optimal strategy with the Profiling Assistant to perform the measurements on the target hardware. Initially, we presume the best-case where we know the full model of the processor architecture and full debugging functionalities are available. If this is not the case, we will continue to fall back to less feasible approaches resulting in a more overestimated upper bound. For example on the ATMEL AT90CAN128 development board [154], we have full debugging access and sufficient input pins to provide input data. For this platform, we perform the profiling using an external device to keep the influence of the measurements to a minimum in order to reduce the probe effect on the hardware [153].

The external measurement device we use is a Field-Programmable Gate Array (FPGA) with custom firmware that allows us on the one hand, to count the exact clock cycles to execute each block on the board with low interference. The FPGA uses the CPU clock of the MCU for counting the elapsed clock cycles. On the other hand, it delivers the input for each measurement as such that the flash memory of the board is relieved from large datasets. As a result, we are not limited by the flash memory size when generating our input sets.

However, it is not always possible to use the previously mentioned procedure. In those cases, a lower tier profiling approach is applied. As an example, the OCTA-Mini developed by IDLab is equipped with an ARM Cortex-M3 [138]. The board itself has no full debugging capabilities or as many I/O-pins compared to the previously mentioned ATMEL development board. The timing measurements are still performed by the external FPGA device. Nevertheless, the input sets are now embedded onto the flash memory. As a result, multiple smaller test runs are required to perform the full analysis. If using an external profiling device cannot be used to probe the target hardware, we will fall back onto on-device alternatives, such as Tic-Toc cycle counter that uses an internal register to count the elapsed cycles from the start until the end of the testbench. Nevertheless, these methodologies are more intrusive on the system's behaviour.

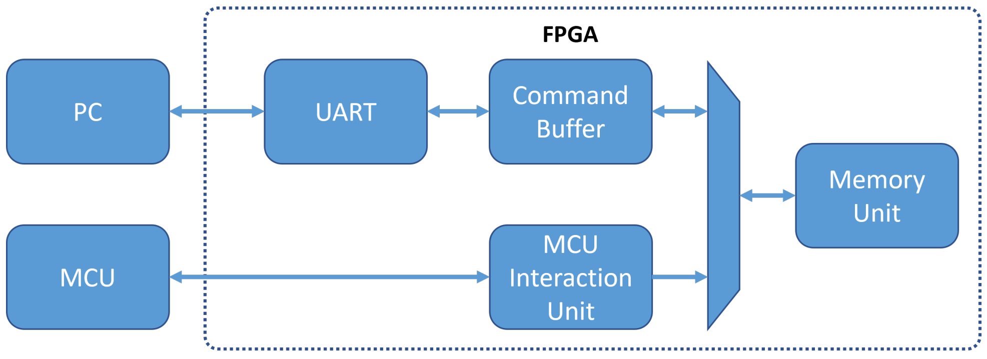


Figure 5.2: Block schematic of the FPGA measurement platform.

5.2.1 FPGA Measurement Platform

In order to perform execution time measurements, we have developed an external measurement unit that is able to profile a device with minimal influence. For this measurement platform, we used a Digilent Nexys4 DDR FPGA development board [155]. This board has a Xilinx Artix A7-100 FPGA unit on board. A block schematic of the design is illustrated in Figure 5.2. The board interacts with the Profiling Assistant on the Personal Computer (PC) through the Universal Asynchronous Receiver-Transmitter (UART) over Universal Serial Bus (USB) connection. The communication with the MCU unit is provided through the Pmod ports on the Nexys4 board.

The data from the UART module is sent to the Command Buffer. This module selects which command the FPGA will perform and when. The design currently supports four command types: loopback, read, write and start. The 'loopback' command is used to test the UART connection. This test consists of sending one byte of data and writing this byte back on the bus. The 'read' command allows us to read the memory of the FPGA. A two-byte argument is used to select which memory address should be send back to the PC. The 'write' command does the opposite operation and write data back into memory. The first two bytes selects the target address, while the last two bytes contain the data that needs to be written to memory. The last commando 'start' prepares the FPGA to perform measurements. First, the FPGA will send a start signal to the DUT. Then, the MCU will execute the code that needs to be profiled and indicate to the FPGA when the code execution has started and ended. During the code execution, the FPGA will count the number of clock cycles that have been passed on the DUT. The result is then stored in memory on the FPGA.

In order to get an accurate WCET reading, we need to synchronise the clocks of the FPGA device and the DUT. It is important that the clock on the FPGA does not drift or jitter. Therefore, we will measure the number of clock cycles on the DUT. As a result, we need to sample this clock signal with a sampling frequency that is high enough to avoid aliasing. According to the Nyquist-Shannon theorem, we need to have a sampling frequency that is at least two times the frequency of the signal that we want to sample. The Nexys4 board is able to run on a clock frequency of 100 MHz and higher. Therefore, we are able to sample clock signals of devices with an internal clock frequency of up to 50 MHz.

The connection between the FPGA and MCU consists of three signals: start, tic-toc and clock signal. The start signal is sent by the FPGA to the MCU to indicate that the system is ready for profiling. The FPGA waits until the Tic-Toc signal line is pulled up high. A rising edge of the signal indicates that the code testbench is currently being executed. The FPGA will start measuring the number of cycles on the clock signal until a falling edge is detected on the Tic-Toc line. Each result will be written to the internal memory of the FPGA. The Profiling Assistant will keep track of each completed test to read the results back from the memory.

Performing these measurements on the DUT will introduce some overhead to the results, as the MCU needs to sense and toggle some General Purpose Input/Output (GPIO) pins to start the experiment and notify the FPGA when an experiment is running on the device. The Profiling Assistant will therefore start with a calibration step to measure this overhead by performing an empty Tic-Toc sample test. We have observed a consistent offset of 53 cycles on the FPGA. When we perform this experiment on the DUT itself with internal registers to count the cycles, we only noted an error of 9 cycles. This large deviation is due to the overhead of manipulating the external GPIO pins on the board. Nevertheless, this error on the Tic-Toc timer has no impact on the measurement as this is a consistent error, we are able to compensate in the results.

5.3 Measuring Energy Consumption

As discussed in Section 2.1.2, the energy consumption does not only depend on the software instructions and the hardware state, but is also influenced by, for example, the environment temperature. Any measurement of the energy consumption is prone to additional noise making it difficult to find an absolute number for the consumption of a given state. In order to make the measurement process more feasible, we created an automated testbench that is able to profile different types of embedded devices and integrates with the COBRA framework. The key requirements we used for the creation of this testbench are:

- Acquire accurate power measurements of the target device during profiling;
- Provide a synchronisation mechanism to profile the code of interest with a high sampling frequency;
- Include easy integration possibilities with Application Programming Interface (API) interface for automated access and control by the computer;
- Configurable power supply for the target hardware on which the voltage and current are measurable;
- Universal design to be compatible with different types of platforms;
- Capable of programming/flashing the target device without manual intervention or influencing the profiling process;
- Full automation of the system with the COBRA framework through scripting.

Based on the requirements above, we created a diagram of the testbench setup as illustrated in Figure 5.3. The next step is the selection and composition of the testbench components. In the remainder of this section, we discuss the different components of the energy measurement testbench.

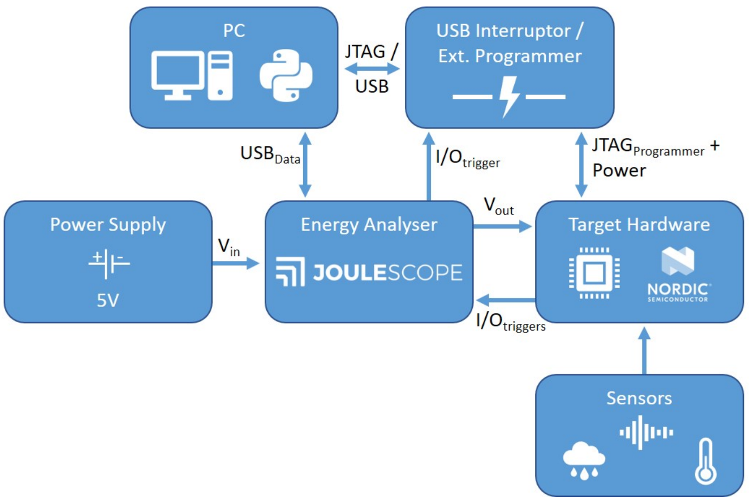


Figure 5.3: Schematic overview of the automated testbench for energy measurements.

Table 5.1: Comparison of measurement devices to use as an energy analyser.

Device	Range	Sample Rate	Accuracy	Price
Keysight Technologies N6705C DC Power Analyser [156]	Up to 600 W Max. 150 V Max. 50 A	50 kHz	0.025% + 8 nA	>€7000
Nordic Power Profiler Kit (nRF51/52 only) [157]	1.7 V ∼ 3.6 V 1 µA ∼ 70 mA	77 kHz	0.2 µA	€77
Silicon Labs WSTK Main board [158]	1 µA ∼ 95 mA	10 kHz	1 µA ($I < 250$ µA) 0.1 mA ($I > 250$ µA)	€76
Jetperch Joulescope [159]	−1 V ∼ 15 V −1 A ∼ 3 A	2 MHz	1.5 nA	€885

5.3.1 Energy Analyser

The first step in the composition of the testbench was the selection of the energy analyser. This component needs to accurately read the power consumption of the target hardware. Therefore, we made a comparison of several devices at our disposal that could perform the task according to the key requirements, as shown in Table 5.1.

The N6705C DC Power Analyser from Keysight [156] is a power analyser with built-in power supply. It has extreme wide ranges with a maximum power of 600 W in the default configuration. The analyser has an interface to save recordings to the operators' computer and GPIO pins that can be used as triggers. However, the sampling rate of the N6705C is rather low compared to the other devices and the price for one unit is extremely high which makes it expensive to run multiple analysers in parallel.

The Nordic Power profiler kit [157] is an extension board that works together with the nRF52 development kit. It supports external triggers for synchronisation and is a very low-cost solution. Nevertheless, the profiler kit is designed to be used only with the nRF51 and nRF52 development boards of Nordic and is no good choice to create a universal testbench.

The main board of the Wireless Starter Kit of Silicon Labs [158] has an on-board Advanced Energy Monitor (AEM) circuitry that enables to perform low-cost energy measurements, such as the Nordic profiler kit. The board can be used for different target boards by using the corresponding peripheral pins. The dynamic ranges and sampling rate are very low as the main board is designed for low-power intermittent computing devices.

From the devices in the table, we have selected the Jetperch Joulescope Energy Analyser [159] as the perfect candidate. This analyser has a decent measurement range that supports a wide range of low-power embedded IoT devices. The sampling rate is also significantly higher compared to the other power analysers and has decent accuracy specifications. The price of one unit is high compared to the low-cost kits but is not as expensive as the Keysight Power Analyser. In addition, the manufacturer provides detailed documentation and API support to operate the Joulescope with Python, which makes it accessible for full integration with the COBRA framework. Furthermore, the Joulescope device has four digital I/O pins that are operable through the API. The input pins are sampled at the same rate as the output power connectors. The states of the pins are piggybacked on the sampled data which allows us to register the exact sample the trigger was received at. The Joulescope requires an external power supply to power the target hardware in contrast to the Keysight Power Analyser. This is not a significant problem, as we can use an appropriate supply that fits the setup or even test the behaviour when using an energy harvester by placing the Joulescope in between.

Another solution, that was not included in Table 5.1, is the usage of a high-end oscilloscope with two probes to measure the voltage and current of the target device, such as the LabNation SmartScope [160] with a USB interface. While oscilloscopes have a high sampling rate with a high accuracy, it still needs a high-precision shunt resistor in parallel with the tested load to read the voltage drop across the shunt resistor to calculate the current flowing through the device. Creating a high-precision shunt resistor setup is not a trivial task to perform. Additionally, a shunt resistor is calibrated to work optimally within certain boundaries which makes this approach less desirable for high fluctuating currents and interchangeable target hardware running at different operational current ranges. Another approach would be to buy a high-precision low-current probe; however the price easily exceeds €3000, without the cost of the oscilloscope itself.

5.3.2 USB Interruptor/External Programmer

One of the requirements of the automated testbench is the ability to program the target hardware during the profiling process without manual intervention or influencing the measurements. Most developer boards have a programmer unit on the board that is able to flash a binary from the computer with USB, such as the Nordic nRF52 development kit. When the development board is connected through USB to the computer, the board and J-Link programmer are powered from the USB-bus. An easy solution would be to use a USB-front plate on the Joulescope to measure the energy consumption over the

USB-wire. However, the J-Link programmer, indicator Light-Emitting Diode (LED), processor debug mode and the internal voltage divider have a significant impact on the measured energy consumption which are not present in normal operation. Therefore, we need to interrupt the USB power after the flash procedure and switch over to the external power supply to have realistic measurements.

Turning off the USB power from the computer's USB-bus programmatically by the OS is, in most cases, not supported by the motherboard, as these settings are controlled by the Basic Input/Output System (BIOS) at boot time. To tackle this issue, we use an external breakout board that physically interrupts the power line of the USB connection. The USB interruptor uses a switching circuit that opens the 5 V line when its trigger pin is pulled to ground. The interruptor is controlled by one of the digital I/O pins of the Joulescope. The software driver we have developed for this setup will control which power supply is used during the profiling process. As the Joulescope has the capability to disconnect the power supply from the V_{out} to the board, we guarantee that both the USB power and power supply are never connected to the target hardware at the same time to prevent any damage to the hardware.

If the target hardware has no programmer on the board, we are able to switch the USB interruptor with an external programmer, such as J-Link. The Joulescope will then keep the target hardware powered while needed. A combined approach with the USB interruptor is possible when the external programmer power to the board cannot be disabled programmatically.

5.3.3 Measurement Synchronisation and Acquisition

The testbench needs to measure the energy consumption of a hybrid block on the target hardware. To gain insight in the behaviour, each block is executed multiple times with different inputs to find an upper bound of the WCEC. In order to achieve this, the testbench needs to know when a hybrid block is executed on the hardware. Therefore, we use both the digital input pins of the Joulescope as triggers to synchronise the execution and indicate the progress. The first trigger indicates the start and end of one run of the hybrid block's code. The second trigger indicates that all test cases are executed and the testbench code has terminated after which the COBRA Profiling Assistant can continue to the next profiling task. The trigger mechanism is achieved by connecting two GPIO terminals of the target hardware to the input pins of the Joulescope. The annotation of these triggers is accommodated by the COBRA framework in the testbench template. The developer needs to provide an implementation to toggle the GPIO pins of the board in the hardware configuration file.

When a new run of the block is initiated, the trigger signal will go high as illustrated in Figure 5.4. This signal stays high for at least two times the sampling period of the Joulescope, indicated by the vertical lines on the graph. This assures that each trigger signal is captured by the analyser. The actual measurement starts when a falling edge of the first trigger is detected and stops on the rising edge of the next observed trigger signal. The measured samples between these two points are used in the discrete integral to calculate the energy consumption of the executed task. This technique allows us to create tight measurement windows capturing the consumption during the execution of the code of interest. Notwithstanding, the Joulescope's high sampling frequency of 2 MHz is still relatively low compared to the higher clock frequency of the Nordic nRF52 processor running at 64 MHz. As a result, a sampling error is introduced as highlighted

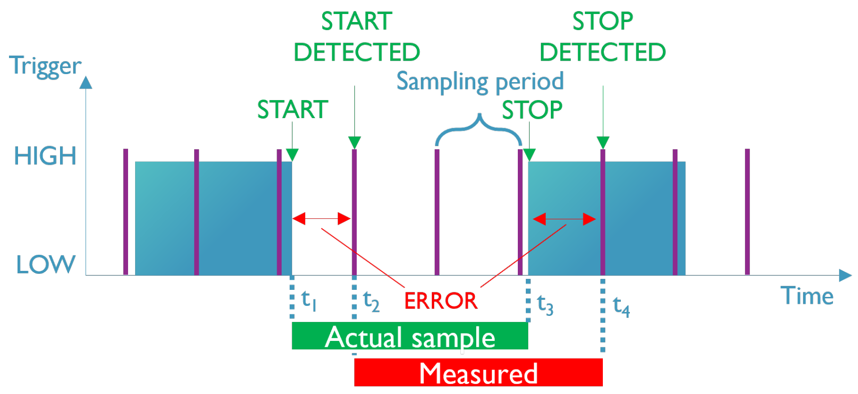


Figure 5.4: Sampling of the trigger signal for synchronisation during profiling. Illustration of the measurement error introduced by the lower sampling frequency of the analyser.

in Figure 5.4. This error is caused by excluding and including energy buckets that respectively are and are not part of the code execution. For example, the falling edge of the code execution starts at t_1 after the last sample was taken. At the next sample point at t_2, the energy consumption of the time interval $[t_1; t_2]$ has not been included in the measurement. The opposite situation occurs at the ending side of the measurement. The rising edge of the stop trigger takes place at t_3 before it gets sampled at t_4. The interval $[t_3; t_4]$ will therefore be included in the energy consumption of the hybrid block while it is not a part of it. In the worst-case scenario, the Nordic board with a clock frequency 32 times higher than the sampling frequency of the Joulescope will have a mismatched window of around 1 µs or 64 clock cycles on the target device.

In order to minimise the impact of the error on the measurements, we apply two strategies. Firstly, the upper bound of the WCEC should always be higher than the actual worst-case to avoid underestimation. Therefore, we propose to include the measured sample before the detected trigger sample. This will introduce additional pessimism to the upper bound, but this overestimation will be less detrimental for the WCEC.

Secondly, the length of the sampling error is independent of the task length. The longer a task runs on the hardware, the smaller the impact of the error will become on the result. For that reason, we have implemented a feature in the COBRA framework that helps expanding the code of a hybrid block when the execution time is too short for sound analysis with the testbench. The Profiling Generator will repeat the code in the block multiple times within one measurement to increase the sample's execution time. There are two strategies available: loop-based iterations or unrolled iterations. The looped-based iterations will wrap the code inside a loop statement. The unrolled iteration approach will duplicate the code section multiple times equal to the iteration count. This last strategy is not just copying the code X number of times as it will not result in valid code by default, due to possible duplicate declaration exceptions. For the experiments, we used the loop-based iterations for stable performance as the unrolled iteration strategy is still in development. Nevertheless, the COBRA-HPA toolchain takes the usage of looped hybrid blocks into account during analysis. The Feature Generator, for example, will multiply all block features by the number of iterations of the loop and

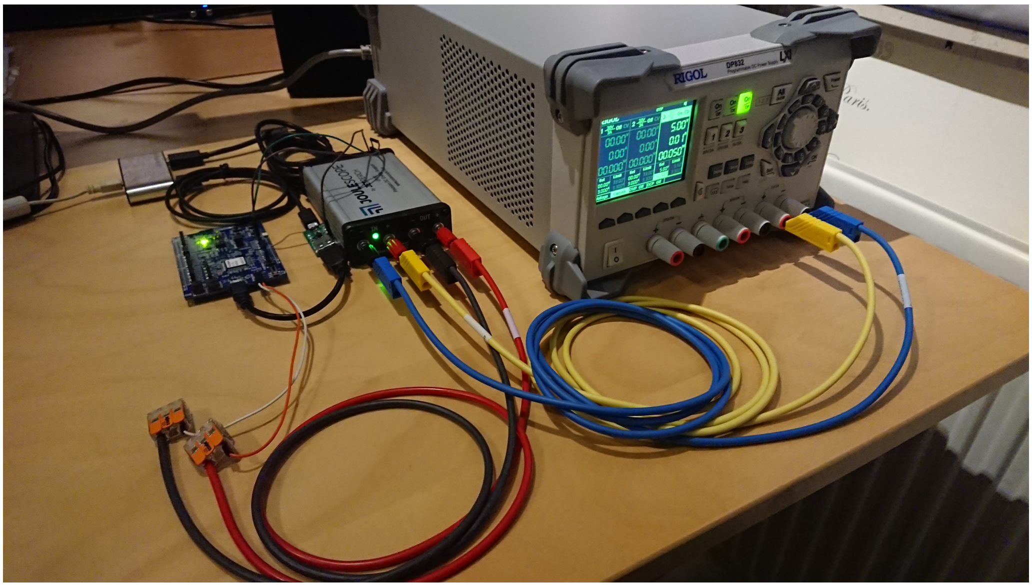


Figure 5.5: Picture of the physical setup of the automated testbench with the Nordic nRF52 development kit.

add the overhead of the iteration statement itself to the total.

The acquisition and processing of the sample data is performed by the host computer of the Joulescope. Each sample point, that contains the voltage and current in 13-bit precision and I/O input states [159], is transmitted via USB to the computer at 2 MHz, or 2 million samples per second. At full speed, we will need around 1.5 GB of RAM to store 60 s of samples. If the computer is unable to process the sample data in time from the Joulescope, then those samples will be dropped by the analyser and lost forever. To handle all these samples in time without requiring a powerful computing system with lots of RAM, we have implemented a moving window on a circular buffer in our testbench drivers. The circular buffer provides a more efficient and stable use of the RAM memory. New samples are added at the end of the buffer while a moving window is accumulating the samples to calculate the energy consumption. The measurement results are written to a log file and reported back to the Profiling Assistant to create the labelled WCEC upper bound datapoints.

5.3.4 Power Supply

The power supply used in our setup is the RIGOL DP832 programmable DC supply. It has three precision output channels of 3 A each and voltages up to 30 V for a total of 195 W [161]. The Nordic nRF52 operates on a voltage level between 1.7 V and 3.6 V within the limits of the DP832 supply. The result of the final setup with the Nordic nRF52 development board is shown in Figure 5.5.

5.4 Static Blocks Combination

The final WCRC value is acquired by statically combining all the measurements of each hybrid block. In order to find this worst-case upper bound, we need to find that path in the control flow that would maximise this cost. There exist different techniques to compute the upper bound. Three major techniques that are commonly used to compute upper bounds are *path-based* [162], *tree/structure-based* [53] and *IPET* [66].

The first computation method is explicit enumerated or path-based calculation. A CFG of the program is used to search for the longest path as it will reflect the WCRC. A significant bottleneck of this methodology is the computational complexity which rises exponentially with the number of control flow branches, as each possible path has to be considered [83], [162]. Therefore, this approach quickly becomes infeasible to perform as the number of possible paths quickly grows, as many paths in the control flow are not feasible to be executed due to code dependencies [163].

The second method involves tree-based calculations. An estimate of the WCRC is created by reverse traversing an Abstract Syntax Tree (AST) of the program. The leaf nodes of the tree contain basic blocks with larger code bases, which are connected by intermediate nodes that are controlling the program execution flow, e.g. iterations, jumps, etc. The resource consumption is estimated by combining the measurement results at the bottom according to the rule of the common parent node when traversing to the top. This method is computationally feasible to perform, however it is not able to consider dependencies between statements [53], [83], [164].

The final method is the IPET technique. Finding the worst-case path is approached by transforming a directed graph, i.e. CFG, with basic blocks into a system of linear constraints. Each block and edge have a traversal counter and a cost, e.g. WCET or WCEC. The given constraints can be solved using ILP [66], [162], [165]. The next step is a matter of solving a maximisation problem. As a result, the sum of the traversal counter and the cost needs to be as high as possible in order to find the worst-case. The outcome will be the worst-case count for each block and edge. A big advantage of this approach is the ability to incorporate additional constraints into the equations to model flow facts, and thus improving the upper bound precision [66], [83], [162], [165].

The CFG as illustrated by Figure 5.6 from the code in Listing 5.1 is a directed graph $G = (N, E)$, with a set of nodes $N = n_i : 0 \leq i < |N|$ and edges $E = e_i : 0 \leq i < |E|$. Each block is represented by a node in the graph. The code inside such node only has one entry and one exit point. The edges indicate the different paths or control flows within the program. When we want to determine the WCRC from the CFG, we need to solve an objective function by maximising its cost. This function $WCRC$ consists of two components [73], the execution frequency f of a node n_i and the actual cost for each node $C(n_i)$ and edge $C(e_i)$, i.e. the measured WCRC. The objective function is described in Equation 5.1. Lastly, we need to apply flow constraints, such as loop bounds, to the CFG in order to avoid infinite flows that the ILP solver cannot resolve [73].

$$WCRC = max((\sum_{n \in N} C(n) \times f(n)) + (\sum_{e \in E} C(e) \times f(e))) \tag{5.1}$$

In our framework, we acquire the final WCRC prediction using an IPET on the CFG that is generated from the block model using graph transformations. Each block type in the COBRA-HPA framework has a corresponding transformation that describes how it connects with its parent and child nodes in the hierarchical block tree or AST. Using

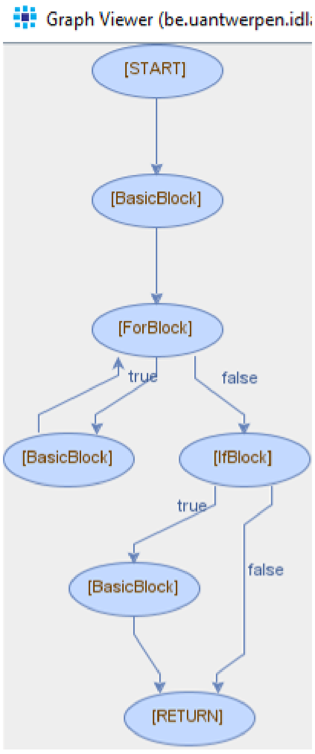


Figure 5.6: CFG of the code in Listing 5.1 generated with the COBRA-HPA framework.

the GNU Linear Programming Kit (GLPK) [166], we are able to find the worst-case path that results in the highest resource consumption with ILP given the CFG, extracted flow facts and the upper bound predictions for each node in the graph.

```
/** The procedure bitonicMerge recursively sorts a bitonic sequence in
   ascending order, if dir = ASCENDING, and in descending order
   otherwise. The sequence to be sorted starts at index position lo,
   the number of elements is cnt.
 **/
void bitonic_merge( int lo, int cnt, int dir )
{
  int k = cnt / 2;
  int i;
  _Pragma( "loopbound min 0 max 16" )
  for ( i = lo; i < lo + k; i++ )
    bitonic_compare( i, i + k, dir );

  if ( k > 1 ) {
    bitonic_merge( lo, k, dir );
    bitonic_merge( lo + k, k, dir );
  }
}
```

Listing 5.1: Code snippet of the ‘bitonic_merge’ function from the TACLeBench Bitonic program

5.5 Use Case: WCET Hybrid Analysis

5.5.1 Experimental Setup

In order to verify the feasibility of the hybrid methodology and functionality of the COBRA framework, we are using benchmark programs from the TACLeBench project. The TACLeBench is a benchmark project of the TACLe community to evaluate timing analysis tools and techniques in order to compare their performances [60]. Each module of the COBRA framework is continuously tested with benchmark programs from this test-bench to verify the integrity and precision of the generated models. Apart from functional testing the toolchain, we need to evaluate the performance of the hybrid methodology. For this experiment, we performed WCET analysis experiments on the OCTA-Mini from IDLab. This board is a prototype platform for embedded low-power wireless IoT sensor devices and is equipped with an ARM Cortex-M3 CPU. We performed the hybrid analysis on three different benchmark programs, e.g. bitonic, bubble sort and recursion; and compared the performance with the static and measurement-based analysis techniques. The WCET experiment presented here is analogous for estimating the WCEC with the energy analyser, therefore the WCEC use case is out of scope.

5.5.2 Results

Figure 5.7 shows the WCET results of the tests performed. For each test, we compared the hybrid analysis used in the framework with the static and measurement-based analysis. The bar graphs in Figure 5.7 are normalised in percentages compared to the real WCET. With this representation, we can observe the deviation of the resulting WCETs with the actual one. Firstly, the hybrid results in the bar graph are obtained from the COBRA-HPA toolchain after the profiling step of the Profiling Assistant. Secondly, the static results are determined by finding the longest feasible path that results in the longest execution time. When the worst-execution path is found, a corresponding input set is composed which will trigger this path. The static analysis of the tests was calculated and verified manually. These results are therefore considered as the absolute WCET value. Finally, the measurement-based results are acquired by executing each program with random generated input sets for considerable long time. In the following section, we will discuss the significance and results of each test shown in Figure 5.7.

5.5.3 Discussion

Bitonic

The bitonic benchmark is a sorting network based on the bitonic algorithm. This sorting algorithm starts dividing the input set into pairs that it sorts alternating ascending and descending. Next, it groups consecutive blocks into bigger blocks, sorts them according to the desired order and repeats the process until the entire input set is sorted. This benchmark has an input array of 32 elements. For this test, we made a small change to the input initialisation method as it is implemented in the benchmark suite. Instead of using the same input set for each run, we assume that each element of the array can vary in the interval [0, 31].

As we compare the results in Figure 5.7, we notice that the measurement-based approach makes an underestimation of the WCET by almost 0.5%. This means that the

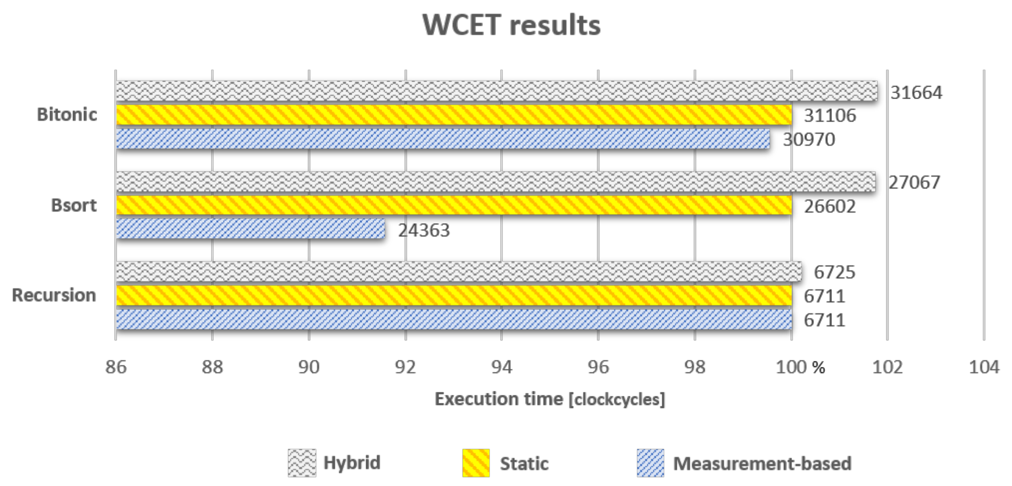


Figure 5.7: WCET prediction results comparison of three analysis methodologies. Results are normalised to the real WCET of each benchmark program.

worst-case input was not tested. However, if we want to exhaustively test the benchmark with every possible input set, we would have 32^{32} different valid sets. When we consider the average execution time on the ARM Cortex-M3 with a clock frequency of 48 MHz, it would take over 3×10^{37} years to measure every input set! This approach is impossible to perform. Therefore, a random input generator delivers the required input data. Nevertheless, the measurement-based approach reached pretty close to the WCET bound for only measuring a fraction of all input sets.

The hybrid analysis on the other hand, has an overestimation of nearly 2% for this benchmark. However, an overestimation of the WCET is acceptable in contrast to an underestimation. The higher boundary is due to the basic approach of the analysis tool used. Each basic block is measured independently from one another. The worst result of each block is then used in the static analysis. However, a worst-case path in one block would not necessarily be followed with the worst-case path of another block when a worst-case input set is given. This phenomenon is referred to as timing anomalies as explained in Section 2.1. Nevertheless, changing the block sizes present the possibility to counter the effect when the upper bound would become too high.

When comparing the profiling effort of the hybrid method with the measurement-based approach, a tremendous improvement is noticed. The block with the largest input set has ‘only’ 1024 combinations which is equal to a complete profiling cycle of 1 s. A complete hybrid analysis took just a few minutes in total compared to the measurement-based approach.

The static analysis was performed manually by determining an input set that would result in the WCET of the program. At this point, it was rather straightforward to find this set for the algorithm because of its simplicity. However, when the program is larger and more complex, it becomes less obvious. With the hybrid method, we are able to create a higher-level abstraction of the program which simplifies the static analysis in exchange for precision of the upper bound.

Bubble Sort

The bsort benchmark is a bubblesort algorithm. This sorting algorithm iterates $n - 1$ times over the input array while it sorts the pointer element and its next neighbour to the requested order. The order will eventually propagate from the end to the begin of the array. In the TACLeBench, this benchmark originally has an input array of 100 fixed elements. Similar to the previous test, we made small adjustments to the code. The size of the input array is reduced to 25 elements. This change allows us to perform the measurements with a better precision without an overflow of the timer counter, i.e. a timer resolution equal to the CPU clock frequency. The second adjustment is to allow the input array to be randomised instead of using a fixed input set.

The results in figure 5.7 show the same trend as the bitonic test. The same problems arise with the measurement-based method. However, an underestimation of more than 8% is recorded now. The total number of combinations for the input set is extremely high, which makes exhaustive testing not feasible.

An equal trend is observed for the hybrid methodology for this benchmark. An additional feature of the bubble sort benchmark is the use of nested iteration statements. These code constructions need to be handled with care when using the hybrid analysis. When the inner loops have variable loop bounds, it could lead to an exaggerated pessimistic upper bound. In order to support analysis tools, all TACLeBench programs are annotated with flow facts, e.g. iteration statements are annotated with the minimum and maximum number of iterations. These loop boundaries hint our tool about flexible inner loops. If this is the case, the tool is able to tackle the problem in two ways. A first approach is to make an abstraction of the encapsulating loop, so it will take the variable loop in consideration. The other approach is to lower the abstraction to the body of the loop and statically determine the total number of iterations if possible. In this experiment, the second approach was applied.

The annotations are an excellent feature to tell analyser tools more specific details about the context and flow facts of a given program. However, when these annotations are not present in the code, the tools have to try determining the boundaries by themselves. Nevertheless, when the boundaries could not be determined, the COBRA framework is still able to adaptively change the block size creating a higher abstraction of the unknown loop boundary.

Recursion

The recursion benchmark calculates the Fibonacci sequence by invoking recursive method calls until the desired Fibonacci number is found. This benchmark only takes one input, namely the requested Fibonacci number. This test is a good test-case to evaluate the functionality of our tool on recursive algorithms.

As there is only one fixed input in the program that results in one fixed trace through the code, we can presume there will be only one final result for the execution time that equals the WCET. The results for the static and measurement-based analysis are therefore exactly the same as shown in figure 5.7.

The hybrid results only show a small overhead of 0.2% compared to the static result. Determining the WCET of a recursive algorithm is a more complicated task for the static and hybrid analysis. The total state space will rapidly grow with each new recursive call which makes finding the longest feasible path hard or even impossible to solve. In the case of the hybrid approach, our tool would consider the recursive call as a separate basic block. This results in an abstraction of the recursion itself by measuring all next recursive invocations. However, it could be desirable to go deeper into the recursion itself to increase the precision by discovering basic mathematical series if present, or alternatively by partially 'unrolling' the recursive loop.

5.6 Conclusion

In this chapter, we discussed our hybrid approach to estimate the WCRC by combining two existing methodologies. This hybrid methodology creates an opportunity to merge the benefits of static analysis with a measurement-based approach. In order to validate this approach, we created the COBRA-HPA framework that allows us to split code into blocks according to the hybrid methodology. The toolchain is able to parse C-source files and generating a corresponding hybrid block scheme. The tool provides an easy-to-use control flow to generate executable testbenches for profiling the code behaviour, i.e. measuring the WCET and WCEC of hybrid blocks.

With the creation of the automated testbench and its integration with the COBRA framework, we have established the tools to facilitate the process of WCET and WCEC analysis on resource-constrained devices, such as batteryless IoT devices.

Finally, we profiled basic benchmarks of the TACLeBench with the framework and compared the performance to the static and measurement-based analysis. We conclude for **RQ 1** (Section 1.3) that our approach reduces the number of measurements, and thus the total analysis effort significantly compared to the measurement-based method. Additionally, the flexibility of the hybrid block sizes allows to create higher level abstraction of the blocks and therefore providing the possibility to find a balance between the static complexity and the overall accuracy from the measurement-based component.

Chapter 6

Estimating the Worst-Case Resource Consumption using Machine Learning

In the previous chapter, we have presented a hybrid analysis approach to determine the WCET and WCEC of a software task. Nonetheless, it is still difficult to acquire early insight of the WCRC during development with this methodology as it relies on the physical hardware and compiled binary code to measure the resource consumption on the device. In addition, the measurement process itself is time consuming to perform. Therefore, we extend the hybrid methodology by including ML techniques to predict the WCRC of the hybrid blocks instead of physically measuring it on the system.

6.1 Machine Learning

AI is a broad term that describes a collection of techniques that enables machines to sense, reason, act and adapt. While this concept was already used in 1955 [18], it recently has gained great popularity and breakthroughs in new technologies due to the advancements in the availability of computational resources to run these models and large datasets. On the Gartner Hype Cycle for Emerging Technologies of 2021, which highlights 25 potential technologies that will have an impact on the industry and society within the next two to ten years, more than five technologies on the graph are within the domain of AI [167].

Within the domain of AI, we find the subgroup of ML techniques. This group contains the algorithms whose performance improve with statistical methods as they are exposed to more data whether or not over time. ML is therefore able to adapt or predict output for new input based on observed patterns in previously seen data during training. This approach allows us to find patterns between different attributes in the dataset without having to manually program these relations in the conventional approach with rule-based systems. The weights and biases of the prediction model gets tweaked during training based on the data it gets fed and are therefore referred to as ‘black box’ models.

The resource consumption of a software task depends on the instructions of the followed program trace on a specific hardware platform. In the case of the WCET, we are interested in the events and interactions that results in the longest path in time of the software task, such as instructions, input data, pre-emption, caches, etc. All these soft- and hardware characteristics can be described as a collection of attributes for a given software task on a specific platform. As a result, it is possible to develop models with these attributes to make predictions on the WCRC. Creating a generic model with classic rule-based programming to assess a given code base for a random platform is nearly impossible. Therefore, we research the possibilities to employ ML techniques to create an estimation model for the WCRC.

Bonenfant et al. [81] propose a method to approximate the WCET early on in the development by applying ML. Their goal is to characterise source code in order to find a formula for a specific target platform and compiler toolchain, which will be achieved by training a neural network on a set of test programs. The prediction is based on the worst-case event count, such as mathematical operations, control flow and memory access events. A static analysis characterises a program by counting events, which would lead to the worst-case result. The training is performed by matching the worst-case event counts with the provided WCET estimates of the test programs. Eventually, the trained network will then predict the WCET of a given program with the event counts from the static analysis and the trained formula of the tested hardware [81]. In addition to the worst-case event counts attributes, the author suggests making a classification based on the code style attributes, e.g. lines of code, loop nesting, auto-generated or handwritten code, etc.

We believe that ML presents a valuable solution to make early WCET predictions by classification of the complex problem statement. However, the approach presented by Bonenfant et al. might suffer from oversimplification [81]. The suggested characterisation by event counting makes a high abstraction from the source code so that valuable information of the code flow gets lost. At the end, the code flow and hardware interactions will become too complicated if a program is classified based on the entire code base.

The ML approach provides us worst-case estimates right from the start of the developed process based on the available system attributes. Therefore, rough estimates with a large deviation are available in the early stages of the development phase. However, the early predictions will provide us insight to explore the hardware design space and exclude target platforms and configurations which will not be suitable. When the design and development process continues, more accurate attributes become available resulting in a gradual improvement of the WCRC estimation. At that point, it is important to verify and gain trust in the trained model to obtain sound predictions.

In other research, Griffin D. et al. used a Deep Learning Neural Network (DLNN) to model the influences of other processes on shared resources [168]. The attributes used for the DLNN are the built-in Performance Monitoring Counters (PMC) in the multi-core processor. These counters keep the number of occurred events, e.g. number of cache misses, pipeline flushes, etc. The output is an interference multiplier which will be the worst-case overhead originating from the interferences of other processes. The methodology shows promising results with small underestimation errors on the final WCET [168]. However, this approach still requires a regular WCET analysis on a single core without interference to acquire the final WCET, as the obtained results needs to be multiplied with the interference multiplier. Additionally, the number of PMCs that can be monitored at a time is limited. Therefore, it is required to run multiple measurements on the platform to acquire different parameters [168], which increases the analysis effort.

While other research already applied ML techniques to estimate the WCET, our approach to apply ML to create WCEC estimation models is a novel approach in the SOTA. With the right approach, we are convinced that a methodology can be created that based on ML is able to perform accurate WCRC predictions for any given architecture

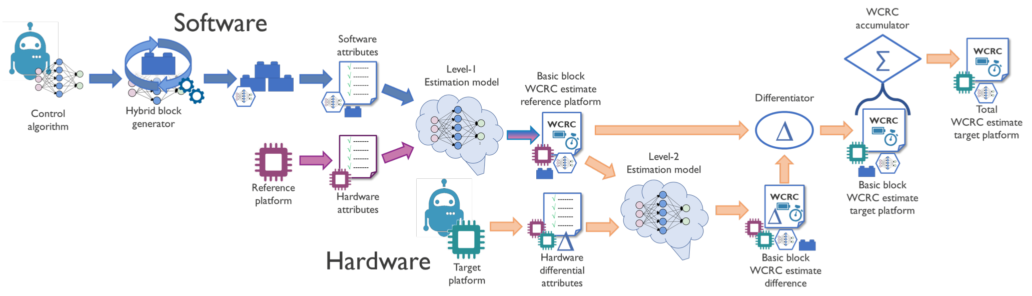


Figure 6.1: Schematic overview of the ML-based hybrid analysis methodology.

by combining/improving the hybrid methodology with ML and characterising the source code at lower levels (i.e. hybrid blocks) to avoid oversimplification (e.g. loss of code flow information, etc.) or too complex classifications.

6.2 Extending the Hybrid Methodology with Machine Learning

ML-based WCRC analysis is a new concept that is still researched in the SOTA. As discussed in Section 6.1, other research is being conducted to WCET estimation. In the work of Bonenfant et al. [81], they concluded that the acquired formula of the source code characteristics in the experiments was too complex and imprecise. In addition, a new neural network needs to be trained for each target hardware platform and toolchain combination. Therefore, we propose a new approach for this methodology. The innovation of this proposal lies in the integration of the target hardware/toolchain attributes in the evaluation and the combination of ML with the hybrid analysis. Therefore, providing the opportunity to have a cross-platform methodology.

In order to create a hardware independent approach, two separate ML models will be created to integrate the platform dependent attributes. The first level will be trained to predict the WCRC on a reference hardware platform. The secondary level predicts the deviation on the WCRC with the desired platform by taking differential attributes between the reference and target platform into consideration. Additionally, the integration of basic blocks from the hybrid methodology allows us to have a gradual selection of precision depending on the developer's needs. The regression can be shifted from an application level for a fast rough estimation to a fine-grained detailed analysis on small hybrid blocks for a higher precision. The combined approach of integrating ML with the hybrid methodology is illustrated in Figure 6.1. The implementation of this approach consists of three main parts.

6.2.1 Code Attributes

The first step is selecting a set of code attributes that have a significant influence on the estimation of the resource consumption. This step is highlighted by the upper left blue path in Figure 6.1 and is performed on the reference platform. This model is solely trained on code attributes, such as code size, number of operations, variable sizes, etc. These attributes originate from application-level characteristics, as proposed by [81]. However, we expect that the characterisation complexity will become too high for larger applications and that the accuracy of program level analysis will be significantly lower due to oversimplification. Therefore, we propose the integration of hybrid blocks attributes to improve the accuracy and tackle the characterisation complexity/oversimplification problems.

The hybrid blocks are generated from the source code as shown in Figure 6.1. The size of a hybrid blocks is variable from small, single instruction blocks to bigger, procedural blocks. This depends on the selected strategy. The objective is to find a balance between accuracy, classification complexity and computational complexity. Smaller blocks are easier to classify due to the smaller number of characteristics that needs to be incorporated in the classification. Additionally, we expect the accuracy will be higher as less abstraction is applied to the result. However, the computational complexity will eventually increase, because more estimations need to be statically combined at the end.

6.2.2 Hardware Attributes

The second step is extending the previous model of the first step with hardware and compiler specific attributes, which is a novel approach compared to the proposal in [81]. This step is illustrated by the purple path in the lower left corner of Figure 6.1. This path will improve the precision of the software attributes of the previous step by including extra hardware/compiler-related attributes. The optimisations techniques used in hardware and software, such as pipelining, caches, compiler optimisation, etc.; are introduced to improve the average scenarios. However, these techniques could have adverse effects on the WCRC. By incorporating these attributes to the analysis, we hint the model about these system attributes in order to improve its predictions.

Additionally, a hardware independent prediction model is created by integrating the hardware attributes in the trained model. This would allow us to switch to another platform of the same family by simply adjusting some parameters without retraining a completely new estimation model. As a result, only a single trained model is needed for a wider range of target platforms in comparison to the created model in the first step. So far, a separate prediction model needs to be trained for each target family (i.e. hardware/toolchain). However, this problem is tackled in the last step where a differentiation model estimates the relative WCRC in respect of a reference platform.

6.2.3 Hardware Differential Attributes

The final step consists of a second trained model that is connected to the first layer. This model estimates the relative WCRC of a target platform by classifying the result based on the differential with a reference platform. The delta (differential) attributes describe parametric differences of hardware/compiler features between a reference platform and the target platform.

Through the addition of this differential model, we can acquire WCRC estimates without actually executing any code on the target platform. The orange path on the right-hand side of Figure 6.1 provides an overview of this step. The output of this model will estimate the relative difference in clock cycles of the target platform with the reference platform used during training. The differentiator block in Figure 6.1 will compare the differential output value with the reference prediction. The calculated result is the final prediction of the WCRC for the target platform. Nevertheless, if the precision drops dramatically or the model complexity becomes too high, we can fall back on one of the following alternative approaches:

- Evaluating other ML models or combining multiple in an ensemble learning model to improve the precision;
- Restricting the analysis on smaller hybrid blocks, which increases accuracy and lowers the classification complexity. However, the WCRC accumulator (static analysis) becomes more complex and thus resulting in a higher computational complexity;
- In the worst-case, a different reference platform can be used for each family of processors (i.e. processors with a similar architecture). Still, this will require training of multiple estimation models.

The proposed third step provides an estimate of the WCRC based on the results of another platform used as reference. The extrapolation in this approach is achieved through the use of ML. In the current stage of this research, the focus is on selecting the most appropriate software features. In the first phase, a set of attributes needs to be derived from the source code. These sets of code attributes are then used as features to train the ML layer and eventually estimate the WCRC.

6.3 Feature Engineering

After generating the hybrid blocks, a 'value' for each attribute (i.e. feature) is obtained from these blocks. Extracting these features from source code is a time consuming and error-prone task. Additionally, to train an ML layer that is able to provide a 'solution', i.e. WCRC estimate, we need to apply a supervised learning strategy [169]. As a result, a large, annotated training set needs to be created. In order to assist us in analysing and collecting features, we developed the Feature Generator extension for the COBRA-HPA framework. The implementation of this tool is explained in Appendix A.4.

6.3.1 Software Features

Software attributes are numeric representations of a hybrid block's code that are used as feature input for the ML models. Table 6.1 lists the software attributes that are extracted from each hybrid block. The selected features are picked in previous experiments based on visual inspection of hybrid blocks and identifying those code-related features that characterise the code execution (e.g., actual instructions) and have a significant impact on the platform as discussed in Chapter 3 and 4. These features are automatically extracted from source code, and therefore, related to the C-code syntax. Most features are based on the number of occurrences per classified operation type. The feature list is kept limited to reduce the complexity in these first experiments. Nevertheless, this list is expandable so it can be improved on in future research by employing feature engineering techniques [170], [171], such as correlation analysis between features, and feature extraction strategies [169].

6.4 Prediction Models

At the core of the prediction mechanism is a target specific trained ML layer. This layer receives a set of features, as the ones in Table 6.1, and provides one output value that

Table 6.1: Code attribute types from the hybrid blocks used as input features for the prediction models.

Attribute	**Description**
Additive	Number of add/subtract operations. [+, ++, +=, -, –, -=]
Multiplicative	Number of multiplication operations. [*, *=]
Division	Number of division operations. [/, /=]
Modulo	Number of modulo operations. [%, %=]
Logic	Number of logic operations. [&&, ——, !]
Bitwise	Number of bitwise operations. [&, &=, —, —=, ˆ, ˆ=, ˜]
Assign	Number of assignments. [=, +=, ++, -=, –, *=, /=, %=, <<=, >>=, &=, ˆ=, —=]
Shift	Number of shift operations. [>>, <<, >>=, <<=]
Comparison	Number of comparison operations. [==, !=, <, <=, >, >=]
Return	Classification if a return statement is present.
Evaluation	Number of logic evaluation statements. [if, switch, ?]
Iteration	Number of iteration statements. [while, for, do-while]
Local variables access	Number of accesses (read/write) to local variables.
Local array access	Number of accesses (read/write) to local arrays.
Global variables access	Number of accesses (read/write) to global variables.
Global array access	Number of accesses (read/write) to global arrays.

resembles its estimation for the WCRC of the hybrid block. The next step is to select an ML technique that would be suitable to perform the task. For our experiments, we have made a selection of regressions and DLNN prediction models to compare.

6.4.1 Regression Models

In order to estimate the WCRC, we need predictors (i.e. features) to predict a numeric value. This approach is referred to as regression [169]. A regression model tries to fit a function h, i.e. hypothesis, through a set of points. The goal is to find a hypothesis h which approximates these points. We are able to find an infinite number of functions h in the hypothesis space H that perfectly fit all the points from the given training set. Nevertheless, this does not result in a useful prediction model. A good regression hypothesis is a model that is able to correctly predict the value of previously unseen datapoints, and thus generalised the problem well. The model tries to minimise the error between the observed datapoints and the target label in the training set. An optimisation algorithm is used during training to minimise this error, such as gradient descent, stochastic gradient descent [169].

A wide range of different regression models exist that use different kernels to find a good fitting hypothesis. The most optimal regression model depends on the correlation between the different features, for example linear, higher-order polynomial, etc. Therefore, it is a good practise to try multiple models side by side, as these relations in the data are in most cases unknown. For our experiments, we have made a selection of seven types of regression models.

- *Linear Regression* is a linear model that creates a linear combination of the features, and minimises the residual sum of squares between the predicted value of the model and the provided label;

- *Polynomial Regression* extends the linear regression model with a higher order polynomial function as hypothesis;
- *Decision Tree Regression* predicts its values based on simple decision rules that are inferred from the features. Each node in the tree contains a selection statement that results in selecting a child node until a final leaf node provides the output of the model;
- *Random Forest Regression* is an ensemble model that combines multiple decision tree models on subsets of the dataset, and finally takes an average output from each of these trees;
- *Support Vector Regression (SVR)* uses a kernel function to fit a vector with an upper and lower margin, analogue to a street, through all points of a class. The goal of the regressor is get as many datapoint on this 'street'. The kernels we used in our experiments are the Linear and Radial Basis Function (RBF) kernels;
- *K-Nearest Neighbours Regression* uses a subset of datapoints from the training set to find the K nearest ones in distance to the new point in order to make a prediction for this new label;
- *Ridge Regression* is similar to the linear regression but imposes a penalty factor on the size of the coefficients. This approach makes the model more robust to random errors in the data for highly correlated features.

6.4.2 Deep Learning Neural Networks

Deep learning is a technique in the ML domain which is gaining popularity in the field [169]. The concept of deep learning techniques is based on the inner workings of neural transmitters in the human brain. The predictor model consists of an interconnected network of neurons that communicate with each other. Therefore, the models are referred to as neural networks. Each of the neurons passes values from one layer to the next (i.e. hidden layers) until it reaches the final output layer. These values or 'signals' propagate between the neurons through links. The *signal* x that traverses a link is multiplied by the *weight* W of that link. The incoming signals are accumulated in the neuron and receive an offset by the *bias* b of the neuron. Finally, a threshold in the neuron determines how the signal should propagate to the next layer. This threshold is defined by an *activation function* f. The output signal of the neuron will then propagate to the next layer until the output layer is reached. Figure 6.2 provides an example of a fully connected neural network to illustrate the operation.

The goal of this approach is to feed the network with reference data instead of rules in order to allow the model to learn how to solve the problem itself. ML techniques can be categorised according to the data problem that needs to be solved, for example:

- *Classification* labels the data with predefined meaningful classes;
- *Regression* labels the data with a numeric value based on the input data;

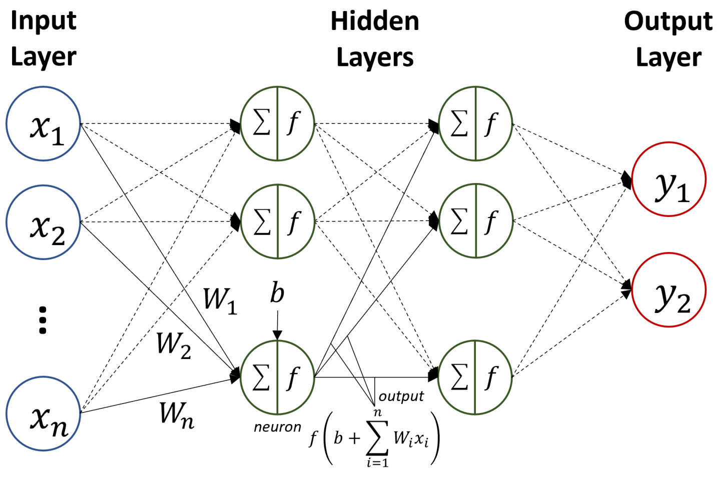


Figure 6.2: Example of a fully connected neural network. The output of a neuron in the hidden layers is equal to the activation function f of the sum of the bias b with the accumulated incoming signals x_i, multiplied by the weight W_i of their respective link.

- *Clustering* groups the data into an unknown number of unlabelled classes;
- *Rule extraction* determines propositional rules based on relations between attributes in the data;
- *Anomaly detection* determines if the data conforms to the expected pattern in the dataset.

Depending on the strategy to handle the problem, we classify each technique as (semi-) supervised or unsupervised learning [169]. Supervised strategies use labelled data to train a mapping between in- and output, in contrast to unsupervised strategies that only receive unlabelled data, i.e. input sets with no corresponding output. The latter requires the model to learn the data distribution/classification on its own. In order to determine the WCRC, we need a predictor model which is able to estimate a numeric value, i.e. WCET or WCEC, based on features. To train such a model, we provide training sets which are annotated with the corresponding output WCRC. For this case, a supervised learning model is required that solves a regression problem.

Oyamada et al. [172] study the possibility to apply deep learning for predicting the WCET of software applications. They divided the assembly instructions into four different categories, i.e. integer, floating-point, branches and load/store operations. These categories were used to predict the number of clock cycles. For their experiment, they used two Feed-Forward Neural Network (FFNN) with one input, one hidden and one output layer. The training data was classified in two groups according to their CFG, namely ‘data-dominated’ and ‘control-dominated’ applications [172]. The distinction is made by the ‘CFG weight’ that they define as a threshold of 1.95 for the division of the weighted arcs by the number of nodes in the CFG [172]. This separation is performed because the processor features respond differently to the structure of the CFG, e.g. branch prediction, caches, etc. For each type, a separate FFNN was trained. The generic estimator had an error ranging between -31% and 33%. For the specialised estimators however, they improved the error slightly to a range of -32% and 26% with a lower mean error. These results look promising, but the errors are rather large to be used as upper bound. However, obtaining such results during (early) development stages would provide a large benefit for the system developers. Therefore, we are performing analysis on code level instead of assembly instructions.

We believe a methodology can be created based on ML that is able to provide early insight into the WCRC of software for a given architecture. Therefore, we propose to enhance the hybrid analysis approach with deep learning. Therefore, we have selected two DLNN models to experiment with.

Feed-Forward Neural Networks

An FFNN consists of an input layer, one or multiple hidden layers and an output layer of neurons, as shown in Figure 6.3. Each neuron is connected to a number of neurons of the previous layer. A weight (W) is assigned to each connection while each neuron receives a bias (b). During a forward pass, the output of the previous neurons is multiplied with the weight of the connection plus the bias of the target neuron. This value is then inserted into the neuron's activation function to become the final output of the neuron. In order for the network to learn something, we apply a back-propagation algorithm to optimise each weight and bias in the network according to the error made, e.g. gradient descent, particle swarm optimisation, hill climbing, etc.

Tree Recursive Neural Networks

The CFG of a software application can be translated into a hierarchical tree structure that represent the order of each block. This representation makes it straightforward to create higher abstraction blocks. The Tree Recursive Neural Network (TRNN) is an approach to incorporate this hierarchical information into a neural network. A TRNN consists out of normal FFNNs that are reused in different nodes of a tree-like structure. In order to achieve this tree structure, we require two different networks, i.e. analysis- and synthesis-network. A schematic representation of these two networks is shown in Figure 6.4. The functionality of the TRNN analysis-network is identical to the FFNN. The TRNN approach has an added value due to its hierarchical approach of the synthesis-network. This network performs the combination step of the WCRC results of each child block without the need to perform a static analysis and thus lowering the computational complexity of the analysis.

6.5 Use Case 1: WCET Hybrid Analysis with Regression Models

6.5.1 Experimental Setup

The Hybrid ML-based methodology for WCET analysis is a new approach. For this first use case, we have selected a set of code-related attributes that are listed in Table 6.1. These attributes are modelled in the Feature Selector tool, so we are able to easily obtain features from the benchmark code. For this first prototype, we kept the size of the generated hybrid blocks small and the number of features limited. For example, iteration

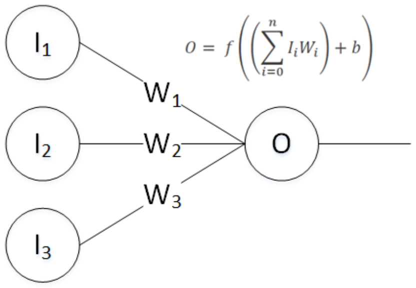


Figure 6.3: Schematic of a feed-forward neural network with activation function.

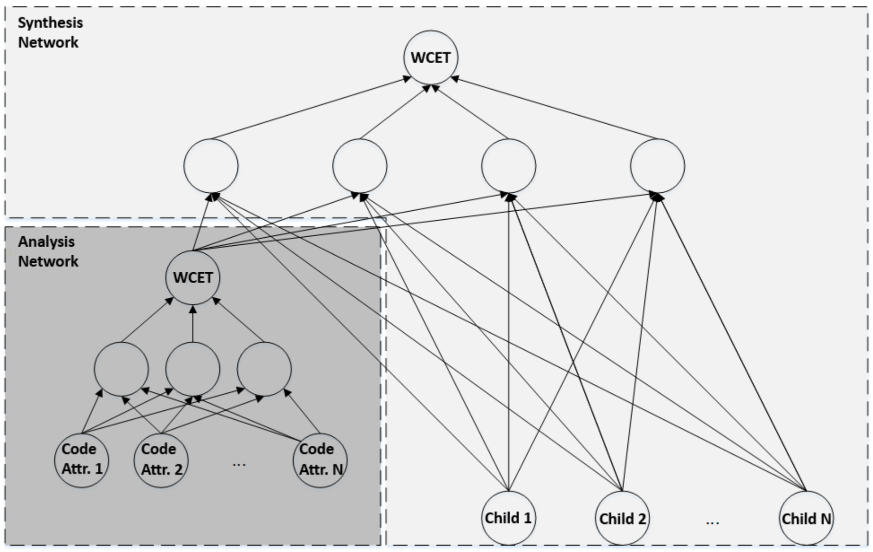


Figure 6.4: Schematic of a tree recursive neural network.

statements were unrolled and not modelled as independent features, as extra features would require more training data which would made the prototype too ambitious. The blocks are generated from benchmark code of the TACLeBench initiative [60].

For this experiment, we have trained a set of standard regression models, as listed in Table 6.2. All models in this experiment were implemented with the Scikit-Learn framework [173]. This framework provides a wide range of tools to create, train and validate ML algorithms in Python. The training and validation sets have a size of respectively 75 and 25 blocks that are used in a 4-fold cross-validation process. Each of those blocks are provided with all attributes from Table 6.1 and annotated with a WCET value to train the model and verify the results.

The target platform used in this experiment is an ARM Cortex-M3 CPU on the EZR32 Leopard Gecko board of Silicon Labs [158], where the WCET of each block was measured according to the hybrid methodology explained in Chapter 5. To evaluate and compare the performance of the regression models, we calculated the Mean Absolute Percentage Error (MAPE) on the validation sets. This error metric takes an average of the absolute

Table 6.2: 4-Fold cross-validated MAPE on the validation set for each trained WCET regression model.

Regression Models	**MAPE**		
	Best	**Average**	**Worst**
Linear	27.17%	77.85%	138.6%
Polynomial (2nd Degree)	79.14%	300.6%	669.3%
Decision Tree	20.45%	33.9%	53.53%
Random Forest	23.18%	51.88%	84.63%
Support Vector (Linear Kernel)	**16.04%**	**27.28%**	**40.24%**
Support Vector (RBF Kernel)	16.11%	49.78%	83.95%
K-Nearest Neighbours	33.91%	38.97%	42.38%
Ridge	28.28%	77.83%	130.4%

percentage errors without regard of the sign and is commonly used in forecasts [174] (e.g., regression problems) and other research using prediction models [175], as it is very intuitive to interpret the relative error. The MAPE metric is defined as Equation 6.1 with h the prediction function (i.e., hypothesis), X the input set of features, m the size of the input set (i.e., number of datapoints) and y the set of actual values or corresponding labels of each datapoint.

$$MAPE(\mathbf{X}, h) = \frac{100}{m} \sum_{i=1}^{m} \left| \frac{y^{(i)} - h(\mathbf{X}^{(i)})}{y^{(i)}} \right| \quad [\%] \tag{6.1}$$

6.5.2 Results

The best, average and worst MAPE scores on the validation sets for each regression model is shown in Table 6.2. The best performing model in this experiment is the SVR with a linear kernel, followed by Tree and K-Nearest Neighbours regression. The graphs in Figure 6.7 plot the predicted WCET values of the validation set with respect to the corresponding real measured results. The closer a point is vertically located to the dotted line, the smaller the prediction error is. However, the box plots of the error distributions in Figure 6.5 provides interesting insights. If we remove the outliers, we see that the 2nd Polynomial regression actually performs significantly better than initially thought when comparing it to the result of Table 6.2. As the MAPE is sensitive for the large outliers the model predicted, e.g. the model had prediction errors (MAPE) up to 4000% and higher!

The MAPE is an excellent metric to gain insight in the performance of the prediction models. Nevertheless, there is no indication if the models are mostly over-or underestimating the WCET. Within the context of WCRC, it is important to keep the upper bound prediction above the real worst-case to avoid compromising the functional behaviour of the system. Therefore, we plotted box plots of the Mean Relative Percentage Error (MRPE) for each model in Figure 6.6. MRPE calculates the percentage difference between the predicted and actual values but keeps the sign of the error.

After validating the regression models on small hybrid blocks, we performed an experiment on three TACLeBench applications, as in Section 5.5, to test the performance

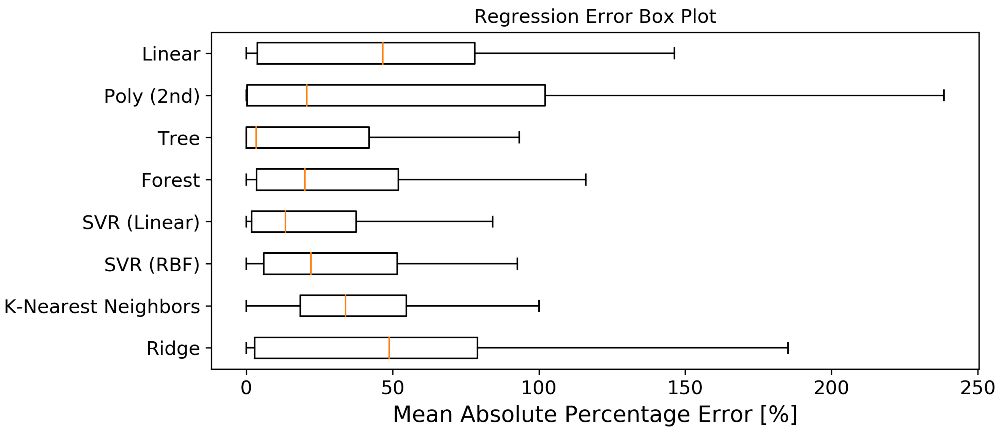


Figure 6.5: MAPE box plots of the WCET regression models with removed outliers.

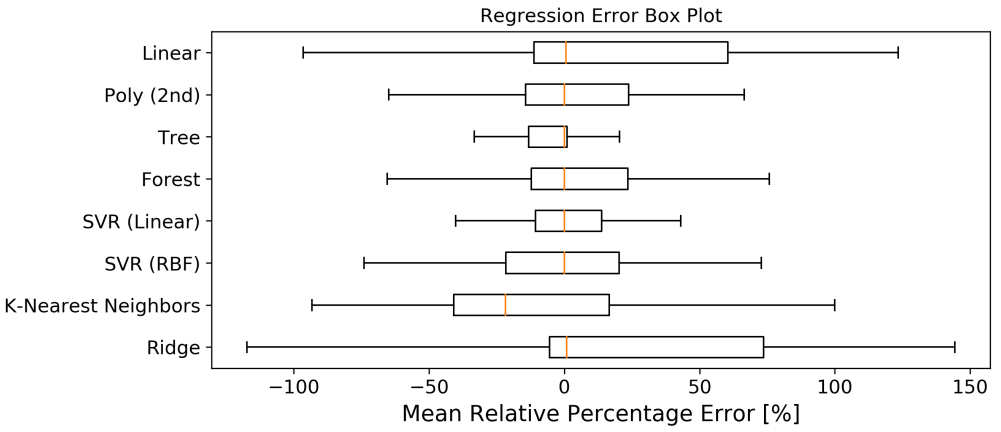


Figure 6.6: MRPE box plots of the WCET regression models with removed outliers.

of this methodology. The results of these experiments are shown in Table 6.3. This table shows the errors of each model's estimation of the WCET. A negative and positive error resembles respectively an under- and overestimation of the real WCET.

The results in Table 6.3 show good results for the Decision Tree and SVR with linear kernel models. However, they perform significantly worse for the *Recursion* benchmark compared to the other regression models. This benchmark has a small code base which repeatedly gets called recursively. In this case, a small error on a hybrid block will result in a rising total error when the number of recursive calls increases.

6.5.3 Discussion

In the context of the WCET, we are mostly interested in the worst performance as we need to evaluate the error margin in order to obtain the upper bound. The results of

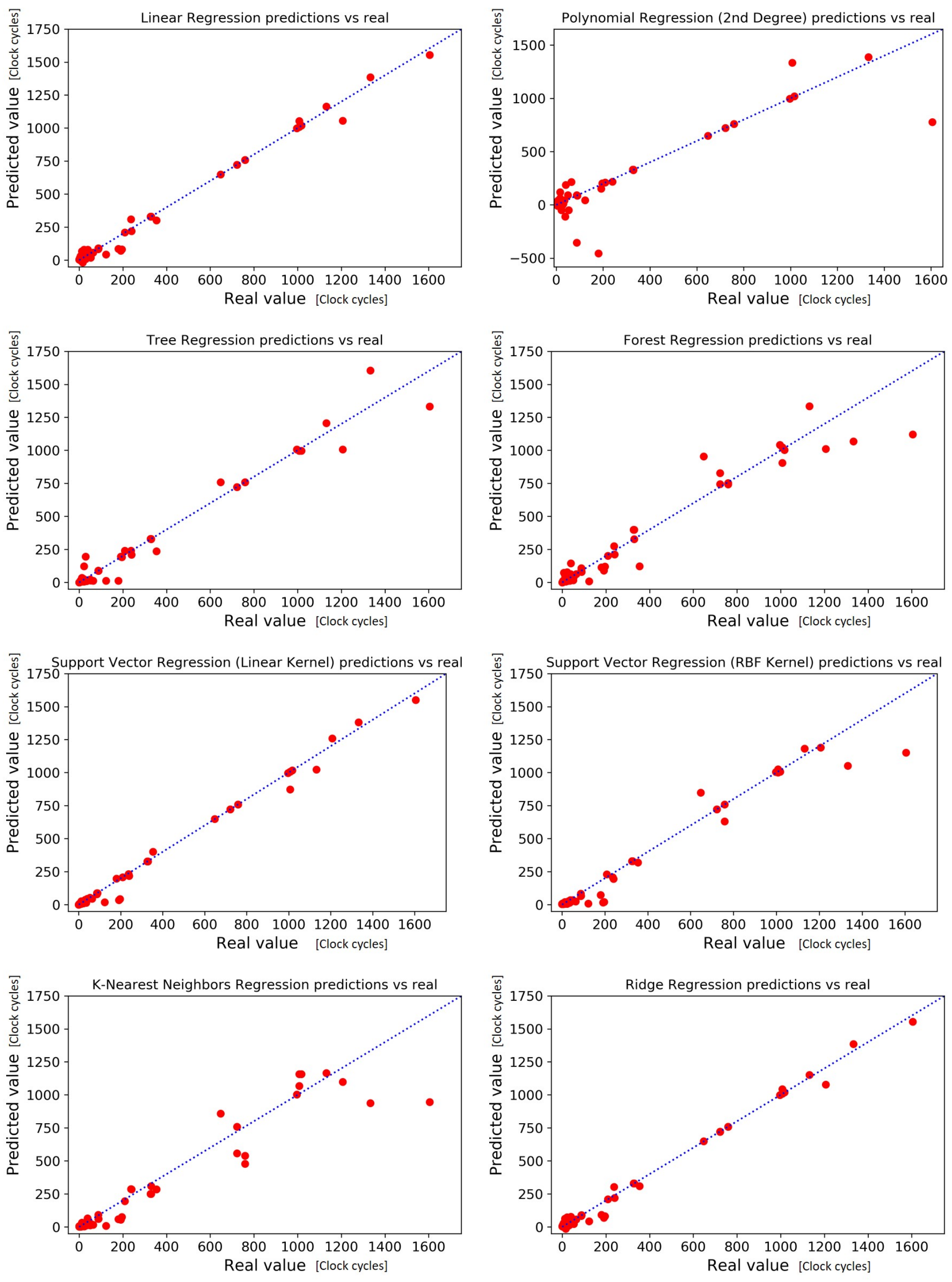


Figure 6.7: Predicted vs. real WCET upper bounds for trained regression models on the validation sets.

Table 6.3: Prediction errors of the hybrid WCET regression models on three TACLeBench applications.

Regression models	**Bitonic**	**Bsort**	**Recursion**
Linear	−49.3%	102.2%	−0.2%
Polynomial (2nd Degree)	100.2%	−266.3%	−10.9%
Decision Tree	18.1%	18%	−52.8%
Random Forest	−11.8%	113.7%	−14.6%
Support Vector (Linear Kernel)	−24%	8.5%	−55.3%
Support Vector (RBF Kernel)	−31.9%	−36.6%	−45.6%
K-Nearest Neighbours	−45.9%	38.5%	−54.1%
Ridge	−47.1%	56.8%	0.5%

this experiment, i.e. Table 6.2, indicate that the SVR with a linear kernel has worst-case the lowest MAPE of 40.2%. However, it reaches an average error of 27.3%. In addition, we see in Figure 6.7 that the SVR with linear kernel accurately predicts all datapoints of the validation set with just a few small outliers when compared to the other trained regression models.

The better performance of the linear kernel models is probably due to the linear characteristics of the trained features on a 'simple' architecture, e.g. single core, no caches, etc. Therefore, these models are better in generalising the problem, as more complex models will overfit the solution [169]. The well-performing SVR model tries to match the data such that the distance between the datapoints and the fit, which is referred to as the supporting vectors, is maximised [169]. On the other hand, the Decision Tree model partitions the datapoints in clusters for which a value is assigned to. Good results were achieved with these models; however, this approach estimates discrete values. If we want to have accurate output values, we need a high partitioning of the data space, and therefore obtain large complex trees.

An upper bound prediction should try to approximate the real WCET as tight as possible to minimise pessimism. Nonetheless, it should never underestimate the actual WCET as this could result in disastrous behaviour of the system, e.g. failing to perform an emergency break on time. In Figure 6.6, we have calculated the relative error to investigate the distribution of under- and overestimated predictions. We note that all regression models are prone to underestimation with their median close to the boundary. Nevertheless, we are able to create an error model based on the validation errors to introduce pessimism to the predictions. This will add a margin to acquire safe upper bound estimates.

Since the goal of the experiment was to examine the feasibility of applying ML techniques to predict WCET values, we did not tune any of the hyperparameters. While the results of Table 6.2 seem bad with relatively high errors, we are mostly interested in the order of magnitude of the errors. In this case, they are still relevant to our application domains and make it feasible to explore the performance between different platforms. Therefore, we conclude that the WCET estimations show promising results on small hybrid blocks, and none of the tested regression models will be excluded at this point. We believe that additional tuning of the features and models will further improve the accuracy of the predictions.

Table 6.4: Code attribute types from the hybrid blocks used as input features for the DNN prediction models.

Attribute	**Description**
Additive	Number of add/subtract operations. [+, ++, +=, -, −, -=]
Multiplicative	Number of multiplication operations. [*, *=]
Division	Number of division operations. [/, /=]
Modulo	Number of modulo operations. [%, %=]
Logic	Number of logic operations. [&&, ——, !]
Bitwise	Number of bitwise operations. [&, &=, —, —=, ˆ, ˆ=, ˜]
Assign	Number of assignments. [=, +=, ++, -=, −, *=, /=, %=, <<=, >>=, &=, ˆ=, —=]
Shift	Number of shift operations. [>>, <<, >>=, <<=]
Comparison	Number of comparison operations. [==, !=, <, <=, >, >=]
Evaluation	Number of logic evaluation statements. [if, switch, ?]

6.6 Use Case 2: WCET Hybrid Analysis with Deep Neural Networks

6.6.1 Experimental Setup

For the second use case, we use DNN models to make upper bound estimations for the WCET. The first step of implementing the neural networks into the hybrid analysis is the training process. The data in this experiment is acquired from the TACLeBench benchmark suite [60]. A total of 532 hybrid blocks were generated from different TACLeBench applications. All time measurements were performed on an ARM Cortex-M3 CPU on the EZR32 Leopard Gecko board of Silicon Labs [158]. The WCET of each block was obtained with the original hybrid analysis technique in order to acquire labelled data.

The input neurons of neural networks require features that are numeric values. These features are extracted from the code of the generated hybrid blocks by the COBRA framework. Then, the Feature Selector extracts the requested features for each block. In this experiment, we generated an attribute set that is listed in Table 6.4.

Before the acquired features are effective for the DNN models, some pre-processing on the data is required. Most features are counters that resemble the number of occurrences a certain attribute is present. These features are natural numbers in the range of [0, 2.147.483.647] (i.e. 32-bit). However, this range does not perform well for DNN as most activation functions will saturate for large numbers. Therefore, we will normalise our data by rescaling them as floating-point numbers between 0 and 1 with Equation 6.2, where x_n is the scaled value of x_o and X_o is the collection of all features of attribute X.

$$x_n = \frac{x_o - min(X_o)}{max(X_o) - min(X_o)} \tag{6.2}$$

The implementation of the data pre-processing and DNN models are performed with the TensorFlow framework [19]. This framework is an extensive open-source library to create and train ML models in Python. To train our models, we apply the Root-Mean Square Error (RMSE) on the test and validation results [176] as shown in Equation 6.3 with h the prediction function (i.e., hypothesis), X the input set of features, m the size of the input set (i.e., number of datapoints) and y the set of actual values or corresponding labels of each datapoint. This error metric is commonly used to evaluate the accuracy of models during training, as it provides insight in the standard deviation σ of the error distribution.

$$RMSE(\mathbf{X}, h) = \sqrt{\frac{\sum_{i=1}^{m}(h(\mathbf{X}^{(i)}) - y^{(i)})^2}{m}} \tag{6.3}$$

In order to evaluate and compare the performance of our models with other experiments, we calculated the MAPE on the validation sets. This error metric takes an average of the absolute percentage errors without regard of the sign and is commonly used in forecasts [174] (e.g., regression problems) and other research using prediction models [175], as it is very intuitive to interpret the relative error. The MAPE metric is defined as Equation 6.1 that we used in the previous experiment.

6.6.2 Results

For this experiment, we conducted two variations on the dataset with different hybrid block sizes to compare the performance. The size of the blocks is determined by its abstraction level. Each level represents the block's hierarchical position in the block model tree. The first set is created with the smallest blocks without abstraction, which we refer to as Set A. The latter set contains large abstract blocks of the code, and is called Set B. The actual training and validation on these sets is performed through a 5-fold cross-validation process. In order to find the best performing network configuration, we experimentally iterated over different designs to minimise the RMSE error on the validation set. The best network configurations are shown in Table 6.5 for FFNN and Table 6.6 for TRNN. The RMSE scores of our current networks are summarised in Table 6.7. The order of magnitude of the MAPEs equals to 10^4. Therefore, only the RMSE scores are included in the results table for clarity.

6.6.3 Discussion

The results show higher errors than initially expected. We observe MAPEs of an order equal to 10^4 for the different DNN models. Furthermore, the error on the number of clock cycles is equal to a factor $\times 20$ when taking the scaling factor of the internal counter into account. This deviation is too large to obtain a tight upper bound on the WCET and is significantly higher compared to the results of the regression models in the other experiments. The FFNN network for small hybrid blocks, i.e. Set A, has the smallest maximum RMSE score (0.4) on the validation set in the results of Table 6.7, and thus it best fits the validation set of all trained models. We notice that the models converge to a minimum at around 40 training iterations. Further increasing the number of epochs does not improve the results. When evaluating the test sets after each training batch, we observe that the models are converging. Nevertheless, the minimum it converges to is definitely not a global one.

During the experiments, we have noted that our best network for dataset A is not always the best network to solve dataset B with larger abstract blocks. Therefore, we continued the experiments with different networks for each dataset. Nevertheless, the results for the bigger blocks of set B are remarkably worse than set A. Due to the higher abstraction level, the number of blocks and thus the training set is much smaller

Table 6.5: Layers and properties of the FFNN models and the analysis-network for the TRNN. *Output layer

Dataset A	**Layers**					**Properties**	
	Input	**1**	**2**	**3**	**4**		
No. neurons	10	32	32	32	1*	Learning rate Optimiser	0.001 Adam
Activation f.	n/a	σ	Leaky ReLU	=	=	No. epochs / batch size	40 / 20
Regularisation	n/a	L2 (β=0.01)	=	=	=	No. samples (train / test)	348 / 86
Dataset B	**Input**	**1**	**2**	**3**	**4**	**Properties**	
No. neurons	10	32	128	1*	n/a	Learning rate Optimiser	0.001 Adam
Activation f.	n/a	σ	Leaky ReLU	=	n/a	No. epochs / batch size	40 / 10
Regularisation	n/a	L2 (β=0.01)	L2 (β=0.06)	=	n/a	No. samples (train / test)	79 / 19

Table 6.6: Layers and properties of the synthesis-network for the TRNN. *Output layer

Dataset A/B	**Layers**				**Properties**	
	Input	**1**	**2**	**3**		
No. neurons	11+1	128 (Dataset A) 32 (Dataset B)	128	1*	Learning rate Optimiser	0.001 Adam
Activation f.	n/a	Leaky ReLU	=	=	No. epochs / batch size	400 / variable
Regularisation	n/a	L2 (β=0.05)	=	=	No. programs (train / test)	14 / 3

Table 6.7: RMSE scores of the training DNN models.

Model	**Set**	**Average RMSE**		**Min. RMSE**		**Max. RMSE**	
		Training	**Test**	**Training**	**Test**	**Training**	**Test**
FFNN	A	0.23	0.37	0.22	0.34	0.25	0.40
FFNN	B	0.52	0.79	0.51	0.73	0.53	0.85
TRNN	A	0.24	0.25	0.16	0.16	0.47	0.45
TRNN	B	5.20	5.40	3.57	3.85	7.32	7.29

(i.e. 98 blocks) compared to the other set. Additionally, the feature distribution of the datasets was not good enough for the models. Some features were scarcely present in the sets, such as modulo and logic operations. Lastly, the size of the blocks is limited by the benchmark programs used. As a result, the number of blocks with small WCETs make up a larger part of the dataset.

The concept of TRNNs replacing the entire analysis is an interesting research track but requires a lot of additional exploration in finding the best model configuration and parameter modelling before it will become feasible. The synthesis-networks had problems with converging. Verifying the results revealed that the models barely learned something. The worse performance of the TRNNs is probably because of the lack of hierarchical information of the CFG. The synthesis-network receives WCETs values of its child nodes, but it does not know how to combine these results to a final upper bound. Incorporating flow facts, such as loop bounds, recursion conditions, etc.; and block type information, e.g. iteration block, function block, etc.; to the model is definitely a track we need to explore further. However, we will focus only on regression models as WCRC estimators in the remainder of this work.

6.7 Use Case 3: WCEC Hybrid Analysis with Regression Models

Creating a sound energy model is an NP-hard problem due to the ‘Circuit Switching Problem’ [70], we want to study the capability of ML to make an abstraction of the hardware and learn its behaviour. In this section, we will discuss the experimental setup we created on which we will train WCEC prediction models and present the results of this experiment.

6.7.1 Experimental Setup

For this experiment, we use a small IoT setup that consists of embedded controllers in a Bluetooth mesh network. The clients are batteryless nodes equipped with sensors, e.g., temperature, to monitor the environment. Each remote sensor node has a power management board with a supercapacitor to store the energy and an energy harvesting device, such as solar panels. Each client has to schedule a set of tasks with a limited energy budget. These tasks vary from reading sensor values, performing small calculations with the data, storing intermediate data to the internal flash memory, and transmitting the data to a server node in the mesh network. A schematic overview of the system is illustrated in Figure 6.8. The target embedded controller used in our setup is the Nordic nRF52 development kit board with an nRF52832 SoC that is built on top of an ARM Cortex-M4 CPU [177]. The onboard energy-aware software scheduler requires a schedule with the requested task’s periodicity, and a list of the WCET and WCEC predictions of each schedulable task. These predictions are determined before final deployment on the hardware and provided in a configuration file to the scheduler. The offline profiling of each software task is performed by an automated testbench, as described in Section 5.3 that we created for the COBRA framework.

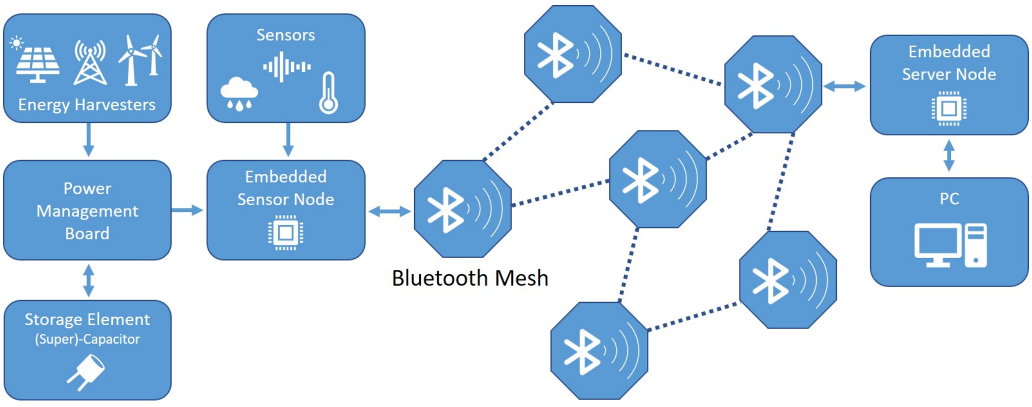


Figure 6.8: Diagram of the experimental IoT setup for WCEC measurements.

Table 6.8: List of regression models selected as WCEC estimators for the experiment.

Linear Regression	Polynomial Regression (3rd Degree)
Decision Tree Regression	Random Forest Regression
Support Vector Reg. (Linear Kernel)	Support Vector Reg. (RBF Kernel)
K-Nearest Neighbours Regression	Ridge Regression

6.7.2 Results

For this experiment, we selected the eight different regression models of Table 6.8. Each of these models are trained with a dataset generated from benchmark code of the TACLeBench [60]. A total of 120 small hybrid blocks are generated with the Block Generator from the benchmarks in Table 6.9. Each of these benchmarks are split with no abstraction enabled. This results in the generation of default basic blocks. Such basic blocks will contain a single trace of instruction that has exactly one entry and one exit point in its code flow. All generated blocks are then used as datapoints in a 4-fold cross-validation process. Specifically, the datapoints are randomly split to create four sets with 90 elements for training and the remaining 30 blocks as validation.

Each datapoint contains the feature list of all software attributes from Table 6.1 and the corresponding labelled upper bound measurement from the automated testbench. The aim of this experiment is to evaluate the ability of regression models to approximate the measured upper bound from the testbench, as we want to assess the feasibility of replacing the physical measurements with prediction models. Therefore, we did not evaluate the measured upper bound or the pessimism introduced by the testbench.

In order to evaluate the performance of our models, we calculated the MAPE on the validation sets. This error metric takes an average of the absolute percentage errors without regard of the sign and is commonly used in forecasts [174] (e.g., regression problems) and other research using prediction models [175], as it is very intuitive to interpret the relative error. The MAPE metric is defined as Equation 6.1 that we used in the previous experiments.

During the initial training run, we noticed poor performance of several regression

Table 6.9: List of benchmarks from TACLeBench used for dataset generation for training and validation.

Testbench	Number of Blocks	Lines of Code
Binary Search	12	67
Bitonic	13	72
BSort	15	60
Complex Updates	6	62
Filter Bank	10	89
Insert Sort	15	69
LMS	19	96
Recursion	8	36
ST	22	127

models, among which are the Forest, Random Forest and SVRs. After close inspection, we notice that the prediction values of these models converge to one and the same value. Our hypothesis is that an issue occurs with the training of the loss function of the models, due to the extremely small order of magnitude of the provided prediction labels. The energy measurements are expressed in the International System of Units (SI)-unit of energy, the Joule [J]. These values are very small for low-power devices. The dataset we captured from the Nordic board contains on average measurements with an order of magnitude of 10^{-6} J. To solve this issue, we scaled the labels by shifting the order with the power of 10^{10} to have enough significant digits before the decimal point. After the scaling, we noticed a significant improvement of the worse performing regression models. In order to further improve the performance of the ML models, we need to optimise the training process, the data pre-processing, and the model by tweaking the 'settings' or hyperparameters of the entire processing pipeline. The field of creating these optimised pipelines based on the training data in an automatic manner is called Automated Machine Learning (AutoML) [178]. One of the first AutoML methods was the Tree-Based Pipeline Optimization Tool (TPOT) by the Computational Genetics Laboratory of the University of Pennsylvania [179]. This software package allows data scientist to automate the process of creating an ML pipeline, i.e., pre-processing of data, selection of regression models and hyperparameter tuning, that is optimised for a given dataset. For this experiment, we did not implement any AutoML methods yet, but made a first comparison between regression models to study the potential of our methodology while keeping the initial complexity low.

The training of these models requires a decent amount of computation time to complete as this depends on the size of the training dataset, the model complexity and the number of epochs. The actual evaluation of a datapoint with a trained model only takes a fraction of a second. During our experiments, the training and validation for all models took around 10 min to complete on a laptop with an Intel i7-6700HQ CPU processor. While the training and evaluation require considerable computational power, its resource requirements are less important compared to the prediction performance as the profiling process is carried out offline on the developer's computer, and not at runtime. The results of these trained models are listed in Table 6.10. For each model, we show the average and worst-case errors on the validation sets. The last row indicates the relative computation

Table 6.10: 4-Fold cross-validated MAPE on the validation set for each trained WCEC regression model with their relative computation time.

Regression Models	**MAPE**			**Relative Compute Time**	
	Best	**Average**	**Worst**	**Train**	**Predict**
Linear	116%	418.9%	1021.7%	1.73	**1**
Polynomial (3rd Degree)	$8.8{\times}10^5$%	$2.3{\times}10^9$%	$7.7{\times}10^9$%	20.6	10.3
Decision Tree	6.5%	206.4%	551.1%	1.35	1.46
Random Forest	62.7%	224%	608.1%	25.5	2.73
Support Vector (Linear)	73.4%	190.4%	438.1%	6663	2.48
Support Vector (RBF)	71.3%	440.8%	876.3%	2454	9.27
K-Nearest Neighbours	42.3%	262.1%	769.9%	**1**	2.46
Ridge	116%	418.9%	1021.7%	3.21	2.83

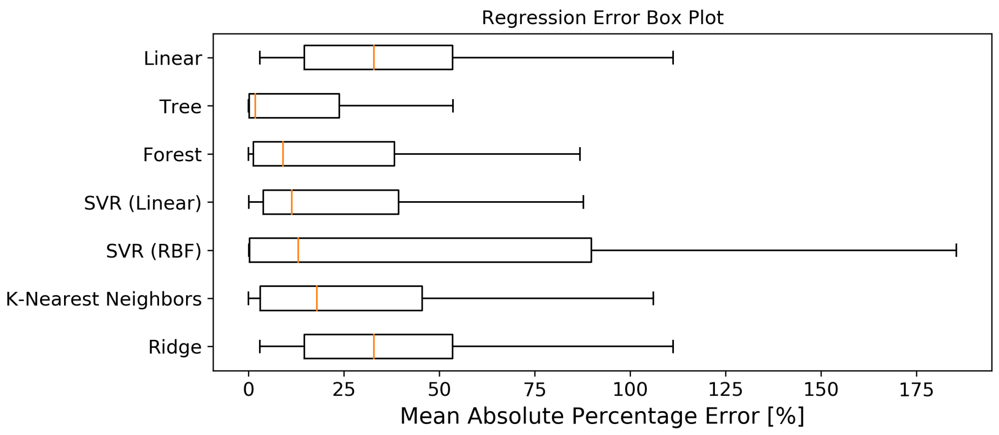


Figure 6.9: MAPE box plots of the WCEC regression models with removed outliers.

time of each model for training and predicting. A computation time of ‘1’ indicates the shortest average execution time of the indicated action between all listed models. The training time for one cross-validation iteration of the K-Nearest Neighbours Regression was the shortest of all models with an average of 1.3 ms. The Linear Regression had the fastest average prediction time of around 1 ms.

The best performing models in this experiment are on first sight: Decision Tree, Random Forest and SVR with Linear Kernel. High deviations are noticeable between the best and worst MAPEs of the cross-validated sets. In Figure 6.9, we plotted a normalised box plot of the error distribution. The polynomial regression was omitted to improve the clarity of the figure.

The graphs in Figure 6.11 plot the predicted upper bounds for the energy consumption with respect to the actual measured results. The vertical distance between each point and the dotted line in the graph indicates the absolute prediction error for a given datapoint.

MAPE is an excellent metric to gain insight in the performance of the prediction

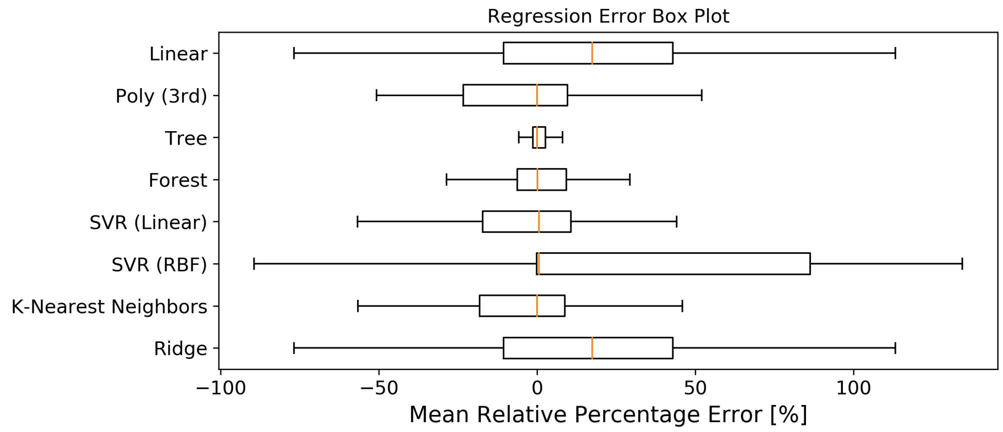


Figure 6.10: MRPE box plots of the WCEC regression models with removed outliers.

models. However, there is no indication if the models are mostly over- or underestimating the WCEC. As previously discussed, it is important to keep the upper bound above the real worst-case to avoid compromising the functional behaviour of the system. Therefore, we calculated the MRPE for each model in Figure 6.10. MRPE calculates the percentage difference between the predicted and actual values but keeps the sign of the error.

6.7.3 Discussion

With the automated testbench for the COBRA framework, we were able to generate hybrid blocks from benchmark code and measure the energy consumption on the physical device for upper bound WCEC analysis. A full profiling cycle by the COBRA framework only takes a few minutes depending on the complexity of the program, which most of the time is spent during the actual measurement phase on the target hardware. Although this process already automates lots of tedious and time-consuming tasks, it still required manual effort to validate the system operation, such as input set generation and coverage, feature generation, testbench generation and measurement. As a result, the dataset for this experiment is rather small. Nevertheless, we were able to train a set of regression models based on software-related features.

The results of Table 6.10 show an average percentage error of around 200% for the best performing regression models. Nevertheless, these results are already impressive given that they were obtained from the default configuration of the regression models without tuning any hyperparameters in the process, with the exception of the Polynomial and SVR with an RBF kernel. Moreover, the order of magnitude of the errors is still relevant for our application domains. The best models from the list are Decision Tree, Random Forest and SVR with a Linear Kernel.

Decision Tree and Random Forest regression are powerful prediction models that make accurate predictions based on the trained data with nonlinear relationships [169]. A decision tree splits the trained datapoints into clusters and assigns a value to them during training, which provides excellent performance on previously seen data. However, the output of the model is discreet. A high partitioning of the tree model is therefore needed

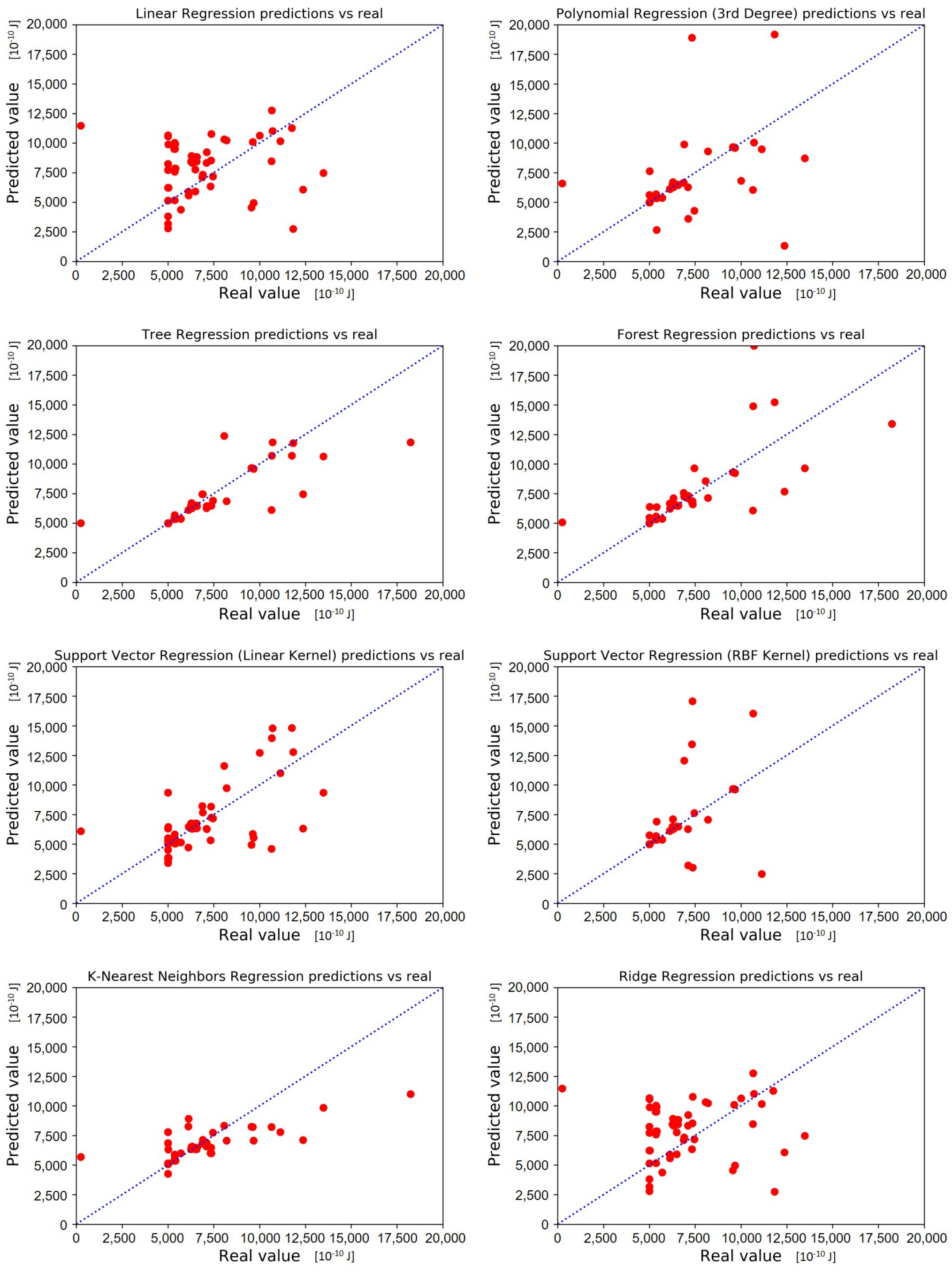


Figure 6.11: Predicted vs. real WCEC upper bounds for trained regression models on the validation sets.

to achieve accurate predictions, but this increases the model's complexity and size tremendously. The model also does not perform well with unseen data during training. The Random Forest, on the other hand, uses an ensemble of randomly generated decision trees whose output is an averaged aggregation from the results of the different decision trees. Therefore, each potentially predictive feature is able to play a role in the decision making instead of being suppressed by others. It is also better in providing predictions of previous unseen datapoints. Nevertheless, both methodologies are very prone to overfitting the training set.

The SVR model has on average the smallest MAPE of all models in the experiment. However, these models are very sensitive for differences in the range between input attributes [180]. Providing an upper bound on each attribute, such as number of additions, is not very feasible to determine for different sizes of hybrid blocks. The impact of the scale differences between features was minimal because of the relatively small sized hybrid blocks in this experiment. We could partially tackle this issue by training different models for different split strategies for hybrid blocks and thus, cluster different sizes of blocks.

The worst performing model on the list is the Polynomial regression. On the validation set, the model had extremely high prediction errors compared to the other models. To improve the performance, we applied a grid search to find a better performing lower order degree as polynomial kernel. An optimum was found for a 3rd degree kernel. However, the predictions are still bad. After close analysis, we noticed that the model overfitted the training data excessively with an average MAPE of only 0.75% on the training sets themselves. This overfitting issue is possibly caused by the small size of the training set compared to the number and complexity of attributes used in order for the polynomial model to generalise the problem. The poor performing Polynomial regression can be replaced with Symbolic Regression tools, such as PySR [181] or TuringBot [182], that try to find the best fitting function for a given dataset while preferring the shortest expression [183]. This approach will generate 'simple' models with better generalisation capabilities, minimising the chances of overfitting the training data due to complex functions.

When we examine the scatter plots of the validation sets in Figure 6.11, we notice smaller errors in the range of $5{,}000\times10^{-10}$ J and $7{,}500\times10e^{-10}$ J where many datapoints are clustered. Datapoints for the higher range are more scarce in the training set, resulting in worse performance. Furthermore, these outliers have an impact on the average error of the models. In order to better compare the performance of the regression models, we have normalised the error distribution and plotted it in a box plot as shown in Figure 6.9. If we compare the MAPEs of the regression models, we see that the Decision Tree model performs the best with the third quartile (Q_3), or 75% of the predictions without outliers, below an absolute error of 23.8% and a median of 1.8%. This model is followed by Random Forest (Q_3: 38.2%, Med: 9%) and SVR with Linear Kernel (Q_3: 39.3%, Med: 11.4%).

An upper bound prediction should try to approximate the real WCEC as close as possible to minimise pessimism. Nonetheless, it should never underestimate the actual WCEC as this could result in detrimental behaviour of the system, e.g. shutdown of the device because of insufficient available energy in the capacitor. In Figure 6.10, we have calculated the relative error to look at the distribution of under- and overestimated predictions. We note that all regression models are prone to underestimation with their median close to the boundary. Nevertheless, we are able to create an error model based on the validation errors to introduce pessimism to the predictions. This will add a margin to acquire safe upper bound estimates.

The acquired predictions from the ML WCRC Estimator provide an upper bound of the energy consumption for each given hybrid block. The last step is to combine the predictions of the individual hybrid blocks according to the CFG of the task to obtain an upper bound of the total task. This is achieved with the Analysis Wizard with the use of IPET as discussed in Section 5.4.

The final process to acquire an upper bound WCEC estimation for the batteryless IoT sensor device we use in the experimental setup (Section 6.7.1) is as follows. Firstly, we split the software task in smaller hybrid blocks using the COBRA framework. Therefore, we create a project configuration that describes the block splitting strategy and describes the target hardware properties. The tool will generate a list of software features for each hybrid block. Secondly, the user selects an estimator model to perform the prediction for the given platform by providing the software features set. Thirdly, the upper bound predictions are then statically combined and exported for each task by the framework to a formatted text file. Finally, the developer uses the upper bound WCEC estimates from the output file to configure the energy-aware scheduler. The scheduler that is being developed in our experimental setup uses a JavaScript Object Notation (JSON)-formatted file that is read during the boot sequence of the IoT device. If code changes are made to a task, the toolchain needs to profile that task again. The complete toolchain is executed offline when a trained WCRC estimator is used. Therefore, the code profiling process can be executed in the background when the code is developed.

6.8 Conclusion

Determining the actual WCRC of a task is not a trivial problem to solve. In this chapter, we extended our hybrid methodology with ML-based predictions to predict an upper bound of the WCRC for small sections of the code, called the hybrid blocks. These predictions are then statically combined to get a final upper bound estimation of the WCRC. We discussed the importance of early-stage WCRC predictions where current analysis methods fall short.

A set of software-related features where selected and extracted by the COBRA-HPA framework. The automated testbench facilitates the process to generate labelled training and validations sets to train different predictions models. In three use cases, we trained and compared the performance of regression and DLNN models to predict the WCET and WCEC. The results show promising performance that are relevant within our application domains, but these models require additional tuning of the input features (e.g. feature engineering) and hyperparameters of the prediction models, for example by employing AutoML techniques. With this new ML-based extension of the hybrid approach, we have created a solution for **RQ 2** (Section 1.3).

Chapter 7

Worst-Case Resource Consumption of Neural Networks

In the previous chapters, we have discussed a hybrid methodology using ML techniques to estimate the WCRC of the traditional programmed tasks. However, recent advances in AI technology have made it possible to create and deploy accurate optimised ML models on hard-constrained embedded devices. In this chapter, we take a look at neural networks used as a mean of control logic. Using the architectural design of neural networks, we extend our hybrid methodology to perform WCRC estimation based on their abstract representation of neurons to achieve early insights without training or deploying the model on the target device.

7.1 Inference on Embedded Devices

Recent technical advancements in computing architectures, GPUs, ML accelerators, have resulted in a significant growth of new and better ML techniques. Training and integration of ML models have become more accessible to research and industry. In the last decade, we have seen the emergence of AI-powered applications, such as autonomous driving vehicles on the road [184], smart virtual assistants [22], anomaly detection [185], diagnosis of medical conditions [186], ubiquitous computing [187], and many more. This approach allows us to create an abstraction of the problem by learning patterns in the data in contrast to fully modelling the system with a rule-based approach. Integrating high-performance ML models instead of conventional programmed logic can have a positive impact on energy consumption, privacy and responsiveness of embedded systems [23], [188]. Nevertheless, applications containing hard-constrained systems, such as CPS or batteryless IoT devices, require insight into their WCRC behaviour to guarantee predictability, safety or efficiency during operation, i.e. inference of the model on the target system.

The usage of ML creates an abstraction of the 'decision logic', which is often referred to as a *black box*. The model makes decisions based on previously seen data during training. The developer has no direct insight on what the model's output for a given input would be, or why. Groundbreaking research in *explainable AI* is being conducted to understand the 'reasoning' behind the AI [189], [190], and so turning these *black box* models into *glass box* models. In contrast, extensive research into the resource behaviour (e.g. latency, energy consumption, etc.) of ML models is particularly scarce. Nonetheless, determining the WCRC of systems with hard resource constraints is crucial to guarantee safe and correct operation. The inference of an ML model on a generic computing device is essentially a compiled binary of instructions from the processor's ISA. This transformation from the abstract model with weights and biases to an executable binary is performed by the underlying ML library used. Therefore, the behaviour of the system depends on this model conversion to code, and thus the implementation of the library used. A vast collection of open-source libraries is publicly available for AI development, such as TensorFlow [19], PyTorch [20] and Scikit-Learn [173].

The most commonly used ML libraries for training and running these models are implemented in higher-level programming languages, such as Python, Java or JavaScript, and optimised to run on high-end computing infrastructures with the ability to use the power of one or many GPUs. While these libraries are very powerful, the support for target platforms is limited and the resource requirements are significantly high. For instance, a typical compiled TensorFlow library has a size of over 20 Mbytes [23]. Nevertheless, frameworks such as TensorFlow and PyTorch have lightweight 'mobile' implementations of their engines to deploy trained models on more constrained edge devices (i.e. smartphones). These *light* versions are made compact by removing infrequently used operators, providing support for quantization of networks to reduce the size of the weights and biases (i.e. conversion of 32-bit float types to 8-bit integer types), and containing optimised libraries for a wide range of mobile CPUs [23].

With the rise of ubiquitous computing in IoT, the interest of deploying ML models on edge devices has become more prominent. However, these edge devices vary from mobile phones to small sensor nodes with very limited resources (e.g. computing power and energy). These resource-constrained sensors are low-power systems with limited memory running bare-metal applications without an OS. Deploying ML on these devices entails new challenges of creating models that fit within the memory and storage of an MCU (i.e. with only Kbytes of memory) and support a wide range of different platforms. *TinyML* is a new research field that involves ML technology and applications that operate on ultra-low-power devices ($\leq 1\,\text{mW}$) [21], [23]. The previously discussed 'mobile' frameworks already contain significant optimisations for memory size and performance. Nonetheless, these 'mobile' frameworks are still not fully suited for TinyML applications, as they still require an OS and dependencies for deployment. In addition, these frameworks depend on floating-point operations that are not available on all embedded platforms and have a compiled library size that is still too large for devices with limited storage. Furthermore, it is impossible for these frameworks to support optimised implementations for the wide variety of embedded platforms on the market [23]. Therefore, specialised frameworks with broad support of embedded platforms and small memory footprints are required to create TinyML systems. Examples of such TinyML libraries are AIfES [191], MCUNet [192], TinyOL [193], Train++ [194] and TensorFlow Lite Micro [23], [195]. The last framework in the list extends upon a TensorFlow Lite model and converts the operators and weights into an optimised memory map with FlatBuffers that are loaded during the first inference [23]. Additionally, its modular build system allows to include specialised implementations of each operator to take advantage of the available hardware to further improve the inference performance. In this chapter, we will focus on neural networks, and IoT devices that are constrained by limited energy budgets.

7.2 Predicting WCRC

The resource consumption of ML has acquired attention in both research and industry as AI is being applied on a growing scale of applications and devices. Based on the target requirements and available resources, we need to optimise our models to fit the intended application. In the domain of ML, there are three prominent resources that are aimed for during optimisation; namely, *latency* (i.e. execution time), *energy consumption*, and *model and binary size*. In our research, we will solely focus on the **latency** and **energy consumption** as resources for which we want to estimate the model's behaviour.

In order to determine the WCRC behaviour, we need to be able to measure or calculate the resource consumption during inference. If we examine a neural network, as shown in Figure 6.2, we notice that it consists out of a matrix of consecutive multiplication and addition operations. A rudimentary approach to determine the latency of a model is to count the number of Floating-Point Operations (FLOPs) (i.e. number of operations in all neurons) that are used during an inference, multiplied by the period of the CPU clock [23]. Using the number of FLOPs, we can also estimate the energy consumption by taking the power consumption of the CPU and multiplying it with the model's latency. These metrics provide an easy benchmark to compare the performance between models but are still incomplete and naive approximations of the model and underlying hardware.

Measurement-based techniques are used in the state-of-practice to obtain absolute results on the resource consumption. These techniques mostly consist of using basic logging, internal timers (i.e. tic-toc profiling), external analysers and probing, and (shotgun) profiling [23]. In the SOTA, little research is dedicated to the resource behaviour that specifically targets AI models. One could argue that all WCRC research is also applicable to AI as each model is essentially an abstraction of the software code that is running underneath. As a result, we need to apply our WCRC analysis methodologies on the converted models. However, we believe that the data-driven approach of deep learning simplifies the control flow, and thus makes it feasible to use the abstract model description (e.g. number of neurons, hidden layers, etc.) as input features to train WCRC estimation models for neural networks. In the work of Reif et al. [24], they trained a linear and random forest regressors to determine the execution time and energy consumption of ML workloads on a hardware accelerator for neural networks (i.e. Google Edge TPU). Using a custom measurement setup, they created a dataset with a limited set of input features (i.e. number of layers, number of network parameters, memory usage, and number of multiplication-accumulation operations). The estimation models showed promising results with small estimation errors. These optimistic results are due to the improved deterministic behaviour of the TPU architecture [196], which is almost linear for inference latency compared to the number of operations. Nonetheless, integrating these TPUs for all embedded applications involving ML is not feasible as the cost and energy consumption is too high for low-cost IoT sensors, and mainly has only support for TensorFlow. In our research, we want to focus on embedded devices with CPU-based processing, as they are used prominently. Nevertheless, the complexity of estimating the resource consumption for these systems will be higher due to influences of the hardware/OS and software.

Resource-aware AI is getting more important with the evolution of edge computing. These embedded devices have strict resource constraints that impact the design process of a system. In order to validate and measure the 'performance' between different platforms, we use benchmarks to relatively compare them, as we previously discussed in Section 2.3.

However, the workload of traditional code benchmarks focuses on properties, such as control flow, that are less relevant for ML-based logic. Therefore, new benchmark suites are being composed with relevant ML workloads for different application domains [188], [197], [198].

As we have established in the previous chapters, the process of profiling the resource behaviour of an application is a long and tedious task to perform. An upper bound estimation of the WCRC is an important KPI used in resource-aware AI research, as explained in Appendix B. For example, a Neural Architecture Search (NAS) algorithm uses the WCRC metric as a KPI to iterate over different network configurations to find an optimal trade-off between resource consumption and performance [199], [200]. As a result, an enormous amount of architectures need to be evaluated and compared to find an optimum. The time it takes to perform the WCRC analysis (i.e. training, compiling, flashing and profiling) has a detrimental impact on the feasibility of the NAS methodology, as a high latency on the upper bound results renders this approach intractable to perform.

The **challenge** we want to tackle is providing early-stage estimations of the WCRC behaviour of neural networks without having to train, or physically profile them on the target embedded system. Therefore, we propose to utilise our ML-based hybrid WCRC methodology and extend it to support the architectural description of neural networks as input. Such a prediction model for WCRC estimation will allow us to iterate over different architectures without losing time and effort in training ML models that would potentially not satisfy the constraints of the system under development.

7.3 Applying the Hybrid ML-based Methodology on Neural Networks

In this final chapter, we want to apply our ML-based WCRC analysis methodology to provide early-stage estimation on the resource behaviour of neural networks. This will allow ML experts to be aware of the resources while iterating over several model architectures without having to train and fully profile each one. The usage of ML-based analysis methodology has been proposed to estimate the latency and energy consumption of convolutional and fully connected neural networks on TPU hardware [24]. In [24], their estimation models for the TPU architecture already provided promising results. In our work, we further expand on this concept and estimate the resource consumption on CPU-based MCUs. Therefore, we updated the schematic of Figure 6.1 in Chapter 6 as shown in Figure 7.1.

The methodology for WCRC analysis on neural networks we propose is analogous to the approach we took on traditional programming (e.g. handwritten code) in the previous chapters. One of the big differences is the way how the logic is split into blocks. In the original schematic (Figure 6.1), we used the hybrid block generator to create hybrid blocks from the software tasks before the measurement-based analysis or prediction model is applied. Nonetheless, we want to keep this flexibility in our toolchain for neural networks. This allows us to split large networks into smaller manageable blocks for the estimation model; for instance, fixed (one-hot) input vector features that limit the size of the network (e.g. weights and biases, type of activation function per neuron, etc.). Different strategies to split DNNs are currently available in the SOTA [201]–[203], some of which even have a positive impact on memory usage and energy consumption with a limited accuracy hit for convolutional networks [202], [203]. However, we will keep the

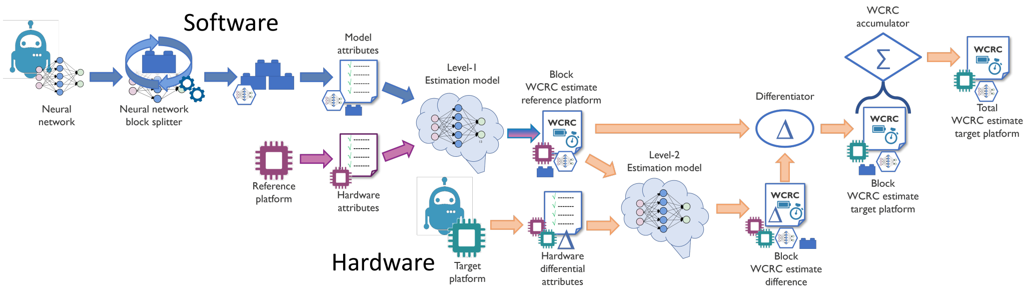

Software
Neural network
Neural network block splitter
Model attributes
Level-I Estimation model
Block WCRC estimate reference platform
WCRC
Reference platform
Hardware attributes
Differentiator
Δ
WCRC accumulator
Σ
Total WCRC estimate target platform
Block WCRC estimate target platform
Level-2 Estimation model
Hardware
Target platform
Hardware differential attributes
Block WCRC estimate difference

Figure 7.1: Schematic overview of the ML-based hybrid analysis methodology for neural networks.

splitting factor equal to 0, and thus not split the neural networks into blocks as it is out of scope for this thesis. The other difference in this approach is the list of prediction attributes used for neural networks by the estimation model, which we briefly discuss in the next section.

7.3.1 Model Attributes

A neural network consists of a matrix of interconnected neurons, as previously discussed in Section 7.1. When defining a new DNN architecture, you need to specify a number of properties, such as the size of the input and output vector, number of hidden layers, activation function and number of neurons at each hidden layer. These model descriptions are used by the ML framework to instantiate a data model with weights and biases that need to be trained. The final model consists of the model description with the trained weights and biases. Using this model, we are able to deploy and run inference on the target platform. Each framework has its own format to represent an ML model. However, standardisation in the ML community has resulted in cross-platform formats. For instance, Open Neural Network Exchange (ONNX) is an open binary format for ML model representation that is supported by a wide variety of tools and runtimes [204]. Based on this abstract data model, we are able to extract a feature set that describes the model, and eventually the code that will be executed on the hardware.

The number of features extracted from the ML model description is much more limited compared to code-related features used in the previous chapters. Our goal is to select those features that allow the estimation model to predict the WCRC of the neural network. In the work of Reif et al. [24], they selected a feature set that represents the number of multiplication and accumulation operations, based on the number of neurons in the network and the filter sizes of convolutional networks. In our work, we extended the COBRA framework with a new tool that extracts these features based on the model description from TensorFlow models. While we have access to both the architecture description, and the weights and biases, we will only focus on features that originate from the architecture of the network in our experiments and not on the actual weights and biases. As a result, we have compiled a feature set for the experiments, as shown in Table 7.1.

Table 7.1: Model attribute types from the neural network architecture used as input features for the prediction models.

Attribute	**Description**
Number of inputs	The size of the input vector of the network.
Number of outputs	The size of the output vector of the network.
Number of layers	The number of hidden layers in the network.
Number of weights	The number of links in the network.
Number of biases	The number of neurons in the network.

7.3.2 Model Generation

Training a prediction model requires an extensive dataset from which we want to generalise this regression problem, i.e. estimating the WCRC. Therefore, we need to generate a labelled training set with the input features from Table 7.1 and the corresponding annotated WCRC labels. In order to facilitate us in obtaining this dataset, we created a new tool that generates DNN model configurations based on a parameterised model description. The *Neural Network Generator* uses extendable generator classes to create these models. In our experiments, we use the *dense neural network* class for our dataset creation. The generator's output is fully customisable by defining a random seed, and ranges for the parameters that need to be randomised. The outcome is a list of randomly initialised TensorFlow models. These models can then be optionally trained in the TensorFlow runtime environment before running them through the COBRA toolchain for analysis.

The output models of the Neural Network Generator are provided to the Profiling Assistant with the automated testbench (Figure A.2), as we used in our previous experiments. Finally, the measurements from the Profiling Assistant are used as labels together with their corresponding features from the model to create a dataset for training and validation of an estimation model with COBRA using the approach of Section A.5.

7.4 Use Case 1: WCET Analysis of Neural Networks

After establishing the hybrid estimation methodology for WCET and WCEC on traditional programmed, we will now apply our strategy on tasks that are driven by neural networks. In this section, we will discuss the adapted experimental setup to train a prediction model for WCET and its performance.

7.4.1 Experimental Setup

For the setup of these experiments, we reuse the small IoT setup (Figure 6.8) of the use case in Section 6.7. However, the tasks that needs to be profiled are replaced by DNNs. The target embedded controller used in the setup is the Nordic nRF52 development kit board with an nRF52832 SoC that is built with an ARM Cortex-M4 CPU [177].

The models for our training dataset are acquired from the Neural Network Generator. In order to profile these models, we need input vectors to measure the resource behaviour during inference. Therefore, we utilise the commonly used Modified National Institute of Standards and Technology (MNIST) database as input for our generated ML models.

Table 7.2: List of regression models selected as WCRC estimators for the experiment.

Linear Regression	Polynomial Regression (2nd Degree)
Decision Tree Regression	Random Forest Regression
Support Vector Reg. (Linear Kernel)	Support Vector Reg. (RBF Kernel)
K-Nearest Neighbours Regression	Ridge Regression

The MNIST database is a large dataset with 70,000 handwritten digits for training and testing ML techniques [205]. Each image is 28 by 28 pixels large and contains a grey-scale image of a handwritten single-digit number with a corresponding label. As a result, the models we generate during this experiment will have a fixed size input and output vector. The input vector will contain the individual pixel data of the image, and therefore has a size of $28 \times 28 = 784$ neurons in the input layer. The output vector is the classification of the digits (0-9) and contains the probability for each digit (i.e. ten neurons in the output layer).

Each generated model is briefly trained to avoid saturation of neurons in the network, as all weights and biases are randomly initialised at first. The accuracy of the MNIST predictions are not considered in the experiments, as we are currently only interested in the resource consumption of the models regardless of their performance. For the inference of the TensorFlow models we use our own custom runtime environment. This custom framework is utilised in our research group to study the impact of the operators on the behaviour of the hardware, provides full control of the operations, and runs bare metal on the target controller without OS. Each model is compiled and deployed on the DUT with the testing harness of our automated testbench. Each model is fed by a randomly selected set of input vectors from the MNIST database for which the execution time and energy consumption are measured. These results are accumulated and processed by the Profiling Assistant. In the next section, we discuss the acquired measurements for WCET and the results of the trained estimation models.

7.4.2 Results

For this experiment, we selected eight different regression models as listed in Table 7.2. These models are implemented and trained within the Scikit-Learn [173] framework. A total of 1014 DNN models for the dataset are generated by the Neural Network Generator with the MNIST dataset as input. Each model was executed ten times on the DUT with different input vectors. A full cycle of generating, compiling and profiling a model takes around one minute for a single model. All the generated models are then used as datapoints in a 4-fold cross-validation process.

Each datapoint contains the feature list of the model attributes from Table 7.1 together with the corresponding labelled upper bound measurements from the testbench. In order to evaluate the performance of our models, we calculated the MAPE on the validation sets. This error metric takes an average of the absolute percentage errors without regard of the sign and is commonly used in forecasts [174] (e.g. regression problems) and other research using prediction models [175], as it is very intuitive to interpret the relative error. The MAPE metric is defined as Equation 7.1 with h the prediction function (i.e.,

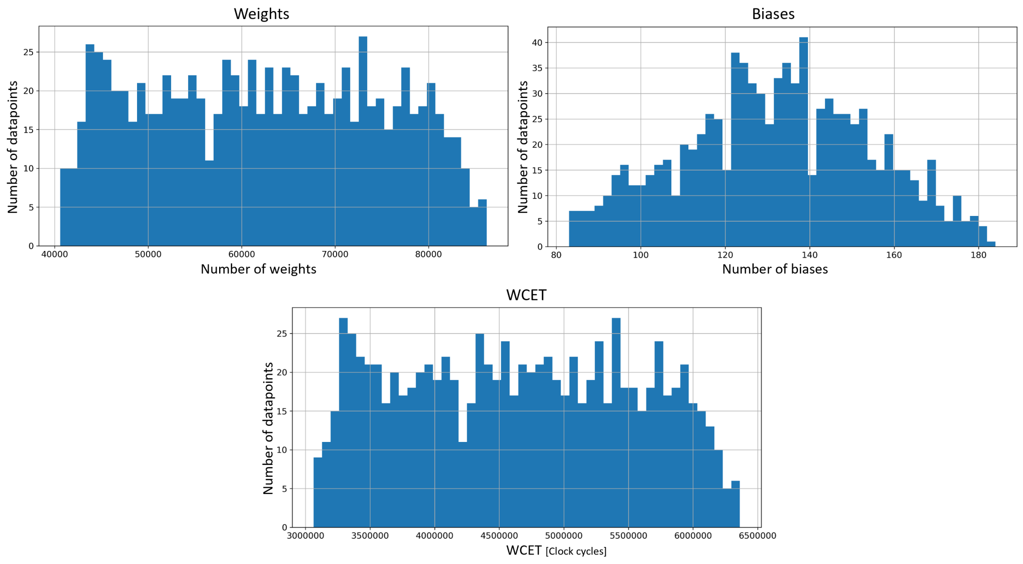


Figure 7.2: Histogram of the attributes (i.e. number of weights and biases) and label (i.e. WCET) in the full dataset of the first experiment.

hypothesis), X the input set of features, m the size of the input set (i.e., number of datapoints) and y the set of actual values or corresponding labels of each datapoint.

$$MAPE(\mathbf{X}, h) = \frac{100}{m} \sum_{i=1}^{m} \left| \frac{y^{(i)} - h(\mathbf{X}^{(i)})}{y^{(i)}} \right| \quad [\%] \tag{7.1}$$

For this use case, we have selected two experiments to gradually increase the complexity of the predictions. In the first experiment, we fixed the total number of layers to four (i.e. one input, two hidden and one output layer). The second experiment loosens the constraint on the number of hidden layers.

Experiment 1: WCET with Fixed Number of Layers

In this experiment, we fixed the number of hidden layers to two. Each model in the training dataset has a variation of the number of neurons (i.e. biases), and thus the number of links (i.e. weights) in the dense model. These variations are randomly determined by a predetermined seed to create reproducible model configurations. The boundaries for the number of neurons in the hidden layers are set to $[50; 100]$, and $[20; 75]$ respectively. The final distributions of the input features in the dataset are presented in the histograms of Figure 7.2. The WCET is measured by an internal register that counts the number of clock cycles that have passed during a single inference. The range of the WCET labels in the dataset is equal to $[3.06 \times 10^6; 6.36 \times 10^6]$ clock cycles.

The training of the models requires a decent amount of computation time to complete as this depends on the size of the training dataset, the model complexity and the number of training epochs that are used for the optimisation of the model. The actual inference time of a datapoint only takes a fraction of a second. During our experiments, the

Table 7.3: 4-Fold cross-validated MAPE on the validation set of the first WCET experiment for each trained regression model with their relative computation time.

Regression Models	**MAPE**			**Relative Compute Time**	
	Best	**Average**	**Worst**	**Train**	**Predict**
Linear	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$	**1**	5.52
Polynomial (2nd Degree)	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$	7.22	**1**
Decision Tree	0.08%	0.09%	0.11%	3.29	7.25
Random Forest	0.07%	0.08%	0.09%	22.72	21.15
Support Vector (Linear)	0.006%	0.007%	0.008%	8785.7	21.4
Support Vector (RBF)	1.41%	1.95%	2.58%	6083.1	40.3
K-Nearest Neighbours	0.07%	0.08%	0.09%	2.33	15.83
Ridge	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$	2.15	11.72

training and validation for all models took around 15 min to complete on a laptop with an Intel i7-6700HQ CPU processor. While the training of the prediction models requires considerable computational power, its resource requirements are less important compared to the prediction performance as the profiling process is carried out offline on a developer's computer before deployment. The results of the trained models are listed in Table 7.3. The table shows the best, average and worst-case errors on the validation sets for each model. The last row indicates the relative computation time of each model for training and predicting. A computation time of '1' indicates the shortest average execution time of the indicated action between all listed models. The training time for one cross-validation iteration of the Linear regression was the shortest of all models with an average of 1.9 ms. The model with the shortest inference time was the Polynomial regression with an average of 0.2 ms.

The best performing models in this experiment are: Linear, Polynomial and Ridge regression. The MAPEs of these models are almost equal to zero, and therefore are annotated with the approximately sign to improve the readability of the table. In Figure 7.3, we plotted normalised box plots of the error distributions. The box plot of the SVR with RBF kernel was omitted to improve the clarity of the figure. The graphs in Figure 7.4 plot the predicted upper bounds of the execution time with respect to the actual measured results for the validation sets of all four performed cross-validations. The vertical distance between each point and the dotted line in the graph indicates the absolute prediction error for a given datapoint of the validation set.

Experiment 2: WCET with Variable Number of Layers

This next experiment builds upon the previous one but adds an additional variation to the parameter list of the DNN models, namely the number of hidden layers. The layer depth is able to vary between two and four layers (including the output layer). The number of neurons that could randomly been assigned to this new third hidden layer are within the boundary [10; 50]. Therefore, a new dataset has been generated with the Neural Network Generator to contain DNN models with varying layer depth. All other configurations, the experimental setup, and used prediction models are identical to the previous experiment.

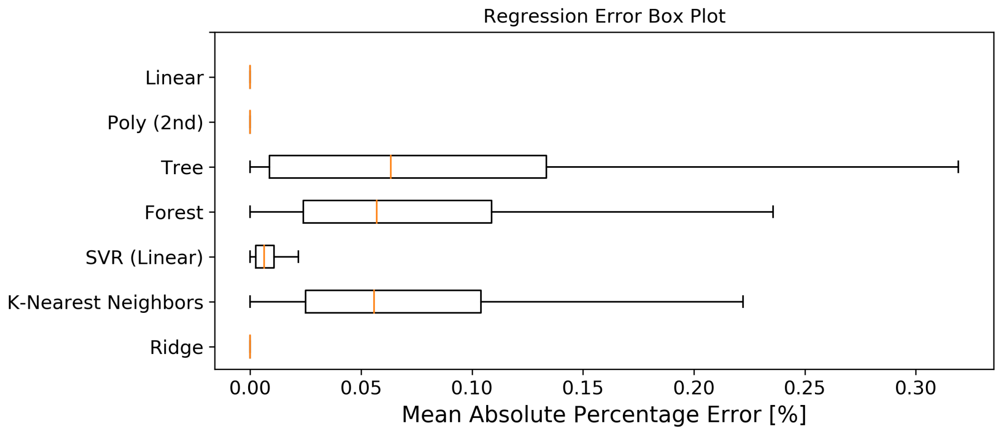


Figure 7.3: MAPE box plots of the regression models from the first WCET experiment with removed outliers.

Table 7.4: 4-Fold cross-validated MAPE on the validation set of the second WCET experiment for each trained regression model.

Regression Models	**MAPE**		
	Best	**Average**	**Worst**
Linear	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$
Polynomial (2nd Degree)	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$
Decision Tree	0.07%	0.09%	0.11%
Random Forest	0.07%	0.08%	0.09%
Support Vector (Linear)	0.008%	0.02%	0.03%
Support Vector (RBF)	1.84%	2.21%	2.54%
K-Nearest Neighbours	0.08%	0.10%	0.11%
Ridge	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$

Histograms of the input feature distributions, including the number of hidden layers and WCET labels, are shown in Figure 7.5. The distribution of the measured WCET labels has become wider within the range of $[3 \times 10^6; 6.41 \times 10^6]$ clock cycles.

The MAPEs of the trained models are listed in Table 7.4. The table shows the best, average and worst-case errors on the validation sets for each model. The best performing models are similar to the previous experiment, and thus are: Linear, Polynomial and Ridge regression. The MAPEs of these models are almost equal to zero, and therefore are annotated with the approximately sign to improve the readability of the table. Normalised box plots of the error distributions have been added in Figure 7.6. The SVR with RBF kernel was omitted once again to improve the clarity of the figure. The graphs in Figure 7.7 plot the predicted upper bounds of the execution time with respect to the actual measured results for the validation sets of all four performed cross-validations. The vertical distance between each point and the dotted line in the graph indicates the absolute prediction error for a given datapoint of the validation set.

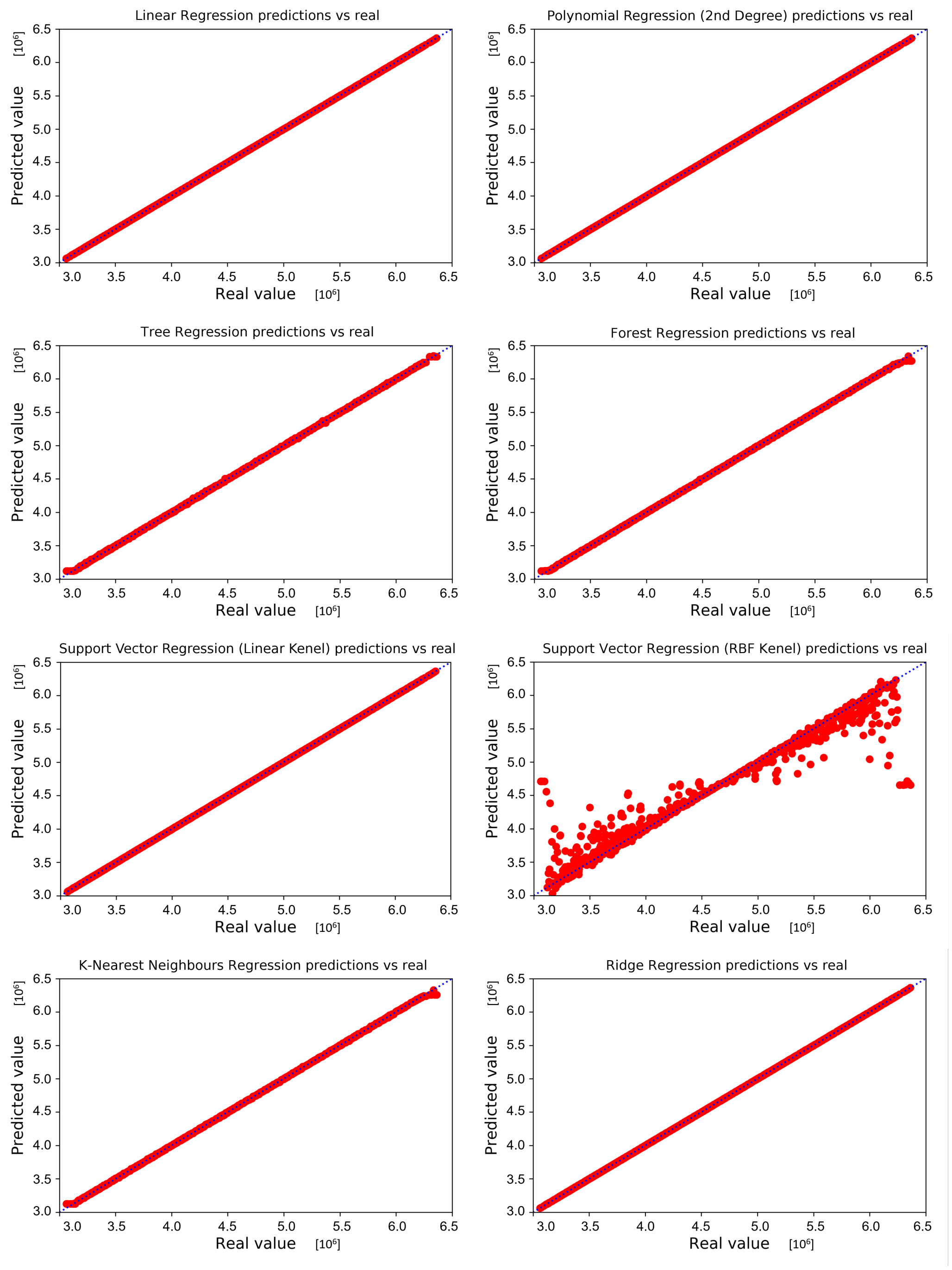


Figure 7.4: Predicted vs. real upper bounds of the regression models in the first WCET experiment on the validation sets.

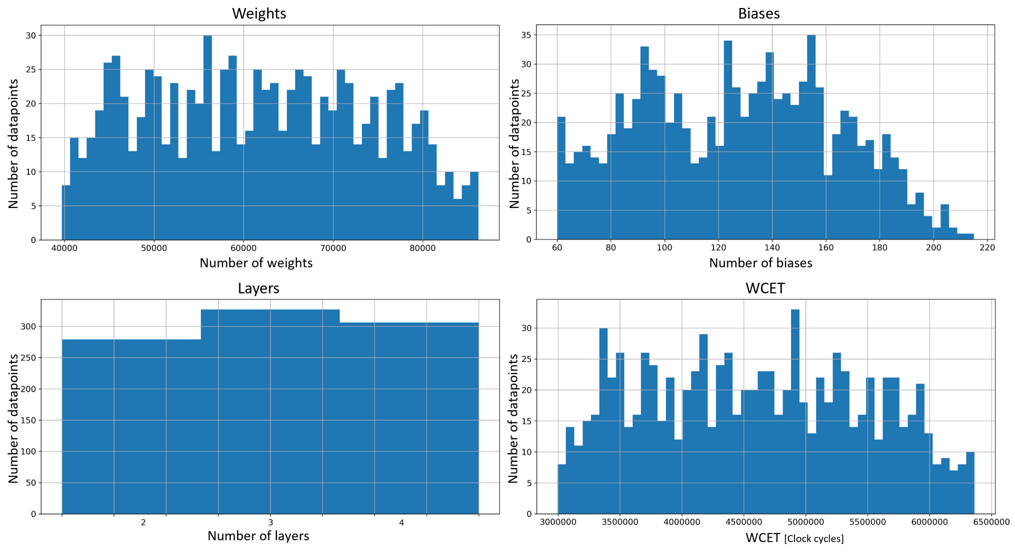


Figure 7.5: Histogram of the attributes (i.e. number of weights and biases) and label (i.e. WCET) in the full dataset of the second experiment.

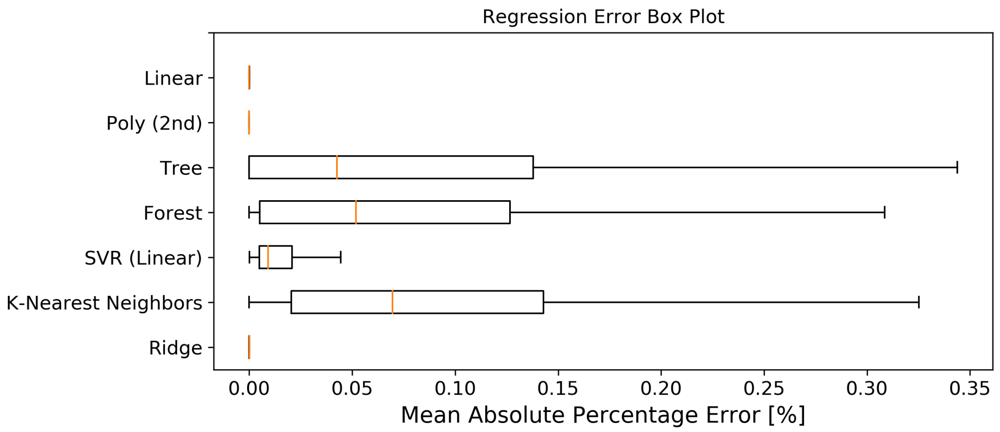


Figure 7.6: MAPE box plots of the regression models from the second WCET experiment with removed outliers.

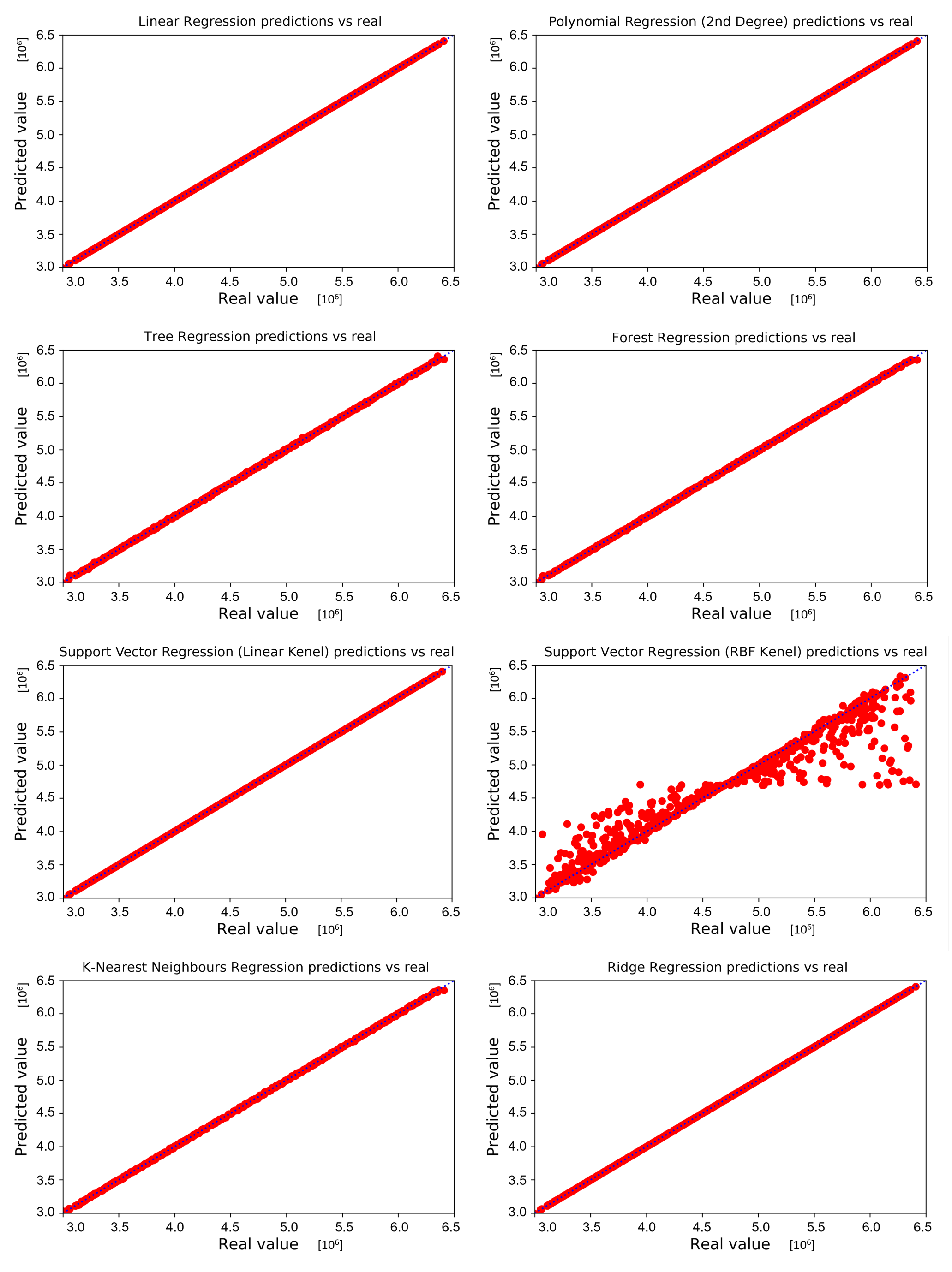


Figure 7.7: Predicted vs. real upper bounds of the regression models in the second WCET experiment on the validation sets.

7.4.3 Discussion

During the runtime of the experiments, we observed that the execution time of the DNN model is very deterministic between inference runs and does not depend on the input vector as no variations are noted between different inputs of the MNIST dataset. This is caused by the straightforward implementation of the runtime environment we used for the experiments. The models have a data-centric approach that revolves purely on a sequence of mathematical operations instead of altering the control flow in structured programming. The independence between the WCET and the input data removes a significant unknown variable which the prediction model would have had to take into account otherwise.

With the automated testbench, we were able to generate random DNN models and measure the execution time on the physical hardware for upper bound WCET analysis. A full profiling cycle took around one minute to complete, and thus a complete dataset of 1014 datapoints was acquired in a timespan of approximately 17 hours. However, the implementation of this testbench is fully automated and does not require any manual effort to complete. As a result, the datasets for these experiments are significantly larger compared to the previous experiments on software tasks with C-code that still required some manual input and validation to operate correctly.

The results of both experiments in Tables 7.3 and 7.4 show the excellent performance of the regression models. The worst recorded MAPE is only 2.58% among all models. For three models in the list, we noted errors that are almost equal to 0%. The actual reported MAPEs are as small as 10^{-7}%, or even 10^{-15}%. These small deviations are due to rounding errors caused by floating-point arithmetic [206] in the validation algorithm, as the output of the regression models are still treated as floating-point numbers. The best models that are able to perfectly fit the WCET behaviour of the system are: Linear, Polynomial and Ridge regression.

Besides the observation that the execution time of the profiled system is independent from the data input, we notice a strong linear correlation between the WCET, and the number of weights and biases. When we look at the linear correlation matrix of Figure 7.8, we see a clear linear correlation of 1 between the WCET and the number of weights. The cause of this strong correlation is the implementation of the fully connected neural network in the runtime environment. A neuron in the dense model is interconnected with every neuron from the preceding and next layer in the network. Each of these links will result in a multiply and accumulation operation, which directly impacts the execution time. The correlation of the biases, or the number of neurons, with the WCET is significantly less as we would need to specify the distribution of the neurons across each layer in order to determine the number of underlying operations. Nevertheless, excluding all but the weights feature has a negative impact on the performance of the estimation models (e.g. MAPE of the Linear regression rises to 0.1%).

The strong linear correlation between the WCET and the number of weights is one of the reasons why the best three regression models are able to perfectly predict the outcome; especially the Linear and Ridge regressions that excel in (multi-)collinearity. As the top 3 regression models have similar performances, we will compare other KPIs of these models. For instance, we have compared the training and inference time of each model in the first experiment. When we compare the relative compute time of the top 3 models in Table 7.3, we notice that the average time to train a Linear regression is the fastest, followed by the Ridge regression which is 2.15 times slower, and the Polynomial regression is the

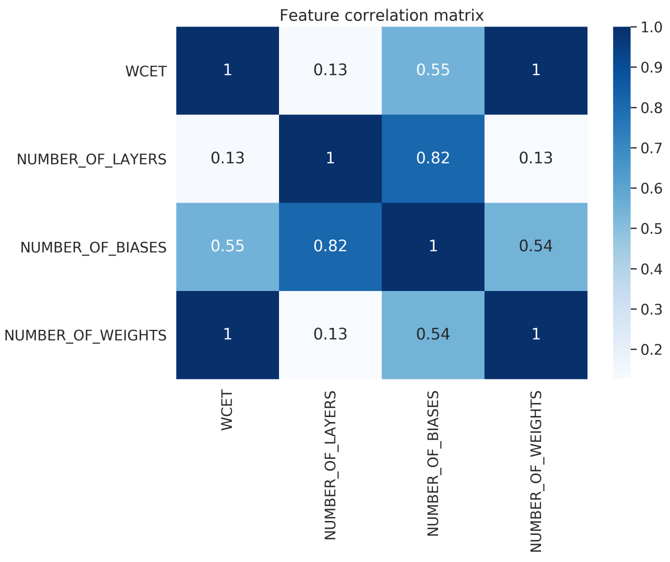


Figure 7.8: Linear correlation matrix of the features and labels in the dataset from the second WCET experiment using the Pearson correlation coefficient. A value equal to 1 indicates a linear correlation, while a value closer to 0 implies a weaker correlation between the two variables.

slowest by a factor 7.22. Nonetheless, the inference time of the Polynomial regression is significantly faster compared to the Linear and Ridge regression by a factor of 5.52 and 11.72, respectively. As each model only needs to be trained once, it may be beneficial to prefer the model with the shortest inference time, i.e. Polynomial regression.

When observing all trained models in the performed experiments, we see impressive results given that all models were trained with the default configuration without any hyperparameter tuning, with the exception of the SVR with RBF kernel. This last model has on average a relatively small MAPE. However, this MAPE is misleading as the over-, and underestimations are balancing each other out, as shown in the validation plots of Figure 7.4 and 7.7. The box plots of the SVR with RBF kernel models were removed from the figures, as the error distributions are significant worse compared to the other models (i.e. containing outliers with an error up to 30%), and thus compressing the other box plots to single lines. In contrast, the SVR with a Linear kernel performs noticeably much better and is the fourth best performing model. Notwithstanding, the computational complexity for training and executing this model is tremendously higher compared to the models of the top 3.

The last remaining regression models have a decent fit on the validation data. Nevertheless, we notice some distinct errors for the datapoints on the edges of the dataset, as shown in the plots of Figure 7.4 and 7.7. The predictions are converging to an upper and lower limit on the edges of the dataset. This behaviour is caused by the inability of these models to extrapolate datapoints that fall outside the boundaries of the training set. As a result, datapoints in the validation set with labels that exceed the 'learned' boundaries will hit the upper or lower limit of this range. If we verify the prediction error on the training sets itself, we observe that those models obtain an error equal to

0, and thus are prone to overfitting of the training dataset. These regression models are powerful predictors that to learn nonlinear relationships in data [169]. Their complexity is less ideal to solve these simple linear problems.

7.5 Use Case 2: WCET Analysis with Compiler Feature

In the experiments of the previous use case (Section 7.4), we have created estimation models that accurately predicted the WCET based on the model description of a DNN. However, these models are trained for a specific combination of target platform, compiler and their configurations. This results in the need for training a vast amount of estimation models for each different hardware platform, but also each difference in the configuration of the compiler, for example. As we discussed in Section 7.3, we want to include hardware/compiler-related features in our prediction model in order to obtain more universal models that are able to generalise over a wider variety of platforms and configurations. In this use case, we try a first attempt in creating a more generic model by including a compiler-related feature to the estimation model, namely the optimisation flag.

7.5.1 Experimental Setup

This experiment uses an identical setup as the one used in Use Case 1 of this chapter (Section 7.4). The WCET of each DNN task will be predicted by an estimation model for the Nordic nRF52 development kit board with the nRF528232 SoC [177]. The cross-compiler that we employ in this setup is the embedded ARM GNU Toolchain, version 10.3-2021.10 for 32-bit ARM Cortex processors.

The original dataset of *Experiment 1* from the previous use case will be reused for the training set. These datapoints are compiled by the ARM GNU compiler with no optimisation enabled (i.e. -O0). An identical set of datapoints is generated by the Neural Network Generator with the same features (e.g. number of weights and biases). However, the compiler optimisation flag for this set is now configured to level three (i.e. -O3). Therefore, the final dataset for this experiment contains 2028 labelled DNN models.

7.5.2 Results

Each datapoint in the dataset contains the features from the model description as shown in Table 7.1. However, we added the compiler optimisation flag as a first new feature that is not related to the software/model. This feature is an enumerated type that indicates the chosen option. We also compared the performance of the regression models by using a *one-hot* encoded vector instead of the enumerated type for the compiler flag, but no differences were noted.

The complete dataset consists of two identical sets of DNN models with a different compiler optimisation flag configured. The distributions of the input features in the dataset are equal to the ones in the first experiment of Use Case 1 as illustrated in Figure 7.2 with the difference that the absolute values are doubled. The WCET labels of the dataset are plotted in the histogram of Figure 7.9. We notice two different curves on the WCET histogram. The curve on the right are the models that are compiled with optimisation disabled (i.e. -O0) and is equal to the WCET histogram of Figure 7.2. The curve on the left contains the models with the optimisation flag set to level 3 (i.e.

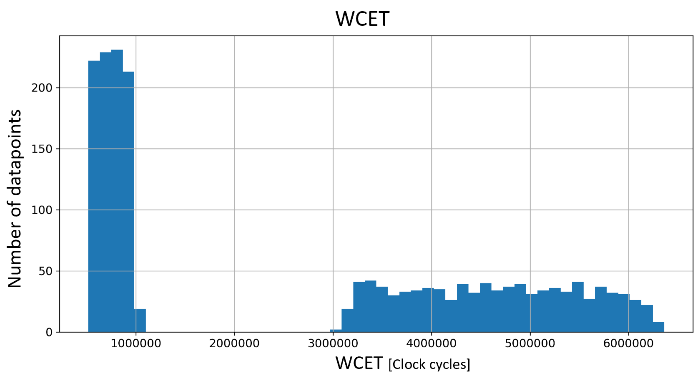


Figure 7.9: Histogram of the WCET label in the full dataset.

Table 7.5: 4-Fold cross-validated MAPE on the validation set with varying compiler optimisation flag for each trained regression model.

Regression Models	**MAPE**		
	Best	**Average**	**Worst**
Linear	24.5%	26.5%	28.2%
Polynomial (2nd Degree)	0.001%	0.001%	0.002%
Decision Tree	0.10%	0.11%	0.12%
Random Forest	0.09%	0.10%	0.11%
Support Vector (Linear)	44.6%	156.2%	277.9%
Support Vector (RBF)	134.9%	152.5%	166.3%
K-Nearest Neighbours	151.5%	156.6%	167.7%
Ridge	24.5%	26.5%	28.1%

-O3). Notice the width of both curves as the standard deviation of the -O3 samples is significantly smaller compared to the -O0 samples.

Using a 4-fold cross-validation process, we trained and validated eight different regression models from the list in Table 7.2. Based on the validation data, we calculated the MAPE error metric as defined by Equation 7.1. The training and validation of all models took around 15 min to complete on a laptop with an Intel i7-6700HQ CPU processor. While the training of the prediction models requires considerable computational power, its resource requirements are less important compared to the prediction performance as the profiling process is carried out offline on the developer's computer before deployment. The results in Table 7.5 provide the best, average and worst-case MAPEs that were registered during the four runs of the cross-validation process for each regression model.

The best performing model in the experiment according to the MAPE on the validation set is the Polynomial regression. The worst recorded MAPE of this regressor is only 0.002%. Figure 7.10 shows normalised box plots of the error distribution from the datapoints of the validation sets for each regression model. Finally, the graphs in Figure 7.11 plot the predicted upper bounds of the execution time with respect to the actual measured results of all four cross-validations performed on the validation set. The ver-

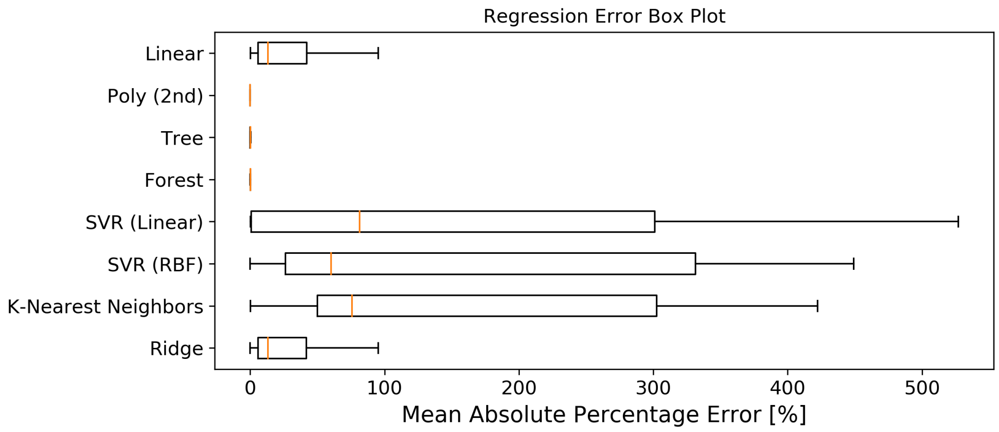


Figure 7.10: MAPE box plots of the WCET regression models from the varying compiler flag experiment with removed outliers.

tical distance between each point and the dotted line in the graph indicates the absolute prediction error for the respective datapoint.

7.5.3 Discussion

After enabling the compiler optimisation, we see a major decrease of the execution time for all models. However, we still observe that the execution time is not dependent on the input vector as no variations are noted between different inputs of the MNIST dataset. The data-centric approach of the model avoids the usage of data-dependent control flow operations in the compiled code (i.e. branchless programming), and thus minimises, or even eliminates variations in the execution time.

The results in Table 7.5 show the very accurate predictions from the Polynomial regression. The Tree and Forest regressor models closely follow its performance with MAPEs that are lower than 0.12%. In contrast, all other trained regression models had a much harder time to accurately estimate the upper bounds for the validation set. Their MAPEs vary between 24.5% up to 277.9%.

The regression models that are based on linear correlations performed significantly worse in this experiment compared to the previous ones in Section 7.4. When we look at the linear correlation matrix of Figure 7.12, we see that there is no full correlation anymore with the number of weights, or biases. This drop in linear correlation is due to the combination of two linear systems (i.e. -O0 and -O3 models) into one. The distribution of WCET labels in Figure 7.9 contains two separate curves with different widths. The standard deviation of the left curve (i.e. -O3 samples) is only 1.34×10^5 clock cycles compared to the wider right curve (i.e. -O0 samples) with a standard deviation of 8.89×10^5 clock cycles. Both curves belong to a linear system with different coefficients. By combining both systems into one, we lose the linear correlation between the features as is shown in the correlation matrix of Figure 7.12. Additionally, we see a negative correlation between the WCET and compiler optimisation flag. This inverse correlation results from the chosen enumeration. The -O0 and -O3 samples have a respectively

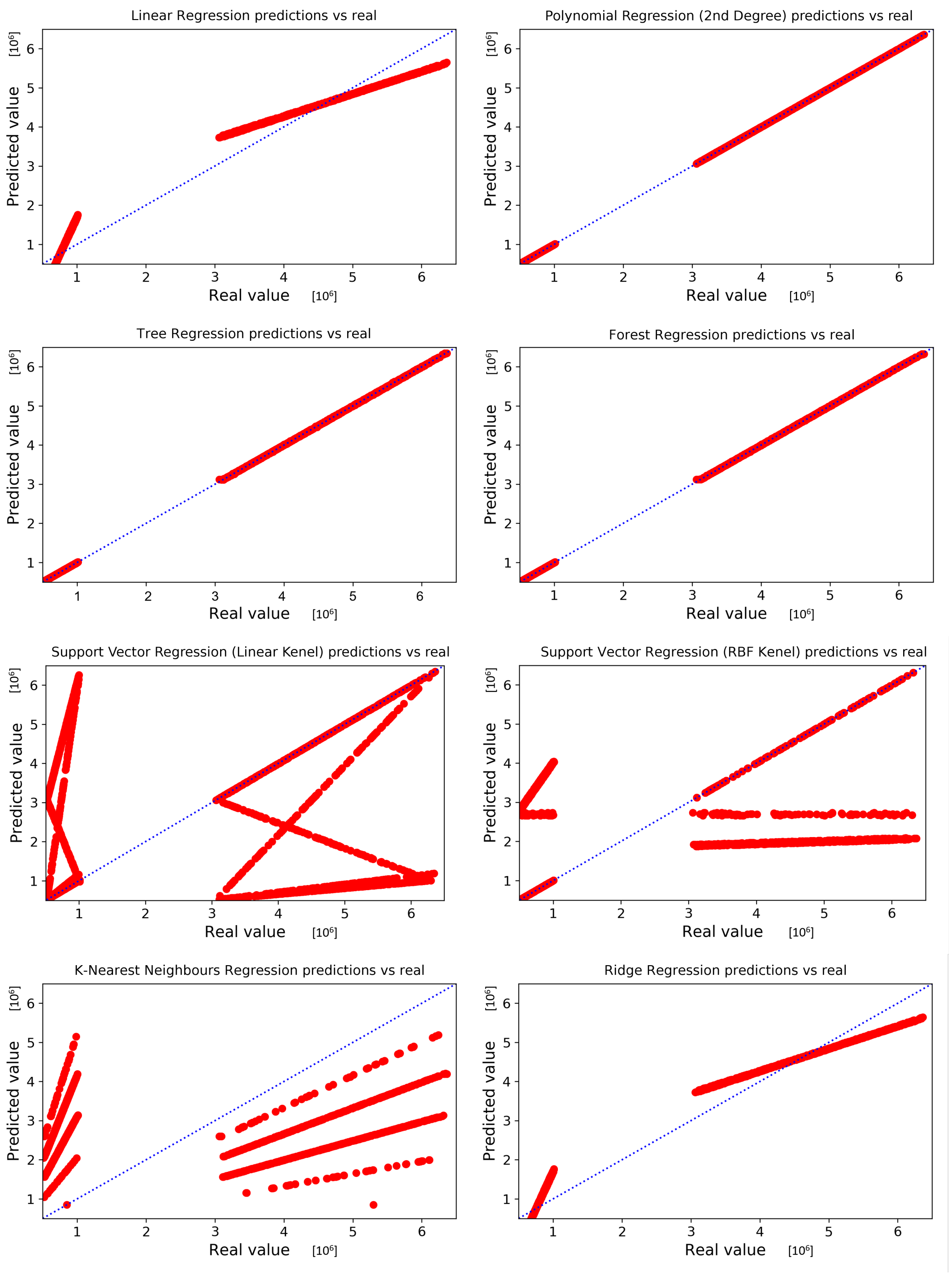


Figure 7.11: Predicted vs. real WCET upper bounds of the regression models in the varying compiler flag experiment on the validation sets.

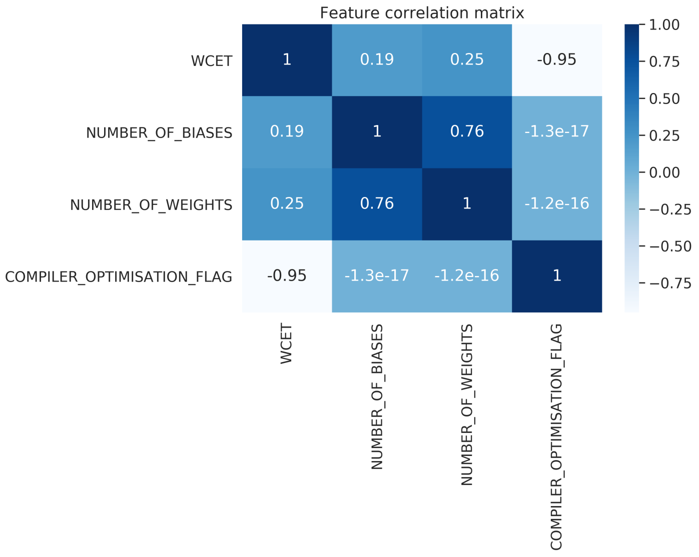


Figure 7.12: Linear correlation matrix of the features and labels in the dataset of the WCET experiment with varying compiler flag using the Pearson correlation coefficient. A value equal to 1 indicates a linear correlation, while a value closer to 0 implies a weaker correlation between the two variables. A value closer to -1 indicates a negative linear correlation.

feature value equal to 1 and 2, while the order of the WCET labels are inverse (i.e. the -O0 samples are higher compared to the -O3 samples).

The Linear and Ridge regression are their best in finding linear correlations in data. As our current system is now from a higher order and not linear anymore, we notice that these models are underperforming. The validation plots of Figure 7.11 clearly show the difference in the incline of both slopes of the -O0 and -O3 samples, as the linear regressors try to fit both sets with minor success. A solution to this problem would be to use a higher order regression, as we clearly see from the results. The results of the other regression models are in particular bad. The crossing and parallel patterns on the validation plots show that each cross-validation run is not able to generalise the problem but is actually overfitting the training samples. Since we already have a high performing estimation model, we will not try to further tune these models to avoid overfitting.

7.6 Use Case 3: WCEC Analysis of Neural Networks

Up to this point, we have applied our hybrid estimation methodology to estimate the WCET of tasks driven by neural networks. In this section, we will extend our setup to train estimation models that are able to provide an upper bound on the energy consumption of the neural network.

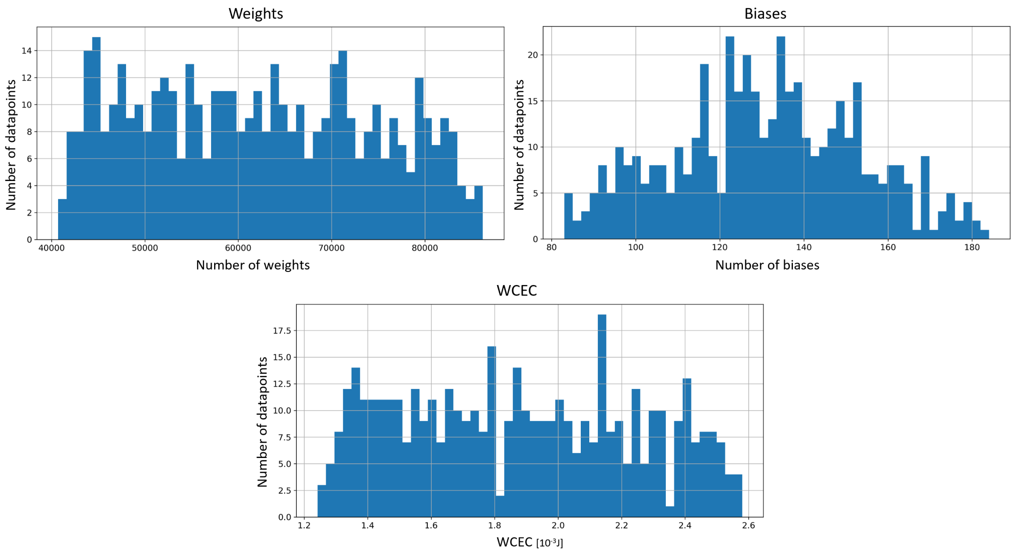


Figure 7.13: Histogram of the attributes (i.e. number of weights and biases) and label (i.e. WCEC) in the full dataset of the WCEC experiment.

7.6.1 Experimental Setup

This experiment uses an identical setup as the one used in Section 7.4. The original dataset of *Experiment 1* from Use Case 1 will be reused for the training set. The profiling setup has been altered to obtain energy measurements. Therefore, we use the automated testbench setup with the Joulescope as described in Section 5.3. The DUT in this experiment is the Nordic nRF52 development kit board with the nRF528232 SoC [177]. Each model in the dataset is executed multiple times on the DUT with a randomised set of input vectors from the MNIST database. By running more input vectors per model, the profiling time for one datapoint rose enormously to 3-5 minutes. As a result, we stopped the experiment when the dataset reached 500 datapoints. All results are captured by the Profiling Assistant to generate an output file for the regression models. In the next section, we discuss the acquired measurements for WCEC and the results of the trained estimation models.

7.6.2 Results

Each of the 500 datapoints contains the features that are listed in Table 7.1 together with the upper bound energy measurement from the testbench as label. The number of hidden layers is fixed to two hidden layers with variations of the number of weights and biases in the model, as described in Experiment 1 of Section 7.4. The distribution of the input features in the final dataset is presented in the histograms of Figure 7.13. The WCEC is measured by the external energy analyser (i.e. Joulescope) that captures the energy in Joules (J) that is consumed during a single inference. The range of the WCEC labels in the recorded dataset is equal to $[1.242 \times 10^{-3}; 2.58 \times 10^{-3}]$ J.

Using a 4-fold cross-validation process, we trained and validated eight different regres-

Table 7.6: 4-Fold cross-validated MAPE on the validation set of the WCEC experiment for each trained regression model.

Regression Models	**MAPE**		
	Best	**Average**	**Worst**
Linear	0.23%	0.25%	0.26%
Polynomial (2nd Degree)	0.23%	0.25%	0.26%
Decision Tree	0.34%	0.36%	0.40%
Random Forest	0.30%	0.31%	0.31%
Support Vector (Linear)	0.23%	0.25%	0.27%
Support Vector (RBF)	5.26%	5.85%	6.50%
K-Nearest Neighbours	0.30%	0.30%	0.31%
Ridge	0.23%	0.25%	0.26%

sion models from the list in Table 7.2. Based on the validation data, we calculated the MAPE error metric as defined by Equation 7.1. The training and validation of all models took around 10 min to complete on a laptop with an Intel i7-6700HQ CPU processor. While the training of the prediction models requires considerable computational power, its resource requirements are less important compared to the prediction performance as the profiling process is carried out offline on the developer's computer before deployment. The results in Table 7.6 provide the best, average and worst-case MAPEs that were registered during the four runs of the cross-validation process for each regression model.

The best performing models in the experiment according to the MAPE on the validation set are: Linear, Polynomial, and Ridge regression. These are the same models that performed extremely well in the first experiment of Section 7.4. The normalised box plots of Figure 7.14 show the error distribution from the datapoints of the validation sets for each regression model. The SVR with RBF kernel was omitted from the figure in order to improve the clarity of the other box plots. Finally, the graphs in Figure 7.15 plot the predicted upper bounds of the energy consumption with respect to the actual measured results of all four cross-validations performed on the validation set. The vertical distance between each point and the dotted line in the graph indicates the absolute prediction error for the respective datapoint.

7.6.3 Discussion

With the automated testbench, we were able to generate random DNN models and measure the energy consumption on the physical hardware for upper bound WCEC analysis. A full profiling cycle took around three to five minutes to complete, and thus the dataset of 500 datapoints was acquired in a timespan of approximately 33 hours. However, the implementation of this testbench is fully automated and does not require any manual effort to complete.

During the runtime of the experiment, we observed that the energy consumption slightly varies with different input vectors, but also between runs with the same input. These variations occur due to noise and sampling errors during measurement as discussed in Section 5.3.3. Furthermore, the dynamic power dissipation depends on the energy consumption caused by the switching of internal logic [70]. Changing the input vector

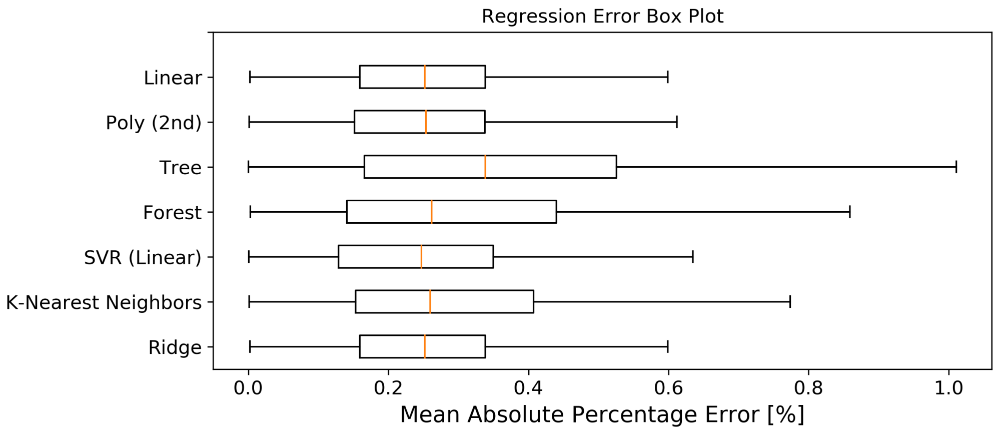


Figure 7.14: MAPE box plots of the regression models from the WCEC experiment with removed outliers.

will result in varying dynamic power consumption as these are data dependent. When we take a closer look at the standard deviation (σ) of measurements with the same input vector, we see an average of $\sigma = 9.91 \times 10^{-7}$ J. The standard deviation between samples of varying input vectors is slightly higher with an average of $\sigma = 1.23 \times 10^{-6}$ J.

The results in Table 7.5 show that the best performing regression models have a MAPE of 0.26% in the worst-case. All models with the exception of the SVR with an RBF kernel have a MAPE of below 1%. These results have a similar pattern compared to the previous WCET experiments. When we look at the linear correlation matrix of Figure 7.16, we see a clear linear correlation of 1 between the WCEC and the number of weights. We presume that the static power consumption takes a major part of the total consumption, and the variation in the dynamic power consumption is small as the input data does not impact the control flow or the number of operations performed. However, a more extensive experiment is required to study the impact of the input vector on the dynamic power consumption of the switching logic.

Based on the results from our previous WCEC experiment in Section 6.7, we have scaled all our labels by the power of 10^{10} to have enough significant digits before the decimal point, as our dataset contains on average power measurements with an order of magnitude of 10^{-3} J. Without this pre-processing of the dataset, we would obtain poor performance for several regression models, such as the Tree, Random Forest and SVR regressors. After this small optimisation, we obtain very accurate estimation models given that no tuning of the hyperparameters has been performed, with the exception of the SVR with RBF kernel. The box plots of the linear regressors (top 3) in Figure 7.17 are nearly identical and indicate that most errors stay under 0.6%.

The MAPE is an excellent metric to gain insight into the performance of the prediction models. However, there is no indication if the models are mostly over- or underestimating the WCEC. In the context of WCRC analysis, it is important to keep the upper bound above the real worst-case to avoid compromising the functional behaviour of the system. Therefore, we calculated the MRPE for each model in Figure 7.17. MRPE calculates the percentage difference between the predicted and actual values, but keeps the sign

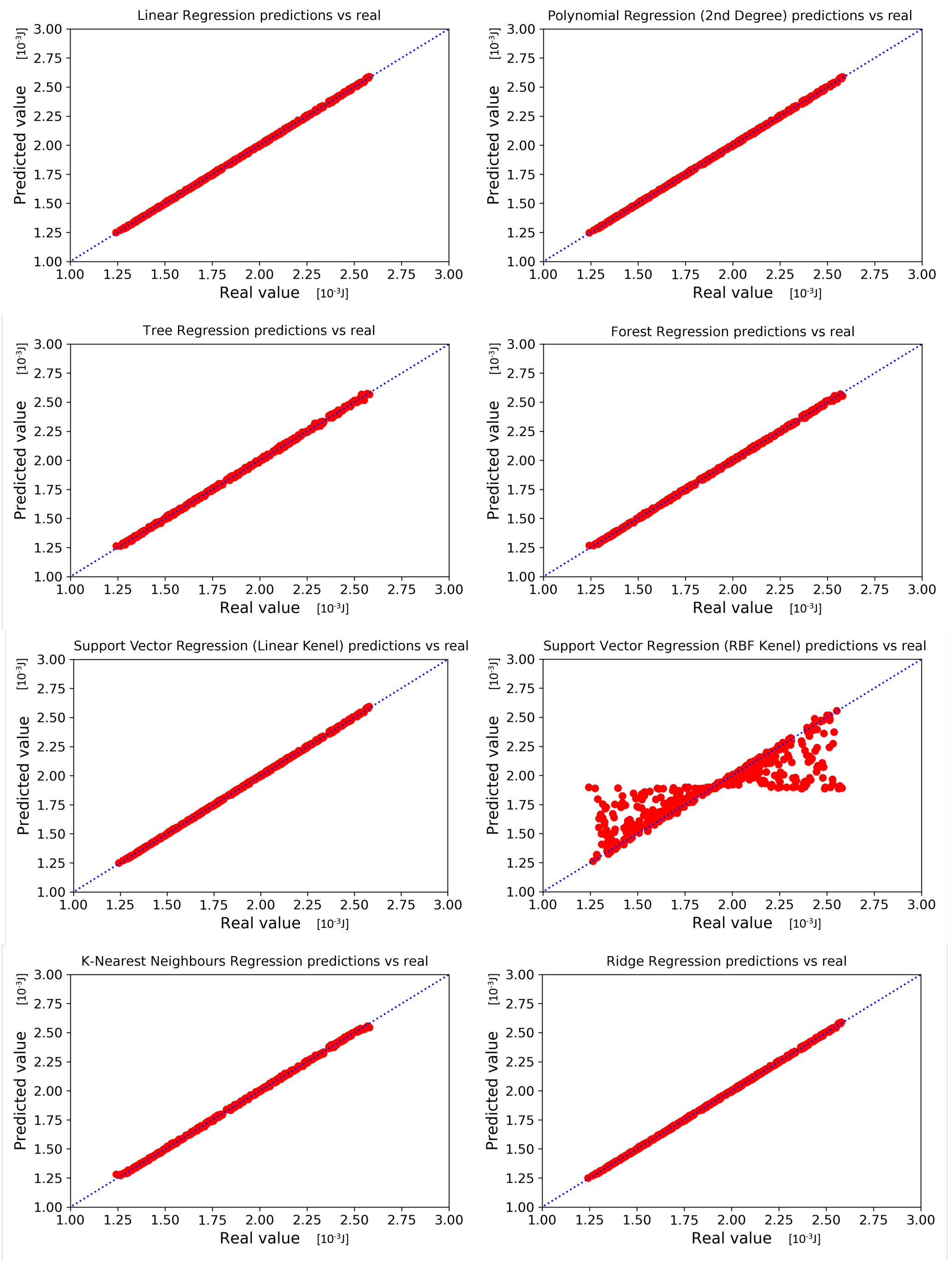


Figure 7.15: Predicted vs. real upper bounds of the regression models in the WCEC experiment on the validation sets.

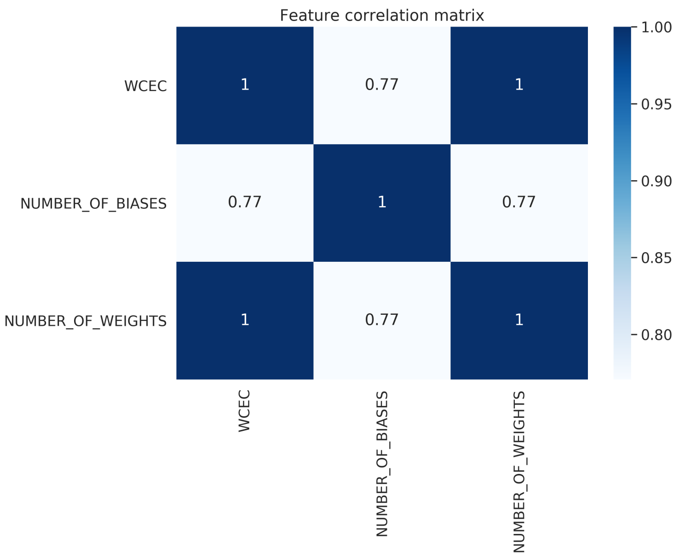


Figure 7.16: Linear correlation matrix of the features and labels in the dataset from the WCEC experiment using the Pearson correlation coefficient. A value equal to 1 indicates a linear correlation, while a value closer to 0 implies a weaker correlation between the two variables.

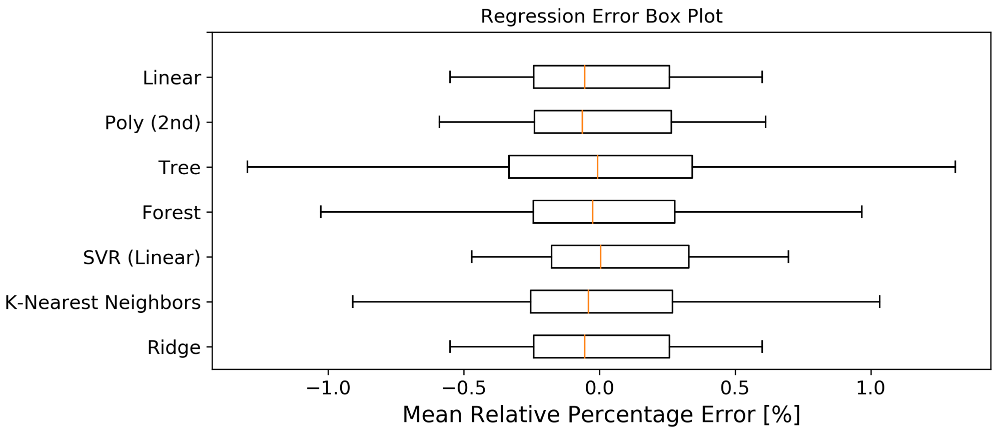


Figure 7.17: MRPE box plots of the regression models from the WCEC experiment with removed outliers.

of the error. These box plots have a nearly symmetric shape with a slight shift to the left, indicating that most of the regression models are underestimating the WCEC. This behaviour is less desirable in our context but achieving models with an accuracy higher than 99% is excellent in the scope of providing early-stage estimation.

7.7 Conclusion

In this chapter, we have adapted the ML-based hybrid WCRC analysis in order to determine the WCET and WCEC of software tasks that are driven by neural networks. The resource behaviour of these tasks differs compared to traditional programmed tasks, due to their data-centric nature that involves purely arithmetic operations instead of branching control flow.

Using a custom runtime framework for TensorFlow models, we were able to train highly accurate estimation models for fully connected DNN. The worst-case predictions are obtained from the high-level model description of the neural network. The deterministic behaviour of the models on the runtime framework made it plausible to acquire regression models with an average prediction error of less than 1%. The best performing regression models in the experiments are linear regressors. Therefore, it is important to select an ML type that best suits the problem.

In a secondary experiment, we have introduced the optimisation flag of the compiler as an external system feature to validate the ability of the regression models to generalise across different platforms and/or configurations. The increase in system complexity was easily tackled by the Polynomial regression, and therefore makes it feasible to extend the research to include more platform-related features to generalise the estimation models even further.

Despite the fact that we utilised our custom runtime environment to perform the model inference on the hardware, we have extended our framework to support future research on embedded DNN resource behaviour, and with the objective of **RQ 3** (Section 1.3) we have demonstrated that we could accurately predict the WCET and WCEC behaviour of our runtime environment using the ML-based hybrid WCRC methodology.

Chapter 8

Conclusions

In this dissertation, we examined a hybrid methodology with ML to approximate the WCET and WCEC of software and neural networks on hard constrained embedded devices, such as IoT and CPS. In order to create this estimation methodology, we have composed a research trajectory based on the following three RQs:

- **RQ 1**: *How can we approximate an upper bound on the WCET/WCEC that is computational acceptable to perform and still provide a sound bound, and can we create a dynamic trade-off between those two?*
- **RQ 2**: *How can the hybrid WCRC analysis be employed to provide early predictions without performing physical measurements on the target platform?*
- **RQ 3**: *How can we predict the WCRC of neural network powered control logic by solely using descriptions of these networks?*

We have translated these RQs in a three-stage plan that elaborates on each of these questions one at a time. In the first stage, we have performed an in-depth study on the influences of the hardware and software on WCRC behaviour, WCRC analysis methodologies, and their advantages and disadvantages in order to propose and examine a hybrid strategy. This methodology creates an opportunity to perform a trade-off between the static and measurement-based approach. By building our own framework COBRA, we are able to generate hybrid blocks, and profile them on the target hardware. The block splitting strategy allows us to facilitate the profiling effort to obtain an upper bound estimation of the WCRC as discussed in Chapter 5, and therefore addresses **RQ 1**.

In the second stage, we elaborated on the hybrid methodology by integrating an ML-based component to exchange the measurement-based layer. This prediction layer allows us to provide upper bound estimates of the hybrid blocks without having to run the code on the target hardware, and therefore is our main contribution in Chapter 6 to tackle **RQ 2**. The early-stage estimations allow developers to gain insight in the resource consumption during code writing. Additional tools and extensions have been built for the COBRA framework to implement this new strategy. In this research, we have focused on software-related features to establish WCET and WCEC upper bound predictions for a set of tasks. We evaluated and compared the performance of different regression and two DNN models, which showed promising results for the former group of prediction models.

The final stage in our research is a synthesis of the previous stages for which we employ our knowledge to predict the WCRC behaviour of neural networks on embedded devices. In order to address **RQ 3**, we have extended our methodology and the COBRA framework to generate and profile neural networks on a target platform. In the context of this thesis, we successfully trained regression models that are able to predict the WCET and WCEC of a fully connected neural network with almost zero error. These predictions are performed on a high-level description of the network which allows us to obtain early-stage estimations of the WCRC. In addition, we examined the feasibility of creating more generalising estimation models by introducing platform-related features. Therefore, we implemented the compiler optimisation flag as an external system feature with almost no penalty on the prediction accuracy.

Throughout the PhD research, we have developed a methodology that enables us to perform WCRC estimations using ML and created an accompanying framework from scratch that implemented these techniques in order to validate them. As we could only focus on some key contributions within the time frame of this thesis, we established to create a more elaborate approach that further extend our hybrid methodology to improve its accuracy and applicability in development environments. Furthermore, the modular design of the COBRA framework (Appendix A) is an adequate basis to further build and extend upon with new and improved methodologies. In Chapter 9, we briefly discuss some opportunities to improve our work, and to further extend the hybrid methodology as presented in this thesis.

Chapter 9

Future Work

With our contributions in this work, we have established a hybrid methodology for WCRC analysis together with a framework to apply it. Nonetheless, there is still a significant amount of future work to be explored within this research field. In this section, we provide a concise list of improvements and extensions that are worth investigating in follow-up research:

- **Size of the hybrid blocks**: One of the key features of the hybrid analysis methodology is the introduction of hybrid blocks that split the code into smaller components for the measurement-based approach in the first layer of the analysis. The size of these hybrid blocks allows us to make a trade-off between the complexity of the static analysis and the precision of the measurement-based layer. The Block Generator implements two strategies to vary the size of the generated blocks. In the experiments of Section 6.6, we used two datasets using differently sized hybrid blocks of the TACLeBench code. In most experiments, we used relatively small hybrid blocks with few instructions. With the hybrid analysis method, we need to explore the balance and impact on the static and measurement-based layer when splitting the code in varying sizes. In future research, we could explore the influence of using larger abstracted hybrid blocks on prediction models, and create new block splitting strategies that optimise the trade-off of both analysis layers;
- **Hardware/compiler features**: The estimation models in the experiments were solely trained on software-related features. Nonetheless, the resource consumption strongly depends on the actual instructions that are executed by the hardware. The dependency of the hardware and compiler in our experiments are covered by the ML models that are specifically trained for the specific target and compiler chain. In the methodology we presented in Section 6.2, we included the hardware component into our approach to help the estimation model to generalise the problem, so that one trained model is able to handle a wider range of target platforms instead of having a separate model for each setup. A first step in future research would be to create models for groups (or families) of target platforms that are distinguishable by key parameters, such as size of cache memory, compiler flags, number of pipeline stages, etc. Therefore, an extensive study on the influences of the hardware on WCRC would aid in defining relevant input features for the estimation models;

- **Feature engineering and parameter tuning**: The selection of the right features is an essential part in creating high performing ML models. On the one hand, we need to include all features that have a significant impact on the resource consumption of a hybrid block. On the other hand, selecting too many features with less or no relevance will eventually lead to poor performance. This process is also referred to as feature engineering [169]. In order to acquire a set of qualitative features, we need to gain insight in the actual data while looking for correlations between them [170], [171]. The current list of features used in the experiments are focused on software-related features that directly resemble instructions and changes code flow. In future research, we need to study the effects of data and interinstruction overhead in order to evaluate which features are relevant for the estimation model to apprehend these influences. For instance, the energy consumption of the processor is influenced by the data itself, as discussed in Section 2.1.2 by other research. The current attributes in our experiments do not take any data-related features into account, except of their scope. Attempts should be made to include more data-related features to the feature list, such as data/array types and sizes, etc.

  Furthermore, each ML model is defined by a set of parameters configuring the training process and the model itself. These adjustable parameters are called hyperparameters [169], e.g., number of hidden layers, learning rate, error penalty, etc. The tuning of hyperparameters is a computational expensive and timing consuming process as the model needs to be trained and re-evaluated over and over again until an optimal MRPE is reached. In practise, hyperparameter tuning can be achieved by a grid search, where each combination of hyperparameters is cross-validated to find the best set [169]; or more efficient stochastic search algorithms that explore the state space of the hyperparameters by introducing randomness or noise to the search, such as the TPOT algorithm [179]. For example, the Support Vector Regression with an RBF kernel in the experiment of Section 6.7 has been tuned with a grid search. The default configuration of the hyperparameters, i.e., C: error penalty and ϵ: learning rate, had a real poor performance compared to the other models. This tuning process can be further optimised and extended to the entire processing pipeline with the use of automated AutoML methods [178];

- **Increase training dataset**: In addition to well-designed features, we need an extensive dataset to train any prediction model. The automated testbench is a first step in quickly acquiring more data. Nevertheless, there are still opportunities to increase the precision of the upper bound measurements and speed up the profiling process. One of these improvements would be to include static data analysis in the flow graph. This would allow us to find and generate data input that aims at triggering WCRC paths and lowers the need of manual code annotation. Another approach is to increase the size of the dataset by applying augmentation techniques as proposed by [82] and [207]. In their research, they created a source code generator that produces pseudo-random code based on the requested characteristics, e.g., flow facts, number of variables, type of instructions, etc.;

- **Selection of estimation models**: In our conducted experiments, we prominently focused on using regression models to predict the WCRC. Using other models, such as DNN, have the potential to learn complex nonlinear behaviour of the system compared to regression models. However, these will require much more training data to achieve any reasonable results. Another approach is to combine different models into one ensemble to create one powerful predictive model, as it eliminates outliers of individual models. The accuracy of an ensemble model is often higher than the best model in the group [169] because it combines the optimal performance of different models in different areas;

- **Estimation models for other ML runtime frameworks**: The runtime environment we employed in our neural network WCRC experiments is a custom-built framework that provides full operational customisation. As we showed in the results of Chapter 7, the behaviour of the runtime is straightforward and very deterministic, which made it plausible to create very accurate estimation models. However, other ML frameworks, such as TensorFlow, and PyTorch, have a high adoption rate in research and industry. Therefore, an interesting research project is to extend our methodology to other frameworks and examine their impact on the system behaviour and WCRC predictability.

Appendix A

Code Behaviour Framework (COBRA)

The COBRA framework developed by the IDLab research group is an open-source tool to perform various types of analysis to examine the performance and behaviour of code on different architectures. The framework allows the developer to optimise the resource consumption on (currently) three main levels:

- **WCET/WCEC analysis**: different techniques exist to determine the WCET and WCEC. The COBRA framework implements both static and dynamic analysis to determine the resource behaviour;
- **Scheduler optimisation**: information about the resource consumption, such as time (i.e. computing units) and energy, are used to optimise scheduling algorithms towards these resources for a specific application [5];
- **Design pattern-based performance optimisation for multi-core processors**: applications can benefit from large speed up on multi-core systems by parallelising their algorithms on different levels (e.g. code-based parallelism, instruction-level parallelism). However, finding and implementing the right design patterns to apply parallelism is a time-consuming task. The framework automates the process in finding the best design pattern for parallelism at the appropriate level [129].

An overview of the COBRA framework is provided in Figure A.1. The framework itself consists of three parts which can be used separately. The first section contains the input data from different sources, such as the TACLeBench suite [60], generic input programs or specific applications for a given target hardware platform. The second stage transforms the input to a framework defined description language format used by the different tools. This standardisation allows new analysis tools to be easily integrated within the COBRA framework. The last stage includes all tools used to analyse and optimise the resource consumption for the three levels discussed above.

In the work of this thesis, the WCRC analysis tools were integrated within the framework. This extension is called the *Hybrid Program Analyser*, or for short COBRA-HPA. This tool will parse a given C-source file and create a hybrid block tree on which we are able to perform algorithms or apply rules to. Next, the framework creates corresponding

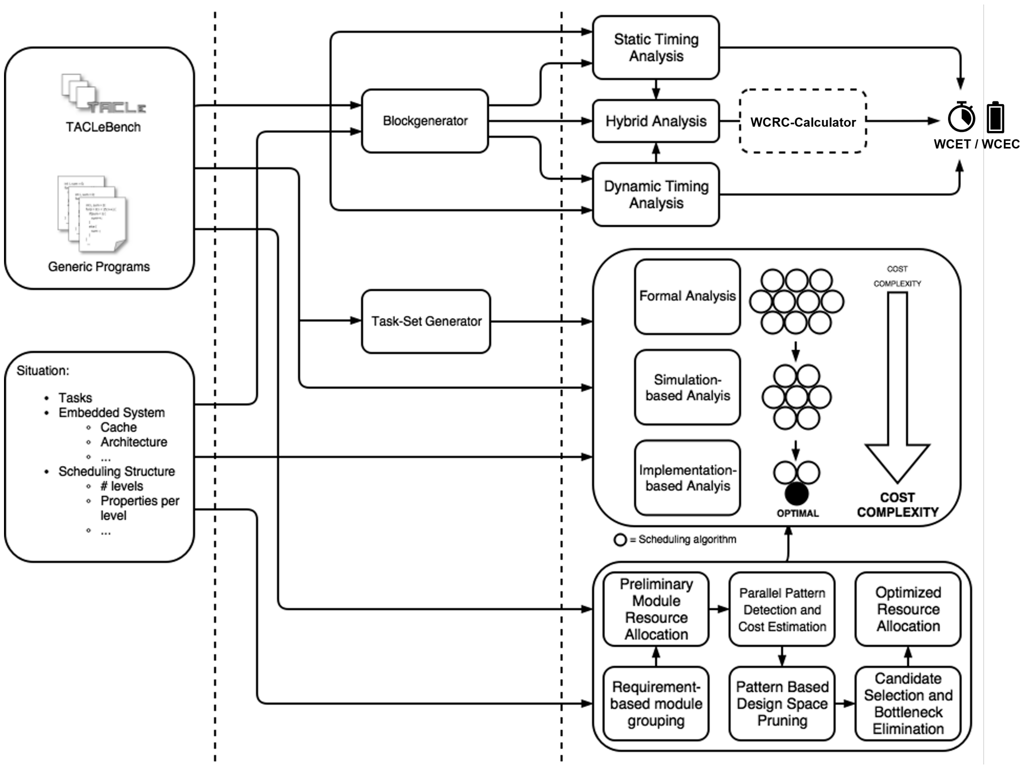


Figure A.1: Schematic overview of the COBRA framework. Three sections are distinguishable by the dotted lines. The first section on the left contains the different sources of input data. The middle section provides a set of tools that transform the input data of the first section and provide the output to the analysis tools of the third section on the right. The analysis tools from the top to bottom are WCRC analysis, optimal scheduling algorithm analysis and design pattern-based performance optimisation for multi-core processors.

source files of each block that are used to run measurement-based analysis on. A schematic overview of the COBRA-HPA framework is shown in Figure A.2. In the following sections, we explain the different modules of this toolchain and their implementation.

A.1 Project Creation

The HPA extension of the COBRA framework is a chain of tools that execute each step sequentially as illustrated in Figure A.2. The functional operation of the framework is based on project files. In order to perform an analysis on a code project, one needs to define a Project Configuration file that contains the file locations of the source code and several configurations of the desired analysis strategy, such as block splitting and target hardware configurations. This configuration file is stored in a dedicated Project folder that will contain all generated files of the project by all the COBRA tools. These project files are passed through the chain and, therefore, allow the user to interrupt and continue operation between steps, or change settings of one step in the chain without having to

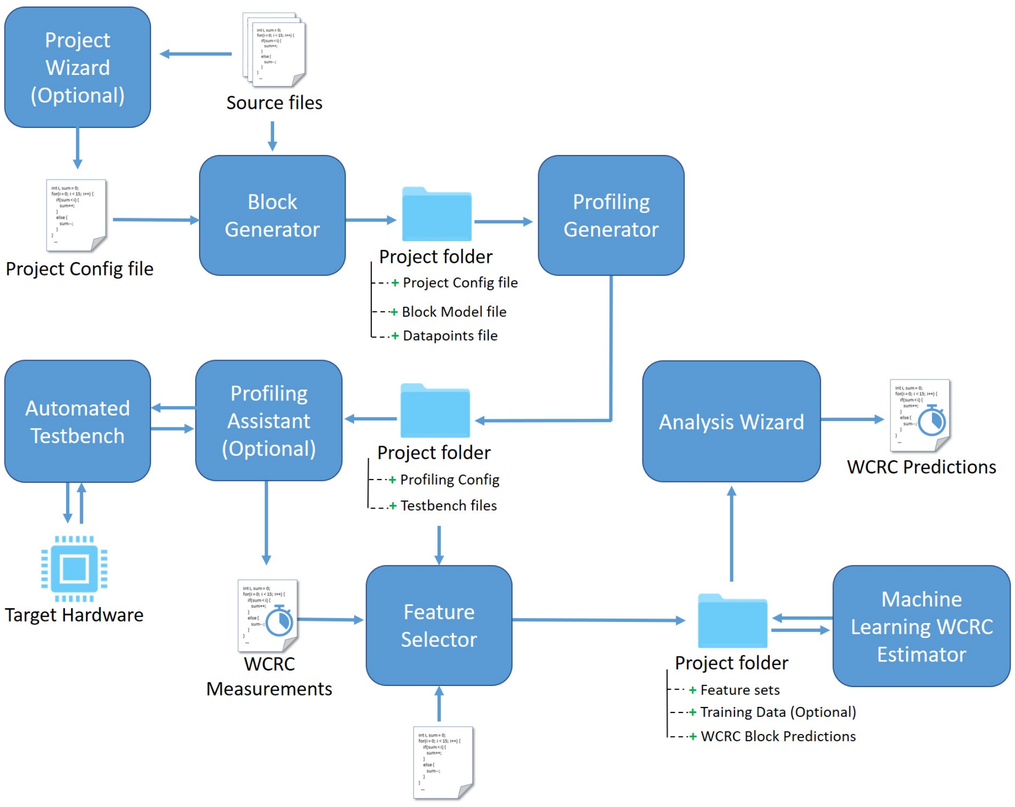


Figure A.2: Schematic overview of the COBRA-HPA toolchain.

rerun the process from the start. The Project Configuration file is an Extensible Markup Language (XML) file which the user is able to write on its own. Nevertheless, we have created an easy-to-use tool that assists the user in creating and managing these project files with the Project Wizard.

A.2 Block Generation

The first module in the HPA toolchain is the *Block Generator*. This tool allows us to generate a block representation of the input application according to the hybrid methodology, as discussed in chapter 5. In order to generate a block model, the Block Generator needs two different input sets as illustrated in Figure A.2, namely a project configuration file and the source files on which the analysis needs to be applied. After completion, the tools will create a project folder with the resulting block models. The project folder will contain a standardised model file which encodes the generated block models and the original source files in order to cross-reference each block to its original code segment. This project folder is then available to be used by the other tools.

The functionality of the Block Generator consists out of two major parts: parsing of the source files and generating a block representation model of these files.

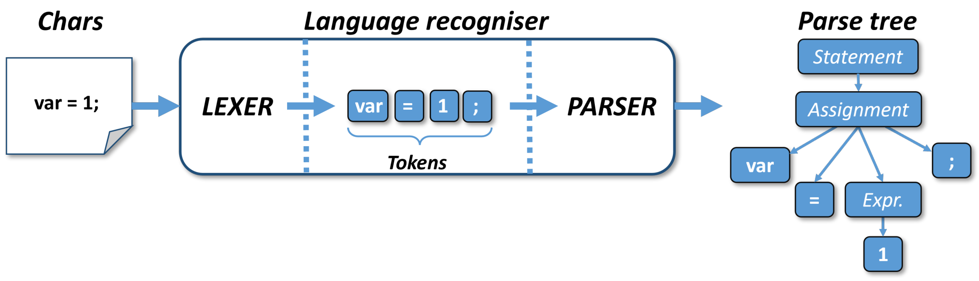


Figure A.3: AST generation using the language recogniser in ANTLR.

A.2.1 File Parsing

The first stage in creating a block model is parsing the source files. In this stage, we need to interpret the source code to understand the structure of the code. In order to assist in this task, we have used the ANother Tool for Language Recognition (ANTLR) v4 framework. This framework is an open-source tool which is able to parse text files according to a given grammar file [208].

The parsing process is done by two components, the *lexer* and *parser*, as shown in Figure A.3. The lexer splits the character stream into a series of meaningful tokens as defined in the provided grammar file [209]. These tokens are fed to the parser unit. The parser will then try to identify the parts of 'speech' according to the syntactic rules in the grammar file. The final result is an AST with the different tokens on its leaves and all rules on the intermediate nodes which were applied for each token [208].

A major advantage of the ANTLR framework is its language independence which allows us to switch to another programming language simply by loading the corresponding grammar file. Additionally, the automatic generated listener class enables us to walk through the tree and facilitates the integration of the ANTLR framework with our own.

In this extension, we implemented a file parsing unit for source files in the C-language. In order to generate this parsing unit, we need to compile an ANTLR-grammar file with the ANTLR-tool. Therefore, we used an existing grammar definition from the ANTLR repository [209] instead of creating one of our own from scratch. The syntax in this template is conform to the C11 specifications. However, some modifications were made to the grammar description to support compiler and flow-facts flags used in the test code of the TACLeBench [60] and to facilitate the generation of blocks in the next stage. For example, a distinction between the *true* and *false* branch of an *if*-statement is created, so we are able to connect each code body of the selection to the corresponding statement result, i.e. *true* or *false*.

A.2.2 Block Generator

The second step in the generation of the block tree is walking the generated AST from the file parser. As previously mentioned, ANTLR v4 implements a visitor pattern with the automatic generated listener class [208]. The pointer object called the walker starts at the root of the tree. It will then travel from a parent node to the next child node in order, i.e. left to right. This walk pattern is also referred to as a depth-first search [208]. Each intermediate node contains a rule that was applied to a token. When the walker enters or leaves a node, it will notify the framework and trigger a listener event. In order to make the connection between the ANTLR framework and the COBRA-HPA framework, we have subclassed the empty listener classes and overwritten the methods of interest. For example, when entering an iteration statement, a message will notify the Block Generator that an iteration statement is detected. All the following statements will be contained inside the body of the created iteration block. The walker will publish an exit message upon leaving the iteration statement to tell the Block Generator that the block is completed, and thus no further actions are required for this particular block.

The block tree used in the COBRA-HPA framework is generated on-the-fly when the AST is traversed by the walker. The result is a tree structure that is built out of blocks. Each block initially contains one instruction or statement of the source code. These blocks are arranged from left to right to their sequential order in the code. Instructions that reside inside a statement body are added as child blocks to the parent block. These instructions are characterised by their encapsulation in brackets (i.e. {}) in the C programming language.

All instructions are classified within one of the seven types of blocks. These block types are shown in the class diagram of Figure A.4, which are the following:

- **Program block**: root block which defines an entire source file with all of its methods;
- **Method block**: subcomponents of the program block that symbolises one program method;
- **Jump block**: groups all instructions apart from iteration/selection instructions which will break the program flow by changing the instruction pointer to another program instruction, e.g. return, goto and break statements;
- **Iteration block**: instructions that results in repeating a section of code depending on a stated condition, e.g. for, while and do . . . while statements;
- **Selection block**: includes those instructions where multiple program traces branch off the original trace. The given condition decides which trace will be followed, e.g. if . . . else and switch statements;
- **Case block**: these blocks indicate the different program traces originating from a selection statement. Case blocks hold the result for which the selection statement must satisfy in order to follow the underlying trace;
- **Basic block**: contains all remaining instructions that cannot alter the program flow.

In Figure A.5, a block tree is rendered from the code of Listing A.1 to illustrate the structure of a block tree. The attentive reader will notice that the first basic block in Figure A.5 contains not one but two instructions. This aggregation of two blocks is the result of another implemented feature in the COBRA-HPA framework. This feature is called the *Block Reduction Rule* system. In Section 5.1, we discussed the importance of block size in our hybrid approach. We need to find a perfect balance to keep the analysis reliable and computable. The default block tree generated by the block generator will always contain one instruction for each block. In order to change the block size, we

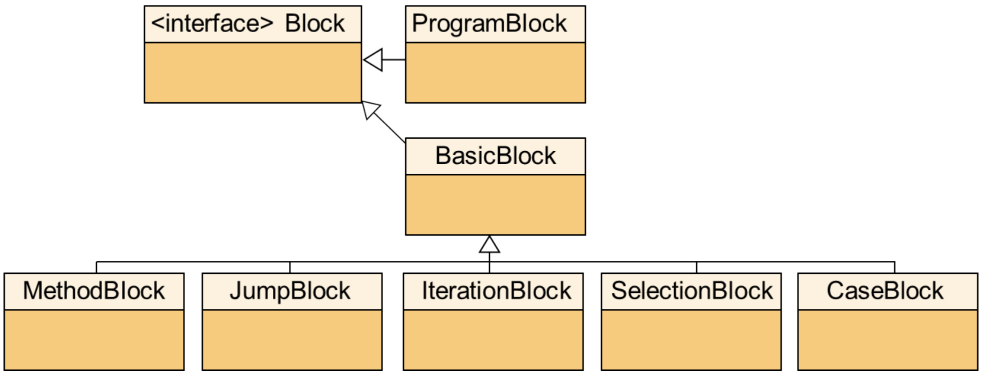


Figure A.4: Block type class diagram used for the block tree generation.

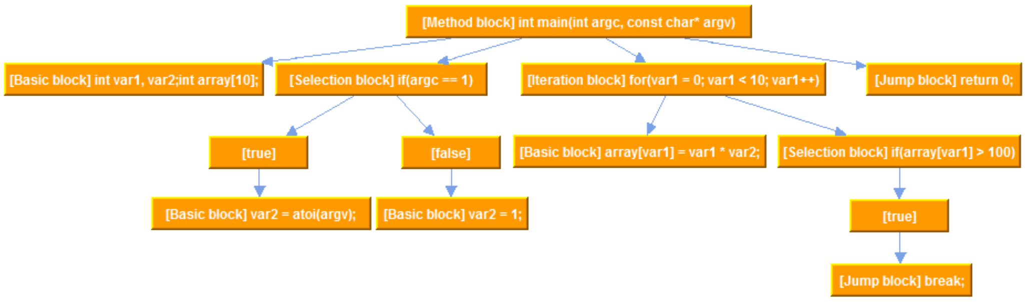

[Method block] int main(int argc, const char* argv)
[Basic block] int var1, var2;int array[10];
[Selection block] if(argc == 1)
[Iteration block] for(var1 = 0; var1 < 10; var1++)
[Jump block] return 0;
[true]
[false]
[Basic block] array[var1] = var1 * var2;
[Selection block] if(array[var1] > 100)
[Basic block] var2 = atoi(argv);
[Basic block] var2 = 1;
[true]
[Jump block] break;

Figure A.5: Block tree rendering with COBRA-HPA of the sample code in Listing A.1.

need to group the smaller blocks into bigger blocks. Therefore, the framework has the Block Reduction Rule system based on rules which allow the user to reduce the number of blocks by creating bigger blocks. When the block generator is finished, the developer has the option to apply a set of rules on the model. The current version of the system has two rules implemented, i.e. basic block reduction and abstraction reduction. The first rule searches for successive basic blocks and replace those by one bigger basic block. The second rule groups all blocks starting from a user defined depth, and thus creating an abstraction from the lower-level blocks. In the future, additional rules can be added to the system when more optimal reduction solutions are discovered in respect to the resulting block sizes, e.g. computational complexity of the static analysis, WCRC upper bound margin, etc.

```
int main(int argc, const char* argv) {
 int var1, var2;
 int array[10];

 if(argc == 1)
  var2 = atoi(argv);
 else
  var2 = 1;

 for(var1 = 0; var1 < 10; var1++) {
  array[var1] = var1 * var2;

```

```
    if(array[var1] > 100)
      break;
  }

  return 0;
}
```

Listing A.1: Sample program as an example input for the COBRA-HPA framework

A.3 Profiling Tools

The next step after creating a block model from a code file is profiling each block on the target platform. In the context of this thesis, it means performing time and energy measurements of the blocks to acquire the WCET and WCEC respectively. The COBRA-HPA framework provides supporting tools to simplify the profiling process and reduce the overall effort. The profiling section consists out of two major parts: *Profiling Generator* and *Profiling Assistant*.

A.3.1 Profiling Generator

The Profiling Generator is used in the preparatory stage of the block profiling. During this step, the tool generates the required code files from the block model. These code files are generated according to a syntax template that describes the syntax in order to create compilable code files for the profiling. Each generated file will include a *main* method to make the source file executable and a *testbench* method with the actual instructions of a block. Furthermore, interfaces are added for each profiling stage that allows the Profiling Assistant to implement platform specific code into the testbench, e.g. initialisation, register configurations, etc.

An additional challenge in creating a profiling testbench is to identify the code dependencies and declarations, as the code of a block is only a part of a larger application. Things get even more complicated when code dependencies are across multiple files. For example, when a variable is declared outside the scope of a block the Profiling Generator has to identify the missing type declaration first. Next, it must find the appropriate type in the provided application code. Last, the variable declaration needs to be added before executing the testbench code, so that the declaration will not interfere with the measurements of the block itself. Therefore, these declarations are placed in a dedicated function which is called before the testbench execution.

As the dependencies are declared, we need to initialise the input variables of the blocks. The goal is to generate input sets that include all possible input combinations during code execution in order to cover as many cases as possible. Furthermore, all non-occurring input sets need to be excluded to prevent too pessimistic results. Different strategies exist to generate valuable input sets to perform representative measurements [210]–[212]. The Profiling Generator performs three stages to derive values for the input sets. The first step is to perform a static analysis on the complete program to derive constant values or the state space of variables if possible. For example, the possible values of an input variable i is in the first place equal to the size of the data type. However, if we consider the context of the entire program this variable i can be defined as an iterator over a fixed interval in the parent block. This will eventually result in a significantly smaller value range of i.

The second step is to generate input sets for input variables which are annotated by the user. This step is optional but is provided if the state space could not be derived in the first step. This brings the opportunity to refine the final results if the possible values are known in advance, e.g. a temperature reading in the range of $-30\,^{\circ}\mathrm{C}$ and $60\,^{\circ}\mathrm{C}$.

The last step is creating the actual input sets for the measurements. The found value sets of the previous steps are used to parse an input variables file. Even a handful of input variables with bounded values can eventually lead to an immense list of tests that becomes unfeasible to test exhaustively. Therefore, a random set creator allows to generate input sets according to the needs of the user. The size of the test set is adjustable through a configuration file. The variables of which the state space is undefined up to this point, will receive a default state space that is defined in the configuration file for each data type.

A last important feature that the Profiling Generator takes care of is gathering all external import and header dependencies. A list of these dependencies is generated during the static analysis of the code file and added to the testbench. At this point, we generated testbench code files for each block. These testbench files are used in the next stage to perform the actual measurements on the target hardware platform.

A.3.2 Profiling Assistant

The next tool in the profiling chain is the Profiling Assistant. As the name of the tool already reveals, its purpose is to assist with the actual profiling of the hardware platform. Therefore, it provides an interface with an extensive range of possibilities to support as many hardware platforms as possible.

Based on the index file of the Profiling Generator, the tool will collect the testbench, source files and hardware configuration to compile a binary file for the target platform. Next, it will flash and restart the board to run the profiling procedure. The COBRA-HPA platform supports execution time and energy measurements. Depending on the target hardware capabilities, the assistant will select an appropriate strategy to measure the consumption usage, e.g., a tic-toc cycle counter. For energy measurements, we created an automated testbench that works together with the Profiling Assistant to perform energy measurements on the physical hardware. This testbench is presented in more detail in Section 5.3. The measurement strategy for execution time depends on the available features on the target hardware. Therefore, we try to select the most optimal, i.e. most accurate, methodology to perform these measurements. A discussion of this strategy is presented in Section 5.2.

A.4 Feature Selection

The Feature Selector module allows us to generate a formatted output file containing all features derived from the hybrid blocks in the project, as shown in Figure A.6. The selection of features allows us to describe the characteristics of the code in the blocks at a higher abstraction level. This makes it less complicated to develop and train an ML layer that is able to generalise the problem [169]. As stated in Section 6.1, we need to examine which features have a significant influence on the WCRC of code. Therefore, feature design requires adequate insight in the domain to compose a list of potentially relevant, quantifiable features. The next step is to assess the importance of the selected

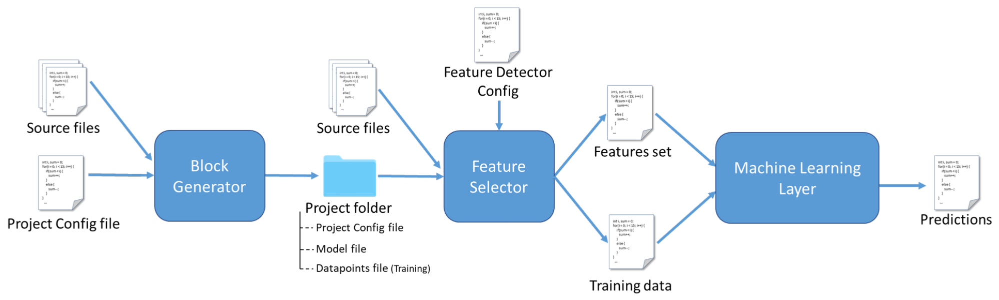


Figure A.6: Schematic overview of the Feature Selector.

features by checking for correlations between them [170], [171] and eventually training different types of ML methodologies to optimise the performance on the verification set.

After generating the hybrid blocks, a 'value' for each code attribute (i.e. feature) is obtained from these blocks. Extracting these features from source code is a time consuming and error-prone task. Additionally, to train an ML layer that is able to provide a 'solution', i.e. estimation of the WCRC, we need to apply a supervised learning strategy [169]. As a result, a large, annotated training set needs to be created. In this case, we need to determine the attributes of each feature and annotate it with the resulting WCRC.

The Feature Selector module is a new addition to the export tools of the COBRA-HPA chain. This tool allows us to generate a formatted output file containing all features derived from the hybrid blocks in the project, as shown in figure A.6. To perform a feature analysis, the tool requires three files in the project folder. The first input is the project configuration which contains basic properties, such as the locations of the source files and the programming language used. The second input required is the standardised model file describing the hybrid blocks generated with the Block Generator tool. The third optional input file contains a list of WCRC results to create annotated features in order to generate datasets for training and validation.

In order to easily generate and adjust the features for the large hybrid block collection, the Feature Selector has a flexible and modular design that enables us to describe a code feature in a readable configuration file. The detection of features is accomplished by defining and configuring basic detector modules which in turn are chainable to accommodate more complex feature detection rules. In the first prototype of this tool, there are three basic detector modules that already provides an extensive range of possibilities:

- **Token Count Detector** counts each occurrence of a set of basic tokens and maps the result to the desired feature;
- **Context Detector** classify if a certain rule is present in the context of the analysed basic block;
- **Collection Utilities Detector** performs basic set operations on the output of two or more detectors, e.g. accumulate results, union between sets, etc.

The functionality of the basic detectors is built on the open-source parsing framework, ANTLR v4. This tool provides the functionality to parse a text file according to a given grammar file [208]. The parsing process consists of two steps. The former step is performed by the lexer which splits the character stream into a series of meaningful tokens. The latter step parses the tokens stream and identifies the syntax of the code. The ANTLR framework is also part of the core of the Block Generator tool, as discussed in A.2.2. As it is programming language independent, the COBRA-HPA is extendible to other languages by loading the corresponding grammar file. In this first version, the feature detection is implemented for source code in the C-language.

In order to define a new detectable feature, a new detector chain needs to be added to the detector configuration file. A detector chain declaration consists of a detector class type and a unique name describing the desired feature. Additionally, the chained detectors are configurable according to key-value pairs. In Listing A.2, an example configuration is shown to define a feature ‘additions’ which will count all add operations in the code. In this example, a simple *Token Count Detector* is used to count the occurrence of add-related operation tokens. These tokens are extracted from the lexer token stream, as each meaningful token has a unique defined identifier. However, this naive approach of counting tokens is not always feasible.

```xml
<detector id="01" type="TokenCountDetector" featureKey="additions">
   <param name="token">'+'</param>
   <param name="token">'++'</param>
   <param name="token">'+='</param>
</detector>
```

Listing A.2: Definition of a simple addition feature detector with a Token Count Detector

A multiplication in the C-language is defined by the *-token (star). Nevertheless, defining a 'multiplications' feature is not as straightforward as the ‘additions’ feature, because the star-token in the C-language is also used to define and access pointers. Therefore, a detector chain needs to be created to intercept the pointer-related syntax of the star-token. At the start of the detector chain, two basic detectors are used, namely a *Token Count Detector* and a *Context Detector*. On the one hand, the Token Count Detector has the same functionality as the 'additions' example, however it will now count star-tokens. On the other hand, the Context Detector is a detector which works on the syntactic level of the code. The purpose of this detector is to find all pointer-related operations with a star-token. The next step in the detector chain is to exclude all found star-tokens from the first detector (e.g. token count) with the tokens returned by the second detector (e.g. context count). This differentiation action gets performed by the *Collection Utilities Detector*. As a result, this detector chain will now return us all multiplicative operations. Listing A.3 shows the configuration of this example detector chain.

```xml
<detector id="02" type="CollectionUtilitiesDetector"
         featureKey="multiplications">
   <param name="collectionAction">'diff'</param>
   <param name="detector">
      <detector id="03" type="TokenCountDetector"
            featureKey="multiplications">
         <param name="token">'*'</param>
         <param name="token">'*='</param>
      </detector>
   </param>
   <param name="detector">
      <detector id="04" type="ContextDetector"
            featureKey="multiplications">
         <param name="context">'Pointer'</param>
```

```
        </detector>
    </param>
</detector>
```

Listing A.3: Definition of a multiplication feature detector with a Token Count and Context Detector

When all attributes of the blocks are determined, a list of features is generated. These features are then ready to be exported to a formatted file. Currently, a Comma-Separated Values (CSV) exporter module is integrated in the tool which allows us to read the features in an ML framework, such as Scikit-Learn [173] or TensorFlow [19]. In addition, a WCRC-annotated version can be created when the corresponding results are provided. This presents the functionality to generate annotated features for training and validation sets.

A.5 Model Training and Execution

The estimation models used by the COBRA framework are created within specialised ML frameworks. Within the research conducted in this thesis, we currently use two Python frameworks to train and run the inference to obtain the upper bound estimations. The former framework used for regression models is *Scikit-Learn* [173]. The latter framework used for creating neural networks is *TensorFlow* [19]. Both frameworks use the features in the CSV-file of the Project folder as generated by the Feature Selector. The implementation for both frameworks is mostly identical. The first step is the data pre-processing. This step is the same for training and inference of the models. All datapoints are parsed within the framework as entries. During the training stage, all datapoints are labelled with the WCRC values to perform supervised learning. Any processing on the features, or input data is performed on the loaded datapoints. These operations could contain scaling, feature selection, normalisation, etc. depending on the model characteristics and the available features.

The next step when we are training the model, is to split the data into three different sets for training, testing and validation of the models. The dataset is split with a K-fold cross validation strategy. This allows us to create K random subsets from the initial dataset for our training, testing and validation to evaluate the performance of the model to generalise on the independent datasets and avoid overfitting. After generating the different datasets, we are ready to train each model by including the corresponding model class and call its training method, i.e. '*fit*' for regression models in Scikit-Learn or '*minimize*' on the gradient descent optimiser of the neural network in TensorFlow. In addition, some models require additional variables to 'tweak' the training process. These variables are called *hyperparameters*. Tuning these parameters is a computational expensive process as the models needs to be trained and evaluated for different combinations to find the optimal setting. Different techniques exist to optimise these parameters, such as grid search or stochastic search algorithms [169], [179]. In order to pick the best estimation model, we compare the performance of each model by calculating the error on the validation set. The data (e.g. weights, biases, function parameters, etc.) of the best model is stored to be used as estimator for upper bound predictions.

To further optimise this process, we are integrating AutoML techniques with the TPOT framework [179]. This allows us to automate the process of creating an optimised pipeline for processing the data, training setting the hyperparameters of the model. The implementation of this framework in our toolchain is still in development at the moment of writing.

Finally, we are able to load a trained model in the ML framework to provide an upper bound estimation of a given application. In this stage, the Feature Selector will not provide WCRC labels in its output as the estimation model will provide them. The model interference will be executed after the data pre-processing phase. The output of each data entry will be the upper bound prediction for the corresponding hybrid block. These predictions will be written back to the Project folder for the Analysis Wizard to obtain a final WCRC estimation for the task.

A.6 Exporting Results

Finally, we provide a set of tools to export the results to other formats or third-party tools for further research. Currently, we have two extensions that enable us to export the results. The Analysis Wizard provides an upper bound analysis on the WCRC and reports this to the user for the resource-aware scheduler. The UPPAAL Model Checker provides an integration with the third-party tool for the creation of timed automata.

A.6.1 Analysis Wizard

By performing the measurements on the target hardware, we have completed the first part of the hybrid analysis, i.e. measurement-based analysis layer. The next step consists of evaluating the acquired results from all blocks and statically combining them according to their interactions in the CFG. As we are interested in the WCRC of the application, we need to find the path with the highest cost. The Analysis Wizard assists in estimating this path and providing an upper bound.

Firstly, the block interactions in the program are modelled with the program flow analysis. The consecutive execution of the blocks is equal to a sequential flow through the program. In this phase, we need to identify all feasible traces, or paths of the application. The flow analysis provides information, such as iteration counts, dependencies between selection statements, function calls, etc. in order to characterise the different paths. Therefore, these hints are called *flow facts*, as they acquire insight of the CFG of the application [83]. Flow facts can be derived implicitly from the program's structure and its semantics, or by annotating the source code [83] [164]. The latter is required for flow facts which cannot be determined statically, such as recursion depths, dynamic calls and jumps, infeasible paths, loop iteration counts, etc. It is important to note that user annotations have to consider all possible input sets in the system in order to obtain sound results.

Secondly, we need to calculate the worst-case path in order to obtain the upper bound of the resource consumption. This is achieved by combining the measurements results from the previous steps and the flow facts of the program flow analysis. Currently, three major techniques are commonly used to compute the upper bound, i.e. *path-based* [162], *tree-based* [53] and *IPET* [66], as presented in Section 5.4.

The upper bound estimation is calculated using an IPET technique on the CFG with the GLPK [166] package. The final results are exported as a list of tasks that can be provided to a resource-aware scheduler [116], [213].

A.6.2 Timed Automata with UPPAAL

In addition to the hybrid methodology discussed in Chapter 5, our research group explores other methodologies to determine the WCET. One of these studies is about determining the WCET with probabilistic analysis [214], [215]. For this research, the theory of timed automata is applied. In order to support the creation of models for the experiments, we implemented a feature to build these automata automatically from source code.

A timed automaton is a finite-state machine extended with the notion of time [216]. The model contains clock variables which are in synchronisation with each other. Therefore, the transitions in this model can also be guarded by comparing clock variables with numeric values.

For modelling the timed automaton, we use the model checker UPPAAL [216]. This toolbox developed by Uppsala University and Aalborg University is designed to verify systems which can be represented as a network of timed automata [216]. The purpose of using this tool is to model the program flow into a timed automaton. By implementing the UPPAAL model exporter, we are able to create timed automata.

First of all, we start by creating a timed automaton data model. Therefore, we translate our hybrid block model to the syntax of timed automata. Each hybrid block is directly mapped to a node in the automaton. The next step is generating the necessary links between the nodes. These links are the transitions which represent the program traces that can be traversed during program execution. In order to create those links, we need to perform a program flow analysis to determine the different links or traces between the nodes. As previously discussed in Section A.2.2, we define seven types of blocks in our hybrid model. Each of these types refer to a specific structure inside the control flow of the program, such as iterating and selecting a program trace. To determine which links that need to be created, we have to interpret the statements inside the blocks with the exception of generic basic blocks and case blocks. Each specific statement will eventually result in another program trace. For example, a while-loop will only iterate over the code as long as the condition is true. Whereas a do . . . while-loop will always execute the code at least once regardless of the result of the condition.

Second, the generated automaton model is exported into an XML project file for UPPAAL. The challenge in creating this file is to comply to the correct syntax defined by the UPPAAL framework and to deliver a proper and clear layout of the model in the Graphical User Interface (GUI)-based tool [216], [217].

In Figure A.7, an automaton is generated in UPPAAL for the example code given in Listing A.1. Each node in the model has a name corresponding to the line numbers in the original source file. The associated code is included as a comment to each node. For a case block, i.e. *true/false* or cases in a switch instruction, the statement is added to the guard of the link that refers to the matching program trace.

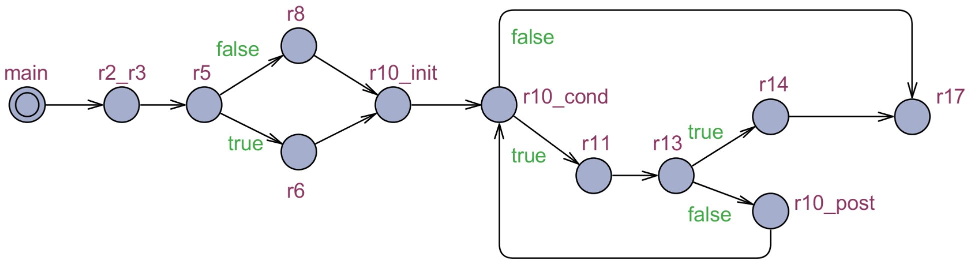


Figure A.7: Exported timed automaton in UPPAAL generated from the sample code of Listing A.1.

Appendix B

Resource-Aware AI

The research conducted within this thesis fits within the broad research field of RAAI. Within our team, we focus on the optimal distribution of AI networks on different scales in distributed constrained environments. These environments consist out of a wide variety of hardware platforms and network infrastructures, ranging from in-vehicle networks of small embedded computers for autonomous driving to large-scale AI running over high-end cloud infrastructure up to the edge nodes of the network and its connected network devices, e.g. sensors, smartphones, etc. The optimal distribution of AI (sub-)graphs will depend on different KPIs which are bound to hard requirements of the system or the available resources of the hardware, such as hard real-time behaviour for CPS, available energy capacity of mobile devices, memory availability, etc. With these KPIs, the (sub-)graphs are dynamically distributed depending on the current needs over the available hardware infrastructure by an orchestration layer with objective-based optimisation. In order to determine an optimal solution, the behaviour of the trained AI models needs to be defined. For resource-constrained systems, we are interested in the worst-case behaviour and utilisation of these resources. The worst-case behaviour is not only depending on the AI graph composition, but also the used hardware platform and communication infrastructure. Therefore, a WCRC model is created for each subcomponent of the distributed AI on different target hardware configurations. These resulting WCRC models provide the necessary insight to optimise the distribution of the AI according to the resource requirements, or further optimising the internal architecture of the individual (sub-)graphs.

The RAAI optimisation approach, that we are developing, is illustrated in the schematic overview of Figure B.1. This methodology consists of two major optimisation strategies. Offline optimisation is performed before deployment of the distributed AI model. The online optimisation on the other hand is continuously performed while the models are actively running in production. Each of the optimisation steps in Figure B.1 flow require inputs as optimisation goals or KPIs. These inputs are the actual load or tasks the system needs to run, the (hard) resource constraints, environment or context-aware knowledge, and WCRC, real-time harvesting and consumption predictions. A summary for each of these components is presented in the following sections.

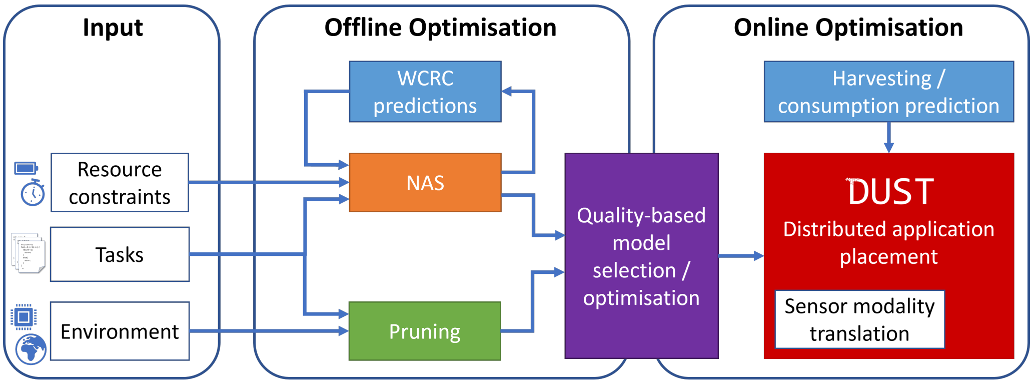

Input
Offline Optimisation
Online Optimisation
Resource constraints
Tasks
Environment
WCRC predictions
NAS
Pruning
Quality-based model selection / optimisation
Harvesting / consumption prediction
DUST
Distributed application placement
Sensor modality translation

Figure B.1: Schematic overview of the RAAI research.

B.1 Neural Architecture Search

The aim for NAS is to automate the design process of neural networks in such a way that both resource consumption and task performance are optimised according to the designer's preference. Being able to accurately predict the WCRC of a neural network is a vital part of this design process, since it allows NAS systems to reduce the worst-case performance of networks to meet device constraints, while retaining acceptable task performance. When utilising reinforcement learning-based NAS approaches, it is not uncommon to consider 1000's of architectures during the search process. This poses a challenge for WCRC determination methods, since the amount of time that is acceptable to be spent determining the WCRC of any given network is limited, to keep the total search time reasonable. This clearly outlines the need for WCRC determination methods that fall outside the traditional static and measurement-based analysis methodologies, since both may take prohibitively long to determine WCRC behaviour. This is also where an ML-based WCRC prediction technique shows its value, since querying an ML model takes only a few milliseconds, compared to minutes, hours or even days for static or measurement-based approaches [199].

B.2 Knowledge-Based Pruning

In the state-of-practice, IoT platforms make complex decisions based on sensor information in centralised data centres. Each sensor transmits its observations to the cloud after which a decision or action is send to actuators. In time-critical applications, the latency imposed by this communication could lead to disastrous consequences. Therefore, decisions should be made on the edge devices themselves. Within the context of context-aware and RAAI, we want to develop edge inference systems that dynamically reconfigure itself to adapt to its dynamic environment and resources constraints. However, with the ever-increasing complexity and capabilities of ML algorithms, this push to the edge leads to new challenges due to constraints imposed by the limited available resources (e.g. computing power, memory, energy) on the edge devices, and thus the WCRC plays an important part in the optimisation process. A form of compression is required in order to run SOTA convolutional neural networks on edge devices [218].

In this work, we focus on extending the current SOTA on neural network compression by incorporating knowledge-based pruning methods. The knowledge-based component first determines the locations of specific task-related knowledge in the network and uses this to guide pruning algorithms. As a result, we are able to remove unnecessary concepts from the network and make them adjustable to environmental characteristics and hardware constraints. For tasks in specific environments, it might be beneficial to reduce the accuracy of certain outputs in favour of resource gain. For example, the classification of certain traffic sign types, such as parking signs, are unnecessary to detect on highways, and thus less accuracy is required than in a city centre. Based on these requirements, we want to selectively prune these networks by locating specific task-related concepts. By removing sections of the network, we expect to achieve higher compression ratios compared to the SOTA.

In a first stage, the concepts will be focusing on a high-level. A high-level concept revolves on locating those paths throughout the neural networks that are responsible for the outputs themselves. Therefore, we generate a sort of heatmap of the network that corresponds to the influence each node has on the outputs. The next stage extends this localisation to lower levels by identifying semantic concepts in the network. These semantic concepts are for example textures, colours, etc. By being able to identify these very low-level concepts, we will have more control over those concepts we want to remove. In the last stage, we want to generalise to other network architectures and applications. The two former stages are focusing on computer vision applications, for which the outputs and semantic concepts are human interpretable. For these stages, many methods from the field of explainable AI exist. In order to generalise to different applications, we need to explore unsupervised clustering techniques to identify relationships between the clusters and outputs of the network. These relationships will allow us to remove irrelevant parts of the network.

B.3 Quality-Based Model Selection and Optimisation

As ML has made its introduction recently in CPS systems, such as robotics, autonomous driving, and inland vessels, it is traditionally trained and validated on a set of datapoints for a given context. It has proven to be very effective to learn and operate in different circumstances. However, problems arise when the algorithm is operating in a context that has not been seen before in datasets or environments. The algorithm then would make predictions that could be ambiguous, and thus result in a cascade of problems. One potential solution is to include these discrepancies into the learning process, so that the ML algorithm learns to generalise these unseen scenarios. Nevertheless, the complexity increases tremendously when including more discrepancies to generalise for.

Another solution is to select the most suitable models that are the most performant in a particular context for a given AI architecture. Therefore, the necessary quality requirements need to be met at the output. This selection is based on how well the performance of the networks are regarding the noise or error present on their inputs.

The data quality-based model selections are employed for recommending the most suitable, reduced neural networks. Within the research project, different pruned versions of a particular neural network are created, which all are optimised towards different objectives. For example, one version of a vehicle detection network could on the one hand be optimised to recognise all sorts of vehicles, but with redundant neurons in the network being reduced. A second version on the other hand could only focus on recognising trucks and cars in particular. The knowledge of this network to recognise other vehicle types, such as boats and aeroplanes, will be highly reduced. As a result, different versions of a particular network imply different internal structures, and therefore also different output would be generated when noise and errors are present in the input. The goal of this research is to design a multi-objective optimisation process that is able to select the most suitable version of a neural network that complies to a selection of prioritised resource-aware objectives. An optimisation objective for the example could be that the requested network should be able to recognise cars and trucks with a high accuracy when Gaussian noise is present, but also should be smaller than 5 MBytes in size to fit within the memory and draw power within certain limits. These last constrains require insight in the WCRC behaviour of these networks in order to perform these multi-objective optimisations.

B.4 Distributed Application Placement

A core component in the online optimisation segment of the framework is ensuring that the ML models get assigned to their most efficient placements by applying a software placement optimisation algorithm. The algorithm takes into account the respective available device and network resources and core metrics of the application, such as WCET and worst-case throughput, among other WCRC metrics. Based on the knowledge of the application and the network, this algorithm allocates the models a specific device to optimise multiple competing objectives, such as the global WCET and the load over the network [3].

In such scenarios, WCRCs are fundamental components to use both as constraints as objectives. The WCRC is required to be below an application-specific threshold, as exceeding this threshold will cause the application to misbehave or the Quality of Experience (QoE) to degrade. Once this constraint is met, the WCRC can be used as an optimisation goal, further improving the QoE. In order to apply this technique in a physical environment, we have developed a framework that facilitates this optimised placement in dynamic heterogeneous networks, which is presented in Section B.6.

B.5 Sensor Modality Translation

Autonomous robotics have made significant impacts in many domains, from manufacturing and logistics applications [219], predictive maintenance [220], [221] to security and surveillance [222], [223]. The navigation for these autonomous agents is performed by processing measurements of perception sensors to understand and anticipate on their environment. As sensor data is used to make navigational decisions, it is evident that the accuracy and reliability of the output should be as high as possible for the safety of the robot and its surroundings. The data selection, i.e. which sensor data to use, combined with the processing methods, substantially impacts the safety and performance that is achievable. The current SOTA systems mostly rely on optical sensors, such as cameras and time-of-flight sensors (e.g. Light Detection And Ranging (LiDAR)), for their environment perception. Many of the existing algorithms and methods are specifically designed for these optical sensors that achieve impressive levels of autonomy and intelligence. Nevertheless, the performance of the methods is heavily impacted when the optical sensing conditions are lowered (e.g. dust, fog, rain, etc.) [224]. In addition to the dependency of the sensing conditions, these optical SOTA perception systems are expensive and have an extremely high computational cost for safety-critical applications. Computational resources are limited, certainly from a financial point of view. Therefore, processing all sensors streams is not desirable in every situation. Resource-constrained systems combined with mainly optical sensing suites limits the application potential these autonomous systems have.

This research addresses the problems and extends the current capabilities of existing autonomous agents. We aim to develop techniques for dynamic heterogeneous sensor signal fusion in order to create an internal representation of the appropriate complexity, which we named *Goldilocks' sensor fusion*, to support novel end-to-end approaches for resource and context-aware navigation. By creating a shared latent representation for a multi-modal sensor setup, e.g. consisting of LiDAR, sonar and radar; the capabilities of the autonomous agents can be significantly extended. Our approach focuses on the flexibility of the sensor data used for navigational decision making. To this extend, each sensor is used separately to update a shared latent environment representation that allows the agent to select an optimal sensing modality given a certain context; and deals with the volatility of its resources. This latent environment representation is used in further processing and decision making by converting or decoding to the desired modality. For example, decoding the latent representation to a point cloud representation of the environment that is applicable to off-the-shelf algorithms designed for this type of data.

The dynamic sensor data fusion and selection approach allows us to easily extend the capabilities of new and existing autonomous agents, as the algorithms designed for current SOTA sensors is still deployable, even in hard sensing conditions.

B.6 DUST Framework

The Distributed Uniform Streaming framework (DUST) framework is a middleware platform developed to create software components that are distributable across heterogeneous networks [225]. These components or modules are able to run on different network nodes, ranging from the central cloud infrastructure up to the network edge nodes and devices (e.g. edge routers, sensors, smartphones, etc.). However, creating such intricate systems requires a good strategy to route all data between each component. In the context of distributed computing, all distributed nodes produce high loads of data on a wide scale that needs to be processed. According to Intel, a single smart car would produce around 4 terabytes of data each day while driving [226]. Performing all computation centralised in the cloud is not feasible as it would result in a considerable strain on the network infrastructure. A better approach is to move the processing of the data to the edges of the network as close as possible to the devices. This approach is called Fog Computing [3]. Creating a large-scale distributed application in a dynamically changing hybrid environment is a tremendous challenge to implement. Therefore, the DUST framework is created to help engineers in developing and examining these systems. In the state-of-practice, we identify a gap in connecting different data from heterogeneous sources to components. With the DUST framework, we want to cover the gap between different data sources or 'data producers' and the systems and services that require this data, which we call 'data consumers'. Data plays a key role in IoT and large-scale agent-based simulations [227] for these revolutionary (distributed) applications.

The DUST framework is developed by the IDLab research group as a platform to build and test distributed computing applications with live migration possibilities [225]. The migration feature provides a mechanic to rearrange the placement of the application modules across the different nodes of network. Therefore, the system is able to adapt to its dynamic environment (i.e. changing network infrastructure, loss of nodes, etc.) to obtain an optimal distribution with respect to a defined cost function [228]. For example, an air pollution monitoring solution using compact air quality sensors that are installed throughout a city. Each sensor module has limited resources available, e.g. battery capacity, computational processing, etc. For small scale solutions, it may seem feasible to send all data to the cloud for processing. However, the amount of data, and thus the transmission cost over the network increases when the number of sensors grows. When small pre-computations are performed on the devices itself, the network saturation will drop as the amount of data traffic reduces significantly. Nevertheless, the impact on the battery life is considerably high as the additional computations require more energy, and therefore reduces the battery lifetime of the remote nodes. By apply a cost function on the block placement strategy, the DUST framework is able to distribute the load over the available resources with respect to its environmental context, for example batch computation of close located nodes on the edges of the network (i.e. fog computing), and restrictions, such as the remaining battery power, network bandwidth, etc. Additionally, each node and link of the network is monitored by the framework. This allows the middleware to create and maintain a most optimal and efficient distribution of the application modules.

A first requirement of the application is its ability to divide the application logic into smaller components. We refer to these application components as *modules* or *blocks*. Each module is connected to others with a *link* in the application graph. The generic architecture of a DUST applications with its modules is illustrated in Figure B.2. The framework can be divided in four components based on this figure:

- Configuration of the module based on a file;
- Discovery of other (dynamic) modules;
- Dynamic configurable channel stacks for data streams;
- Application logic and interfacing.

A DUST application is a collection of modules that work together in a distributed environment by sending predefined messages to each other over data links. The development of such an application consists of two major parts. On the one hand, the development of DUST modules that contain the business logic or algorithms of the application. Each module is built upon the DUST block interface of the DUST-Core library, or native interface bindings for other programming languages. The DUST-Core handles all necessary operations and configurations to transmit and receive messages, monitor the state of the system and perform migration and recovery of modules with minimal effort of the developer. By separating each ‘task’ or concern into its own module, a collection of modular services is obtained that is reusable in other applications when the right level of abstraction is applied.

On the other hand, all DUST modules are connected and working together to obtain emergent behaviour to solve domain-specific problems. This interconnected graph of DUST modules is a DUST application. To acquire the desired behaviour, the connections and interactions between each module is described in a configuration file. This file

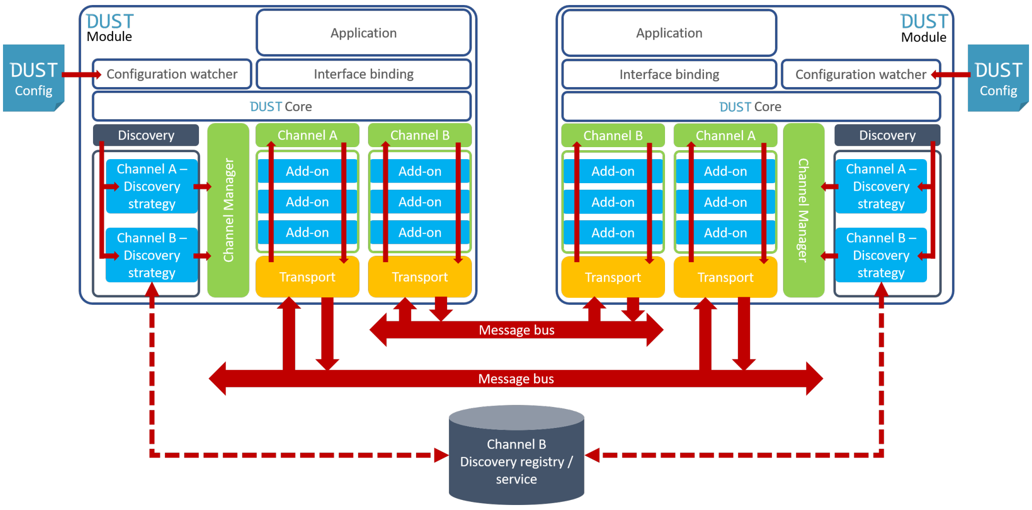


Figure B.2: Schematic overview of the DUST framework.

contains the preferred add-on stack configuration and transport configuration (i.e. publisher or subscriber). Additional static parameters in the file are used by the DUST-Core to configure itself. Nevertheless, the framework is able to reload a new configuration at runtime. This feature allows us to dynamically adapt the system's configuration to the state of the network (e.g. network load, battery capacity of mobile nodes, etc.). To achieve this functionality, the DUST framework uses an *orchestrator* node to monitor the state parameters of the system to determine the most optimal module configuration across the network. The orchestrator will update the configuration when modules are migrated to other network nodes. An additional orchestration file is required to hint the orchestrator about the behaviour and characteristics of each DUST module (e.g. WCET, energy consumption, etc.). Determining these characteristics is accomplished by using behaviour and resource analysis tools, such as the COBRA framework [26], [116].

In the following subsections, the functionality and implementation for each of the four basic components of the framework are explained in more details.

B.6.1 Configuration

The configuration of DUST modules is performed through the use of a file-based key-value mapping. This file is called a *DUST config* as shown in Figure B.2. The structure of a DUST config is formatted as JSON. This provides the flexibility of using a standardised format to communicate all configuration parameters across heterogeneous platforms and being accessible for humans to interpreter and update the file content.

The configuration file is the mandatory interface to adjust any settings in the DUST module. The content of the file is automatically loaded by the *Configuration watcher* during the initialisation phase. A config file may contain the configuration definitions for one or more modules of an application. It consists of two main sections. The first section is the collection of templates. A template describes assets that are shared among different modules. This includes for example the prototype definitions of links between two or more modules, or the configuration of channel stacks between complementary publishers and subscribers which will be discussed in Section B.6.3. The second section are the individual configurations of each module separately. Each module has a Unique Identifier (UID) that is used to indicate its corresponding settings in the DUST config. These individual module configurations combine the shared templates with module specific parameters. An example of a configuration file is provided in Listing B.1.

```json
{
  "_templates": {
    "batched-template": {
      "addons": [
        {
          "type": "batch",
          "batch_size": 2
        }
      ],
      "transport": {
        "type": "socket",
        "protocol": "tcp",
        "address": "127.0.0.1",
        "port": 3000
      }
    }
  },
  "publisher-module": {
    "links": {
      "channels": {
        "publish-tcp": {
          "_template": "batched-template",
          "transport": {
            "publish": true,
            "host_server": false
          }
        }
      }
    }
  },
  "subscriber-module": {
    "links": {
      "channels": {
        "subscribe-tcp": {
          "_template": "batched-template",
          "transport": {
            "publish": false,
            "host_server": true
          }
        }
      }
    }
  }
}
```

Listing B.1: Sample configuration file for a DUST application

After parsing the configuration file, the parameter sets are passed to the *Discovery* (Section B.6.2) and *Channel Managers* (Section B.6.3) to end the initialisation phase. Nonetheless, the Configuration watcher stays active in order to track changes to the config file. When an update is made to the file, the Configuration watcher will compare the new file with the previously loaded version to create a differential set list. This list will be used to notify the discovery and channel managers to update the current configuration with the new settings. The DUST config is therefore used to push new configurations to the modules when better placements are discovered.

B.6.2 Discovery

Dynamic module discovery is a new feature that has been introduced in the revised release of the DUST framework. In previous versions of the DUST-core, all configurations needed to be statically defined in the DUST config. The Configuration watcher would parse all parameter sets and instruct the Channel manager to build the requested channel stacks, as discussed in Section B.6.3. This approach suffices for applications with a finite number of predefined DUST modules, as each module requires a unique entry in the configuration file. However, applications in a dynamic environment with an undefined number of node-bounded modules that may enter and leave the network at any given time are impossible to implement due to the static nature of the configuration file. For example, cars that communicate with the roadside infrastructure have a high dynamic behaviour as they are constantly switching between infrastructures based on their location. If each car requires a dedicated DUST module on board, it becomes practically impossible to declare all cars in the world as each entity will require its own unique configuration in the file. Additionally, updating the DUST config for every new car that is produced is infeasible to sustain. Therefore, another approach is required to maintain a unique context to each block while providing the flexibility to add/remove modules at runtime.

In the roadside communication use case with car-bounded modules, a practical issue arises that it is not feasible to define each and every car on this planet in the static approach of the configuration file. The problem of those car-bounded modules can be described as unique blocks with identical logic of which there are zero, one or more instances present in the network at any given time. Starting from that insight, we extended on the template construct used for shared definitions in the DUST config, as explained in Section B.6.1. A module template describes the generic input properties of that module in the configuration file. It is not bound to one specific module in the application, but points to a group of modules with the same application logic on top. The exact number of dynamic modules in a group is undefined, and thus depends on the number of connected entities at runtime. Nevertheless, the input parameters of a module may heavily depend on its physical placement in the network and/or locality in the environment.

The dynamic modules need to identify their surrounding and location within the network/application in order to configure itself. Therefore, we introduced discovery functionality into the DUST framework. The discovery feature is responsible to advertise newly connected modules to the application and remove all references to disconnected modules. The DUST framework does not implement one fixed discovery strategy but provides an interface to dynamically specify a discovery strategy for each channel in the application graph. The developer is able to integrate multiple strategies within one DUST application. It is possible to select the most appropriate strategy based on the features and limitations of the underlying physical network. For example, one module may contain three channels over different physical links (e.g. local Ethernet, Long Range (LoRa) protocol, Wi-Fi) to connect to dynamic modules. Based on the physical implementation of the links, a distinct strategy will be required for each one of them. The application developer is not limited in using just one approach or obligated to create one multipurpose solution that fits all cases. For example, discovery based on multi-casts messages is a feasible approach in local Ethernet networks. However, multi-cast messages are never forwarded over L3 network routers, and thus not applicable for nodes that are connected over the Internet.

Each discovery strategy is contained within its own package that is dynamically loaded in the framework when it is requested by the DUST config. Developers are able to integrate their own proprietary procedure or third-party libraries as a discovery strategy, such as the Simple Participant Discovery Protocol (SPDP) of the Data Distribution Service (DDS) protocol [229] or a broker-based discovery approach [230].

When a discovery strategy is requested for a channel, the corresponding discovery module is insert between the Configuration watcher and the Channel manager. A processing pipeline for the configuration object is created. The interfacing of the discovery module is based on parsing the configuration as input and substitute placeholder values in the configuration with static parameters as output. Listing B.2 shows an example of a discovery-based channel configuration.

```
"discovery": [{
    "type": "multicast",
    "id": "mldisc",
    "port": 2578
}],
"_templates": {
    "channel_template": {
    "addons": [
    {
        "type": "batch",
        "batch_size": $<mldisc:BATCH_SIZE>
    }],
    "transport": {
        "type": "socket",
        "protocol": "tcp",
        "address": "$<mldisc:IP>",
        "port": $<mldisc:PORT>
    }
}
...
```

Listing B.2: Snippet of a module template with discovery in a DUST config

In the discovery section of Listing B.2, each element of the array is a discovery module that have all a ‘type’ set to indicate the discovery strategy and a unique ‘id’ field as a reference. All input parameters for a discovery module are presented as a map of key-values which are used as inputs variables. When a new DUST module has advertised itself in the network, the discovery module will generate a new static configured channel description that is passed on to the Channel manager. This channel description is constructed based on the templates and placeholder keys. These placeholders are configuration expression that the discovery uses to fill in the connection specific parameters of the advertising DUST module. The syntax of those expressions is: $<ID_MODULE:VARIABLE_NAME>. The first value indicates the ‘id’ of the discovery module. The second value specifies the output value that needs to be substituted by the discovery module.

B.6.3 Channels

The core task of the DUST framework is handling the streaming of data between the modules. All modules are connected in the application graph with *links*. A link is a basic interpretation of a data stream in the framework. In the current DUST release, these links are implemented as data *channels*. A channel is the representation of a topic-based data stream that connects two or more modules with a publish-subscribe paradigm [231]. Each channel contains a topic with a predefined message type. Multiple publishers and/or subscribers are able to connect on the same channel (i.e. many-to-many). As channels are topic-based, it is possible to define multiple channels between the same modules to have a clear separation of concerns between data streams.

As illustrated in Figure B.2, a channel is a stack of blocks that consists of three parts: an interface for the application layer, a chain of configurable add-ons and a transport block on the bottom. The application interface accepts serialised byte streams that are send as a data stream over the channel. On the receiving end of the channel, the data stream is offered back to the application layer as the serialised byte stream. The format of the byte stream is fully transparent for the DUST framework. The interface of the channel makes the underlying stack fully transparent for the application layer, as the channel guarantees the delivery of the original message to all subscribers of the channel.

The second layer of the channel stack is a fully configurable chain of add-on blocks. An add-on block is an interchangeable component that performs a basic operation on the byte stream. These operations provide data enhancing and transmission optimisations, such as bandwidth optimisations, security, error detection and correction, etc. Each block implements an encoding and decoding data stream. The add-on stack of interconnected channels are always identical copies of each other. As data flows top-down the stack when transmitted, each add-on will 'encrypt' the stream with its operation. On the receiving stack, the data will traverse the chain bottom-up. The stream will now be 'decrypted' by reversing each operation until the original data is obtained. The stack configuration is therefore linked to a channel configuration and not to a module. Each add-on is a dynamic component that is loaded into memory by the Channel manager when it is requested by the DUST config.

The end layer of the channel stack is the transport block. These blocks are functionally the same as an add-on. However, they only have one pair of in-and outputs that are used to terminate the chain of add-ons. Transport blocks are dynamically loaded libraries that contain transport protocol logic. The DUST framework is not bounded to selective transport layers, as new protocols can be integrated as new transport blocks. A wide variety of transport blocks are implemented, including, but not limited to, MQTT, ZeroMQ, DDS and Transmission Control Protocol (TCP)/User Datagram Protocol (UDP) sockets. The flexibility of the channel stacks makes it possible to adjust them based on the environmental context. For example, it may be feasible to swap from sockets to a shared memory transport block when the linked modules are placed on the same node to optimise the network and CPU usage.

All channels are managed by the Channel manager. This component is responsible to load all dynamic modules, i.e. add-ons and transmission blocks, into memory. Channels are created and destructed at runtime when an updated configuration from the Configuration watcher or Discovery is provided to the manager.

B.6.4 Application Logic

The last component is the application layer. This layer contains the application logic of the module itself on top of the DUST-Core interfaces. The DUST framework is written in C++ to provide optimised performance for low latency data streaming. The code base is compatible with different OS, such as Linux, Windows and MacOS. Additionally, it can be compiled for execution on bare-metal embedded devices without OS. This increases the deployability on a broad range of devices in the heterogeneous network, such as embedded sensor nodes. Furthermore, the DUST framework provides native interfaces to other programming languages, e.g. Java, Python, etc., to provide a language-agnostic platform to develop on.

A minimalistic '*light*' version of the Core is in development for power efficient and/or micro controller devices where computational overhead of the framework should be as low as possible. For example, battery-less devices use small (super)capacitors to store a limited amount of energy from an energy harvester [232]. Therefore, energy is a scarce resource for those systems.

The Core library of DUST is provided as a package with a concise set of interfaces the application layer has to invoke during the application lifecycle. This provides the module developer the responsibility and liberty to have full control over the program flow. However, an optional framework package for the DUST-Core is developed to invert the control with a best-practice implementation of the Core library. The framework will manage the general program flow and provide abstract functions for different events in the lifecycle, i.e. initialisation, program execution loop, message received, shutdown, etc. The developer has to fill in the right logic that needs to be executed for each event without writing any lifecycle logic itself.

The DUST framework provides full transparency of the underlying network and communication stack as discussed in Section B.6.3. The application layer has a set of channel end points to send and receive messages to and from other modules. The channel stack guarantees the delivery of each message to other static modules (or connected dynamic modules). As a result, the developer no longer has to consider the implementation or configuration of the transport layer. The transport layer and channel stacks are able to change without influencing the module's operations. This full transparency allows the DUST framework to migrate modules to other network nodes without application logic noticing it.

Appendix C

Publications

C.1 First Author

1. Conference Papers (P1)

    a) Huybrechts T., De Bock Y., Li H., Hellinckx P. (2017) Hybrid Approach on Cache Aware Real-Time Scheduling for Multi-Core Systems. In: Xhafa F., Barolli L., Amato F. (eds) Advances on P2P, Parallel, Grid, Cloud and Internet Computing. 3PGCIC 2016. Lecture Notes on Data Engineering and Communications Technologies, vol 1. Springer, Cham. DOI: 10.1007/978-3-319-49109-7_73

    b) Huybrechts T., Vanommeslaeghe Y., Blontrock D., Van Barel G., Hellinckx P. (2018) Automatic Reverse Engineering of CAN Bus Data Using Machine Learning Techniques. In: Xhafa F., Caballé S., Barolli L. (eds) Advances on P2P, Parallel, Grid, Cloud and Internet Computing. 3PGCIC 2017. Lecture Notes on Data Engineering and Communications Technologies, vol 13. Springer, Cham. DOI: 10.1007/978-3-319-69835-9_71

    c) Huybrechts T., Mercelis S., Hellinckx P. (2018) A New Hybrid Approach on WCET Analysis for Real-Time Systems Using Machine Learning. In: Brandner F. (eds) OpenAccess Series in Informatics (OASIcs). WCET 2018. 18th International Workshop on Worst-Case Execution Time Analysis, vol 63. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. DOI: 10.4230/OASICS.WCET.2018.5

    d) Huybrechts T., Cassimon T., Mercelis S., Hellinckx P. (2019) Introduction of Deep Neural Network in Hybrid WCET Analysis. In: Xhafa F., Leu FY., Ficco M., Yang CT. (eds) Advances on P2P, Parallel, Grid, Cloud and Internet Computing. 3PGCIC 2018. Lecture Notes on Data Engineering and Communications Technologies, vol 24. Springer, Cham. DOI: 10.1007/978-3-030-02607-3_38

    e) Huybrechts T., Vanneste S., Eyckerman R., de Hoog J., Mercelis S., Hellinckx P. (2020) DUST Initializr - CAD Drawing Platform for Designing Modules and Applications in the DUST Framework. In: Barolli L., Hellinckx P., Natwichai J. (eds) Advances on P2P, Parallel, Grid, Cloud and Internet Computing.

3PGCIC 2019. Lecture Notes in Networks and Systems, vol 96. Springer, Cham. DOI: 10.1007/978-3-030-33509-0_62

f) Huybrechts T., Mercelis S., Hellinckx P. (2021) A Survey on the Software and Hardware-Based Influences on the Worst-Case Execution Time. In: Barolli L., Takizawa M., Yoshihisa T., Amato F., Ikeda M. (eds) Advances on P2P, Parallel, Grid, Cloud and Internet Computing. 3PGCIC 2020. Lecture Notes in Networks and Systems, vol 158. Springer, Cham. DOI: 10.1007/978-3-030-61105-7_27

2. Journal Papers (A1)

a) Huybrechts T., De Bock Y., Haoxuan L., Hellinckx P. (2019) COBRA-HPA: a block generating tool to perform hybrid program analysis. In: Xhafa F. (eds) International Journal of Grid and Utility Computing, vol. 10, No. 2. Inderscience. DOI: 10.1504/IJGUC.2019.098211

b) Huybrechts T., Eyckerman R., Van den Langenbergh R., Vanneste S., Mercelis S., Hellinckx P. (2020) DUST Initializr — Graph-based platform for designing modules and applications in the revised DUST framework. In: Xhafa F. (eds) Internet of Things, vol. 11. Elsevier. DOI: 10.1016/J.IOT.2020.100229

c) Huybrechts T., Reiter P., Mercelis S., Famaey J., Latré S., Hellinckx P. (2021) Automated Testbench for Hybrid Machine Learning-based Worst-Case Energy Consumption Analysis on Batteryless IoT Devices. In: Shahid A., Zaidi S. A. (eds) Energies, vol. 14, No. 13. MDPI. DOI: 10.3390/en14133914

C.2 Coauthored Papers

1. Conference Papers (P1)

a) Vanneste, S., de Hoog, J., Huybrechts, T., Bosmans, S., Eyckerman, R., Sharif, M., Mercelis, S., Hellinckx, P. (2019) Distributed Uniform Streaming Framework: Towards an Elastic Fog Computing Platform for Event Stream Processing. In: Xhafa F., Leu FY., Ficco M., Yang CT. (eds) Advances on P2P, Parallel, Grid, Cloud and Internet Computing. 3PGCIC 2018. Lecture Notes on Data Engineering and Communications Technologies, vol 24. Springer, Cham. DOI: 10.1007/978-3-030-02607-3_39

b) Eyckerman R., Huybrechts T., Van den Langenbergh R., Casteels W., Mercelis S., Hellinckx P. (2021) Towards the Generalization of Distributed Software Communication. In: Barolli L., Takizawa M., Yoshihisa T., Amato F., Ikeda M. (eds) Advances on P2P, Parallel, Grid, Cloud and Internet Computing. 3PGCIC 2020. Lecture Notes in Networks and Systems, vol 158. Springer, Cham. DOI: 10.1007/978-3-030-61105-7_26

c) Slamnik-Kriještorac N., Van den Langenbergh R., Huybrechts T., Gutierrez S. M., Gil M. C., Marquez-Barja J. M. (2021) Cloud-based virtual labs vs. low-cost physical labs: what engineering students think. In: Klinger T., Kollmitzer C., Pester A. (eds) IEEE Global Engineering Education Conference. EDUCON 2021. Proceedings of the 2021 IEEE Global Engineering Education Conference (EDUCON). IEEE. DOI: 10.1109/EDUCON46332.2021.9454091

   d) Slamnik-Kriještorac N., Pinheiro J. F. N., Huybrechts T., van den Akker D., Marquez-Barja J. M. (2021) Adaptive Remote Experimentation for Engineering Students. In: Gaggi O., Manzoni P., Palazzi C. (eds) Conference on Information Technology for Social Good. GoodIT '21. Proceedings of the 2021 Conference on Information Technology for Social Good (GOODIT). ACM, New York. DOI: 10.1145/3462203.3475931

2. Journal Papers (A1)

   a) Vanneste, S., de Hoog, J., Huybrechts, T., Bosmans, S., Eyckerman, R., Sharif, M., Mercelis, S., Hellinckx, P. (2019) Distributed Uniform Streaming Framework: An Elastic Fog Computing Platform for Event Stream Processing and Platform Transparency. In: Giuli D., Hudson-Smith A. (eds) Future Internet 2019, vol. 11, No. 158, MDPI. DOI: 10.3390/FI11070158
   b) Carlan V., Huybrechts T., Hellinckx P., Vanelslander T. (2020) A universal middleware streaming framework and data analytics: Analysing their economic feasibility in road transport planning. In: Attard M., Budd L., Ison S. (eds) Research in Transportation Business & Management, vol. 34, Elsevier. DOI: 10.1016/J.RTBM.2019.100424

3. Journal Papers (A3)

   a) De Bock Y., Altmeyer S., Huybrechts T., Broeckhove J., Hellinckx P (2018) Task-set generator for schedulability analysis using the TACLebench benchmark suite. In: Bogomolov S., Sokolsky O. (eds) SIGBED Rev., vol. 15, No. 1, ACM Digital Library. DOI: 10.1145/3199610.3199613

Bibliography

[1] L. S. Vailshery. (Jan. 2021). Number of internet of things (IoT) connected devices worldwide in 2018, 2025 and 2030. English, Statista, [Online]. Available: `https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/` (visited on 23/06/2021) (cited on p. 1).

[2] L. Goasduff. (Oct. 2020). Gartner Survey Reveals 47% of Organizations Will Increase Investments in IoT Despite the Impact of COVID-19. English, Gartner, [Online]. Available: `https://www.gartner.com/en/newsroom/press-releases/2020-10-29-gartner-survey-reveals-47-percent-of-organizations-will-increase-investments-in-iot-despite-the-impact-of-covid-19-` (visited on 23/06/2021) (cited on p. 1).

[3] R. Eyckerman, S. Mercelis, J. Marquez-Barja and P. Hellinckx, 'Requirements for distributed task placement in the fog', *Internet of Things*, vol. 12, p. 100 237, 2020, ISSN: 2542-6605. DOI: `https://doi.org/10.1016/j.iot.2020.100237` (cited on pp. 1, 144, 145).

[4] R. Jayaseelan, T. Mitra and X. Li, 'Estimating the Worst-Case Energy Consumption of Embedded Software', in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, 2006, pp. 81–90. DOI: `10.1109/RTAS.2006.17` (cited on pp. 1, 16–19).

[5] Y. De Bock, S. Altmeyer, T. Huybrechts, J. Broeckhove and P. Hellinckx, 'Task-Set Generator for Schedulability Analysis Using the TACLebench Benchmark Suite', *SIGBED Review*, vol. 15, no. 1, pp. 22–28, Mar. 2018. DOI: `10.1145/3199610.3199613` (cited on pp. 3, 127).

[6] E. Mezzetti and T. Vardanega, 'On the Industrial Fitness of WCET Analysis', in *The 11th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2011 (cited on p. 3).

[7] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm and W. Yi, 'Building Timing Predictable Embedded Systems', *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4, Mar. 2014, ISSN: 1539-9087. DOI: `10.1145/2560033` (cited on p. 3).

[8] M. McManus, 'Environmental consequences of the use of batteries in low carbon systems: The impact of battery production', *Applied Energy*, vol. 93, pp. 288–295, 2012, ISSN: 0306-2619. DOI: 10.1016/j.apenergy.2011.12.062 (cited on p. 3).

[9] R. Rapier. (Jan. 2020). Environmental Implications Of Lead-Acid And Lithium-Ion Batteries, [Online]. Available: https://www.forbes.com/sites/rrapier/2020/01/19/environmental-implications-of-lead-acid-and-lithium-ion-batteries/ (visited on 23/06/2021) (cited on p. 3).

[10] Everactive. (2019). Overcoming the Battery Obstacle to Ubiquitous Sensing - Finally, [Online]. Available: https://everactive-media.s3.amazonaws.com/content/uploads/2019/06/17095844/Overcoming-the-Battery-Problem-to-Ubiquitous-Sensing_Everactive_June-2019.pdf (visited on 23/06/2021) (cited on p. 3).

[11] J. Hester and J. Sorber, 'The future of sensing is batteryless, intermittent, and awesome', in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '17, Delft, Netherlands: Association for Computing Machinery, 2017, ISBN: 9781450354592. DOI: 10.1145/3131672.3131699 (cited on p. 3).

[12] F. K. Shaikh and S. Zeadally, 'Energy harvesting in wireless sensor networks: A comprehensive review', *Renewable and Sustainable Energy Reviews*, vol. 55, pp. 1041–1054, 2016, ISSN: 1364-0321. DOI: 10.1016/j.rser.2015.11.010 (cited on p. 3).

[13] P. Spanik, M. Frivaldsky and A. Kanovsky, 'Life time of the electrolytic capacitors in power applications', in *2014 ELEKTRO*, 2014, pp. 233–238. DOI: 10.1109/ELEKTRO.2014.6848893 (cited on p. 3).

[14] R. Kötz and M. Carlen, 'Principles and applications of electrochemical capacitors', *Electrochimica Acta*, vol. 45, no. 15, pp. 2483–2498, 2000, ISSN: 0013-4686. DOI: 10.1016/S0013-4686(00)00354-6 (cited on p. 3).

[15] K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak and J. Hester, 'InK: Reactive Kernel for Tiny Batteryless Sensors', in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '18, Shenzhen, China: Association for Computing Machinery, 2018, pp. 41–53, ISBN: 9781450359528. DOI: 10.1145/3274783.3274837 (cited on p. 3).

[16] A. Sabovic, C. Delgado, D. Subotic, B. Jooris, E. De Poorter and J. Famaey, 'Energy-Aware Sensing on Battery-Less LoRaWAN Devices with Energy Harvesting', *Electronics*, vol. 9, no. 6, 2020, ISSN: 2079-9292. DOI: 10.3390/electronics9060904 (cited on p. 4).

[17] F. Yang, A. S. Thangarajan, W. Joosen, C. Huygens, D. Hughes, G. S. Ramachandran and B. Krishnamachari, 'AsTAR: Sustainable Battery Free Energy Harvesting for Heterogeneous Platforms and Dynamic Environments', in *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN '19, Beijing, China: Junction Publishing, 2019, pp. 71–82, ISBN: 9780994988638. DOI: 10.5555/3324320.3324329 (cited on p. 4).

[18] J. McCarthy, M. L. Minsky, N. Rochester and C. E. Shannon, 'A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, August 31, 1955', *AI Magazine*, vol. 27, no. 4, p. 12, Dec. 2006. DOI: `10.1609/aimag.v27i4.1904` (cited on pp. 4, 69).

[19] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu and X. Zheng, 'TensorFlow: A System for Large-Scale Machine Learning', in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, Savannah, GA, USA: USENIX Association, 2016, pp. 265–283, ISBN: 9781931971331 (cited on pp. 5, 83, 96, 137).

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala, 'PyTorch: An Imperative Style, High-Performance Deep Learning Library', in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., 2019, pp. 8026–8037. [Online]. Available: `https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf` (cited on pp. 5, 96).

[21] L. Dutta and S. Bharali, 'TinyML Meets IoT: A Comprehensive Survey', *Internet of Things*, vol. 16, p. 100 461, 2021, ISSN: 2542-6605. DOI: `10.1016/j.iot.2021.100461` (cited on pp. 5, 96).

[22] G. Chen, C. Parada and G. Heigold, 'Small-footprint keyword spotting using deep neural networks', in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2014, pp. 4087–4091. DOI: `10.1109/ICASSP.2014.6854370` (cited on pp. 5, 95).

[23] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*, First Edition. O'Reilly Media, Inc., 2019, ISBN: 9781492052043 (cited on pp. 5, 95–97).

[24] S. Reif, B. Herzog, J. Hemp, T. Hönig and W. Schröder-Preikschat, 'Precious: Resource-Demand Estimation for Embedded Neural Network Accelerators', in *First International Workshop on Benchmarking Machine Learning Workloads on Emerging Hardware*, Austin, Texas, USA, Mar. 2020 (cited on pp. 5, 97–99).

[25] T. Huybrechts, Y. De Bock, H. Li and P. Hellinckx, 'Hybrid Approach on Cache Aware Real-Time Scheduling for Multi-Core Systems', in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, F. Xhafa, L. Barolli and F. Amato, Eds., vol. 1, Cham: Springer International Publishing, 2017, pp. 759–768, ISBN: 978-3-319-49109-7. DOI: `10.1007/978-3-319-49109-7_73` (cited on pp. 7, 8).

[26] T. Huybrechts, Y. De Bock, L. Haoxuan and P. Hellinckx, 'COBRA-HPA: a Block Generating Tool to Perform Hybrid Program Analysis', *International Journal of Grid and Utility Computing*, vol. 10, no. 2, F. Xhafa, Ed., pp. 105–118, 2019. DOI: `10.1504/IJGUC.2019.098211` (cited on pp. 7, 8, 19, 20, 36, 147).

[27] T. Huybrechts, S. Mercelis and P. Hellinckx, ‘A New Hybrid Approach on WCET Analysis for Real-Time Systems Using Machine Learning’, in *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*, F. Brandner, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 63, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2018, 5:1–5:12. doi: `10.4230/OASICS.WCET.2018.5` (cited on pp. 7, 8, 18, 19, 36).

[28] T. Huybrechts, T. Cassimon, S. Mercelis and P. Hellinckx, ‘Introduction of Deep Neural Network in Hybrid WCET Analysis’, in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, F. Xhafa, F.-Y. Leu, M. Ficco and C.-T. Yang, Eds., vol. 24, Cham: Springer International Publishing, 2019, pp. 415–425, isbn: 978-3-030-02607-3. doi: `10.1007/978-3-030-02607-3_38` (cited on pp. 7, 8).

[29] T. Huybrechts, P. Reiter, S. Mercelis, J. Famaey, S. Latré and P. Hellinckx, ‘Automated Testbench for Hybrid Machine Learning-Based Worst-Case Energy Consumption Analysis on Batteryless IoT Devices’, *Energies*, vol. 14, no. 13, A. Shahid and S. A. R. Zaidi, Eds., 2021, issn: 1996-1073. doi: `10.3390/en14133914` (cited on pp. 7, 8).

[30] T. Huybrechts, S. Mercelis and P. Hellinckx, ‘A Survey on the Software and Hardware-Based Influences on the Worst-Case Execution Time’, in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, L. Barolli, M. Takizawa, T. Yoshihisa, F. Amato and M. Ikeda, Eds., vol. 158, Cham: Springer International Publishing, 2021, pp. 271–281, isbn: 978-3-030-61105-7. doi: `10.1007/978-3-030-61105-7_27` (cited on pp. 7, 16).

[31] T. Huybrechts, S. Vanneste, R. Eyckerman, J. de Hoog, S. Mercelis and P. Hellinckx, ‘DUST Initializr - CAD Drawing Platform for Designing Modules and Applications in the DUST Framework’, in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, L. Barolli, P. Hellinckx and J. Natwichai, Eds., vol. 96, Cham: Springer International Publishing, 2020, pp. 661–670, isbn: 978-3-030-33509-0. doi: `10.1007/978-3-030-33509-0_62` (cited on p. 8).

[32] T. Huybrechts, R. Eyckerman, R. Van den Langenbergh, S. Vanneste, S. Mercelis and P. Hellinckx, ‘DUST Initializr — Graph-based platform for designing modules and applications in the revised DUST framework’, *Internet of Things*, vol. 11, F. Xhafa, Ed., p. 100 229, 2020, issn: 2542-6605. doi: `10.1016/j.iot.2020.100229` (cited on p. 8).

[33] L. Thiele and R. Wilhelm, ‘Design for Timing Predictability’, vol. 28, pp. 157–177, 2 2004. doi: `10.1023/B:TIME.0000045316.66276.6e` (cited on p. 10).

[34] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger and B. Becker, ‘A Definition and Classification of Timing Anomalies’, in *6th International Workshop on Worst-Case Execution Time Analysis (WCET’06)*, F. Mueller, Ed., vol. 4, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006. doi: `10.4230/OASIcs.WCET.2006.671` (cited on pp. 10, 13, 16, 25, 33, 36).

[35] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat and P. Stenström, ‘The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools’, *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, May 2008, issn: 1539-9087. doi: `10.1145/1347375.1347389` (cited on pp. 12, 13, 15, 16, 18, 19, 23).

[36] L. Lamport, 'Time, Clocks, and the Ordering of Events in a Distributed System', *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978, ISSN: 0001-0782. DOI: `10.1145/359545.359563` (cited on p. 12).

[37] C. L. Liu and J. W. Layland, 'Scheduling algorithms for multiprogramming in a hard-real-time environment', *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973, ISSN: 0004-5411. DOI: `10.1145/321738.321743` (cited on p. 12).

[38] E. Bini and G. C. Buttazzo, 'The space of rate monotonic schedulability', in *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, 2002, pp. 169–178. DOI: `10.1109/REAL.2002.1181572` (cited on p. 12).

[39] R. I. Davis and A. Burns, 'A Survey of Hard Real-Time Scheduling for Multiprocessor Systems', *ACM Computing Surveys*, vol. 43, no. 4, Oct. 2011, ISSN: 0360-0300. DOI: `10.1145/1978802.1978814` (cited on p. 12).

[40] J. C. Palencia and M. González Harbour, 'Schedulability Analysis for Tasks with Static and Dynamic Offsets', in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, ser. RTSS '98, USA: IEEE Computer Society, 1998, pp. 26–37, ISBN: 081869212X. DOI: `10.1109/REAL.1998.739728` (cited on p. 12).

[41] T. Lundqvist and P. Stenstrom, 'Timing anomalies in dynamically scheduled microprocessors', in *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*, 1999, pp. 12–21. DOI: `10.1109/REAL.1999.818824` (cited on p. 13).

[42] R. Heckmann, M. Langenbach, S. Thesing and R. Wilhelm, 'The influence of processor architecture on the design and the results of WCET tools', *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, 2003. DOI: `10.1109/JPROC.2003.814618` (cited on p. 13).

[43] V.-A. Paun, B. Monsuez and P. Baufreton, 'On the Determinism of Multi-core Processors', in *1st French Singaporean Workshop on Formal Methods and Applications (FSFMA 2013)*, C. Choppy and J. Sun, Eds., ser. OpenAccess Series in Informatics (OASIcs), vol. 31, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2013, pp. 32–46. DOI: `10.4230/OASICS.FSFMA.2013.32` (cited on pp. 13, 32).

[44] R. L. Graham, 'Bounds for certain multiprocessing anomalies', *The Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966. DOI: `10.1002/j.1538-7305.1966.tb01709.x` (cited on p. 14).

[45] D. B. Stewart, 'Measuring Execution Time and Real-Time Performance', in *Embedded Systems Conference*, Boston, Sep. 2006, pp. 1–15 (cited on p. 14).

[46] J. Gustafsson and A. Ermedahl, 'Experiences from Applying WCET Analysis in Industrial Settings', in *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, IEEE, 2007, pp. 382–392. DOI: `10.1109/ISORC.2007.36` (cited on p. 14).

[47] J. Reineke, 'Caches in WCET Analysis', PhD thesis, Universität des Saarlandes, 2008 (cited on pp. 14, 18, 28, 30–32, 43, 46).

[48] M. Chisholm, B. C. Ward, N. Kim and J. H. Anderson, 'Cache Sharing and Isolation Tradeoffs in Multicore Mixed-Criticality Systems', in *2015 IEEE Real-Time Systems Symposium*, IEEE, 2015. DOI: `10.1109/rtss.2015.36` (cited on pp. 14, 34).

[49] R. Balasubramonian, N. P. Jouppi and N. Muralimanohar, ‘Multi-Core Cache Hierarchies’, *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–153, 2011. doi: `10.2200/s00365ed1v01y201105cac017` (cited on pp. 14, 34).

[50] M. Lv, N. Guan, J. Reineke, R. Wilhelm and W. Yi, ‘A survey on static cache analysis for real-time systems’, *Leibniz Transactions on Embedded Systems*, vol. 3, no. 1, pp. 05–1, 2016 (cited on pp. 14, 15, 30, 46).

[51] M. Wolf, D. Maydan and D.-K. Chen, ‘Combining loop transformations considering caches and scheduling’, in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, IEEE Comput. Soc. Press, 1996. doi: `10.1109/micro.1996.566468` (cited on pp. 14, 43).

[52] R. Arnold, F. Mueller, D. Whalley and M. Harmon, ‘Bounding worst-case instruction cache performance’, IEEE, 1994, pp. 172–181, isbn: 0-8186-6600-5. doi: `10.1109/REAL.1994.342718` (cited on pp. 14, 15).

[53] A. Betts, G. Bernat, R. Kirner, P. Puschner and I. Wenzel, ‘WCET Coverage for Pipelines’, University of York, Tech. Rep., 2006 (cited on pp. 14, 36, 63, 138).

[54] J. Engblom and B. Jonsson, ‘Processor Pipelines and Their Properties for Static WCET Analysis’, in *Embedded Software. EMSOFT 2002*, ser. Lecture Notes in Computer Science, vol. 2491, Springer Berlin Heidelberg, 2002, pp. 334–348. doi: `10.1007/3-540-45828-x_25` (cited on pp. 14, 36).

[55] S. Wilhelm, ‘Efficient Analysis of Pipeline Models for WCET Computation’, in *5th International Workshop on Worst-Case Execution Time Analysis (WCET’05)*, R. Wilhelm, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 1, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007, isbn: 978-3-939897-24-8. doi: `10.4230/OASIcs.WCET.2005.814` (cited on p. 14).

[56] J. Rosén, A. Andrei, P. Eles and Z. Peng, ‘Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip’, in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 2007, pp. 49–60. doi: `10.1109/RTSS.2007.24` (cited on pp. 14, 15).

[57] S. Chattopadhyay, A. Roychoudhury and T. Mitra, ‘Modeling Shared Cache and Bus in Multi-Cores for Timing Analysis’, in *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, ser. SCOPES ’10, St. Goar, Germany: Association for Computing Machinery, 2010, isbn: 9781450300841. doi: `10.1145/1811212.1811220` (cited on p. 14).

[58] R. Kirner and P. Puschner, ‘Transformation of path information for WCET analysis during compilation’, in *Proceedings 13th Euromicro Conference on Real-Time Systems*, 2001, pp. 29–36. doi: `10.1109/EMRTS.2001.933993` (cited on p. 15).

[59] R. Kirner, J. Knoop, A. Prantl, M. Schordan and I. Wenzel, ‘WCET Analysis: The Annotation Language Challenge’, in *7th International Workshop on Worst-Case Execution Time Analysis (WCET’07)*, C. Rochange, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 6, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007, isbn: 978-3-939897-05-7. doi: `10.4230/OASIcs.WCET.2007.1197` (cited on p. 15).

[60] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. Bo Sørensen, P. Wägemann and S. Wegener, 'TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research', in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, M. Schoeberl, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 55, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, 2:1–2:10. DOI: `10.4230/OASIcs.WCET.2016.2` (cited on pp. 15, 20, 21, 42, 65, 78, 83, 87, 127, 130).

[61] J. Gustafsson, B. Lisper, C. Sandberg and N. Bermudo, 'A tool for automatic flow analysis of C-programs for WCET calculation', in *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems, 2003. (WORDS 2003).*, IEEE, 2003, pp. 106–112. DOI: `10.1109/WORDS.2003.1218072` (cited on p. 15).

[62] J. Gustafsson, A. Ermedahl and B. Lisper, 'Towards a flow analysis for embedded system C programs', in *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, IEEE, 2005, pp. 287–297. DOI: `10.1109/WORDS.2005.53` (cited on p. 15).

[63] C. Healy, M. Sjödin, V. Rustagi and D. Whalley, 'Bounding loop iterations for timing analysis', in *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium*, IEEE, 1998, pp. 12–21. DOI: `10.1109/RTTAS.1998.683183` (cited on p. 15).

[64] F. Stappert, A. Ermedahl and J. Engblom, 'Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects', in *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '01, Atlanta, Georgia, USA: Association for Computing Machinery, 2001, pp. 132–140, ISBN: 1581133995. DOI: `10.1145/502217.502240` (cited on p. 15).

[65] A. Colin and G. Bernat, 'Scope-tree: A program representation for symbolic worst-case execution time analysis', in *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, IEEE, 2002, pp. 50–59. DOI: `10.1109/EMRTS.2002.1019185` (cited on p. 15).

[66] Y.-T. S. Li and S. Malik, 'Performance Analysis of Embedded Software Using Implicit Path Enumeration', in *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*, ser. DAC '95, San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 456–461, ISBN: 0897917251. DOI: `10.1145/217474.217570` (cited on pp. 15, 63, 138).

[67] A. Ermedahl, 'A Modular Tool Architecture for Worst-Case Execution Time Analysis', PhD thesis, Uppsala Universitet, 2003 (cited on p. 15).

[68] T. Lundqvist and P. Stenstrom, 'An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution', *Real-Time Systems*, vol. 17, pp. 183–207, Nov. 1999. DOI: `10.1023/A:1008138407139` (cited on p. 16).

[69] A. Biere, J. Knoop, L. Kovács and J. Zwirchmayr, 'The Auspicious Couple: Symbolic Execution and WCET Analysis', in *13th International Workshop on Worst-Case Execution Time Analysis*, C. Maiza, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 30, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 53–63, ISBN: 978-3-939897-54-5. DOI: `10.4230/OASIcs.WCET.2013.53` (cited on p. 16).

[70] J. Morse, S. Kerrison and K. Eder, 'On the Limitations of Analyzing Worst-Case Dynamic Energy of Processing', *ACM Transactions on Embedded Computing Systems*, vol. 17, no. 3, Feb. 2018, ISSN: 1539-9087. DOI: `10.1145/3173042` (cited on pp. 16, 17, 50, 86, 116).

[71] A. Sinha and A. P. Chandrakasan, 'Energy aware software', in *Proceedings of the 13th International Conference on VLSI Design*, ser. VLSID '00, USA: IEEE Computer Society, 2000, p. 50, ISBN: 0769504876. DOI: `10.1109/ICVD.2000.812584` (cited on p. 16).

[72] G. Ascia, V. Catania, M. Palesi and D. Sarta, 'An Instruction-Level Power Analysis Model with Data Dependency', *VLSI Design*, vol. 12, p. 9, 2001. DOI: `10.1155/2001/82129` (cited on pp. 17, 50).

[73] P. Wägemann, 'Energy-Constrained Real-Time Systems and Their Worst-Case Analyses', PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2020 (cited on pp. 17, 20, 63).

[74] V. Tiwari, S. Malik, A. Wolfe and M. Tien-Chien Lee, 'Instruction level power analysis and optimization of software', *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 13, pp. 223–238, 2 1996. DOI: `10.1007/BF01130407` (cited on pp. 17, 18, 39).

[75] S. Steinke, M. Knauer, L. Wehmeyer and P. Marwedel, 'An accurate and fine grain instruction-level energy model supporting software optimizations', in *Proceedings of International Symposium on Power And Timing Modeling, Optimizations and Simulation (PATMOS)*, IEEE Circuits and Systems Society, 2001 (cited on pp. 17, 18, 26).

[76] K. Eder, J. P. Gallagher, P. López-García, H. Muller, Z. Banković, K. Georgiou, R. Haemmerlé, M. V. Hermenegildo, B. Kafle, S. Kerrison, M. Kirkeby, M. Klemen, X. Li, U. Liqat, J. Morse, M. Rhiger and M. Rosendahl, 'ENTRA: Whole-systems energy transparency', *Microprocessors and Microsystems*, vol. 47, pp. 278–286, 2016, ISSN: 0141-9331. DOI: `10.1016/j.micpro.2016.07.003` (cited on pp. 17–19, 26, 27, 39).

[77] S. Kerrison and K. Eder, 'Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor', *ACM Transactions on Embedded Computing Systems*, vol. 14, no. 3, Apr. 2015, ISSN: 1539-9087. DOI: `10.1145/2700104` (cited on pp. 17, 19, 39).

[78] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse and K. Eder, 'Static Analysis of Energy Consumption for LLVM IR Programs', in *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '15, Sankt Goar, Germany: Association for Computing Machinery, 2015, pp. 12–21, ISBN: 9781450335935. DOI: `10.1145/2764967.2764974` (cited on pp. 17, 19, 26).

[79] P. Wägemann, T. Distler, T. Hönig, H. Janker, R. Kapitza and W. Schröder-Preikschat, 'Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems', in *2015 27th Euromicro Conference on Real-Time Systems*, IEEE, 2015, pp. 105–114. DOI: `10.1109/ECRTS.2015.17` (cited on pp. 17–19, 26).

[80] P. Altenbernd, J. Gustafsson, B. Lisper and F. Stappert, 'Early Execution Time-Estimation through Automatically Generated Timing Models', *Real-Time Systems*, vol. 52, no. 6, pp. 731–760, Nov. 2016, ISSN: 0922-6443. DOI: `10.1007/s11241-016-9250-7` (cited on pp. 18, 24).

[81] A. Bonenfant, D. Claraz, M. de Michiel and P. Sotin, 'Early WCET Prediction Using Machine Learning', in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, J. Reineke, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 57, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 5:1–5:9, ISBN: 978-3-95977-057-6. DOI: `10.4230/OASIcs.WCET.2017.5` (cited on pp. 18, 24, 70–72).

[82] T. Bachard, V. Careil, A. Gonon and G. Pacreau, 'Using Machine Learning for Timing Model Generation in Worst-Case Execution Time Estimation', Université de Rennes, Research report, 2019. [Online]. Available: `http://perso.eleves.ens-rennes.fr/people/antoine.gonon/WCET_rapport.pdf` (cited on pp. 18, 24, 124).

[83] P. Lokuciejewski and M. Peter, *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer Netherlands, 2011, 262 pp., ISBN: 978-90-481-9929-7. DOI: `10.1007/978-90-481-9929-7` (cited on pp. 18, 19, 53, 63, 138).

[84] B. Wegbreit, 'Mechanical Program Analysis', *Communications of the ACM*, vol. 18, no. 9, pp. 528–539, Sep. 1975, ISSN: 0001-0782. DOI: `10.1145/361002.361016` (cited on p. 18).

[85] M. Rosendahl, 'Automatic Complexity Analysis', in *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '89, Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, pp. 144–156, ISBN: 0897913280. DOI: `10.1145/99370.99381` (cited on p. 18).

[86] S. K. Debray, N.-W. Lin and M. Hermnegildo, 'Task Granularity Analysis in Logic Programs', *SIGPLAN Notices*, vol. 25, no. 6, pp. 174–188, Jun. 1990, ISSN: 0362-1340. DOI: `10.1145/93548.93564` (cited on p. 18).

[87] P. B. Vasconcelos and K. Hammond, 'Inferring cost equations for recursive, polymorphic and higher-order functional programs', in *Implementation of Functional Languages*, P. Trinder, G. J. Michaelson and R. Peña, Eds., vol. 3145, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 86–101, ISBN: 978-3-540-27861-0. DOI: `10.1007/978-3-540-27861-0_6` (cited on p. 18).

[88] E. Albert, P. Arenas, S. Genaim and G. Puebla, 'Closed-Form Upper Bounds in Static Cost Analysis', vol. 46, no. 2, pp. 161–203, 2011. DOI: `10.1007/s10817-010-9174-1` (cited on p. 18).

[89] A. Betts and G. Bernat, 'Tree-based WCET analysis on instrumentation point graphs', in *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, 2006, p. 8. doi: `10.1109/ISORC.2006.75` (cited on p. 18).

[90] B. Lisper, 'SWEET – A Tool for WCET Flow Analysis', in *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, T. Margaria and B. Steffen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 482–485, isbn: 978-3-662-45231-8. doi: `10.1007/978-3-662-45231-8_38` (cited on p. 19).

[91] C. Ballabriga, H. Cassé, C. Rochange and P. Sainrat, 'OTAWA: An Open Toolbox for Adaptive WCET Analysis', in *Software Technologies for Embedded and Ubiquitous Systems*, S. L. Min, R. Pettit, P. Puschner and T. Ungerer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–46, isbn: 978-3-642-16256-5. doi: `10.1007/978-3-642-16256-5_6` (cited on p. 19).

[92] N. Holsti and S. Saarinen, 'Status of the Bound-T WCET Tool', in *2nd International Workshop on Worst-Case Execution Time Analysis (WCET'2002)*, Austria, 2002, pp. 36–41 (cited on p. 19).

[93] D. Hardy, B. Rouxel and I. Puaut, 'The Heptane Static Worst-Case Execution Time Estimation Tool', in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, J. Reineke, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 57, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 8:1–8:12, isbn: 978-3-95977-057-6. doi: `10.4230/OASIcs.WCET.2017.8` (cited on p. 19).

[94] X. Li, Y. Liang, T. Mitra and A. Roychoudhury, 'Chronos: A timing analyzer for embedded software', *Science of Computer Programming*, vol. 69, no. 1, pp. 56–67, 2007, issn: 0167-6423. doi: `10.1016/j.scico.2007.01.014` (cited on p. 19).

[95] R. Heckmann and C. Ferdinand, 'Worst-Case Execution Time Prediction by Static Program Analysis', AbsInt Angewandte Informatik GmbH, Science Park 1, D-66123 Saarbrücken, German, Tech. Rep. [Online]. Available: `https://www.absint.com/aiT_WCET.pdf` (cited on p. 19).

[96] P. Wägemann, C. Dietrich, T. Distler, P. Ulbrich and W. Schröder-Preikschat, 'Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems', in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, S. Altmeyer, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 106, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 24:1–24:25, isbn: 978-3-95977-075-0. doi: `10.4230/LIPIcs.ECRTS.2018.24` (cited on p. 19).

[97] C. Hümbert. (Jul. 2021). EnergyAnalyzer — Static Energy Analysis for Embedded Applications, YouTube, [Online]. Available: `https://youtu.be/VhK3WpTNfYk` (visited on 15/06/2022) (cited on p. 19).

[98] S. M. Petters and G. Färber, 'Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible', in *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, ser. RTCSA '99, 1999, p. 442, isbn: 0769503063 (cited on p. 19).

[99] R. Kirner, I. Wenzel, B. Rieder and P. Puschner, 'Using Measurements as a Complement to Static Worst- Case Execution Time Analysis', *Intelligent Systems at the Service of Mankind*, vol. 2, pp. 205–226, 2006 (cited on p. 19).

[100] D. Kästner, M. Pister, S. Wegener and C. Ferdinand, 'TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis', in *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, S. Altmeyer, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 72, Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, 1:1–1:11, ISBN: 978-3-95977-118-4. DOI: `10.4230/OASIcs.WCET.2019.1` (cited on pp. 20, 54).

[101] 'Measurement-based timing and WCET analysis with RapiTime', Rapita Systems Ltd., Atlas House, Osbaldwick Link Road, York, Tech. Rep. [Online]. Available: `https://www.rapitasystems.com/files/MC-PB-101%20RapiTime%20Product%20Brief_3.pdf` (cited on pp. 20, 54).

[102] J. Gustafsson, A. Betts, A. Ermedahl and B. Lisper, 'The Mälardalen WCET Benchmarks: Past, Present And Future', in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, B. Lisper, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 15, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 136–146, ISBN: 978-3-939897-21-7. DOI: `10.4230/OASIcs.WCET.2010.136` (cited on p. 20).

[103] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge and R. Brown, 'MiBench: A free, commercially representative embedded benchmark suite', in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4*, 2001, pp. 3–14. DOI: `10.1109/WWC.2001.990739` (cited on p. 21).

[104] J. Pallister, S. J. Hollis and J. Bennett, 'BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms', *CoRR*, vol. abs/1308.5174, 2013. arXiv: `1308.5174`. [Online]. Available: `http://arxiv.org/abs/1308.5174` (cited on p. 21).

[105] C. Eichler, P. Wägemann and W. Schröder-Preikschat, 'GenEE: A Benchmark Generator for Static Analysis Tools of Energy-Constrained Cyber-Physical Systems', in *Proceedings of the 2nd Workshop on Benchmarking Cyber-Physical Systems and Internet of Things*, ser. CPS-IoTBench '19, Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 1–6, ISBN: 9781450366939. DOI: `10.1145/3312480.3313170` (cited on pp. 21, 26).

[106] J. Gustafsson, P. Altenbernd, A. Ermedahl and B. Lisper, 'Approximate Worst-Case Execution Time Analysis for Early Stage Embedded Systems Development', in *Software Technologies for Embedded and Ubiquitous Systems*, S. Lee and P. Narasimhan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 308–319, ISBN: 978-3-642-10265-3. DOI: `10.1007/978-3-642-10265-3_28` (cited on p. 23).

[107] P. Puschner and C. Koza, 'Calculating the Maximum, Execution Time of Real-time Programs', *Real-Time Systems*, vol. 1, no. 2, pp. 159–176, Sep. 1989, ISSN: 0922-6443. DOI: `10.1007/BF00571421` (cited on p. 23).

[108] B. W. Boehm, *Software Engineering Economics*, R. T. Yeh, Ed. Upper Saddle River, New Jersey: Prentice-Hall PT, 1981, ISBN: 0-13-822122-7 (cited on p. 23).

[109] J. Folkestad and R. Johnson, ‘Resolving the conflict between design and manufacturing: Integrated Rapid Prototyping and Rapid Tooling (IRPRT)’, *Journal of Industrial Technology*, vol. 17, p. 7, 4 Jan. 2001 (cited on p. 23).

[110] D. Bhandarkar and D. W. Clark, ‘Performance from architecture: comparing a RISC and a CISC with similar hardware organization’, in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems - ASPLOS-IV*, ACM Press, 1991. DOI: `10.1145/106972.107003` (cited on p. 26).

[111] D. Branco and P. Rangel Henriques, ‘Impact of GCC Optimization Levels in Energy Consumption During Program Execution’, *Acta Electrotechnica et Informatica*, vol. 16, pp. 20–26, Mar. 2016. DOI: `10.15546/aeei-2016-0004` (cited on p. 26).

[112] K. Georgiou, S. Kerrison, Z. Chamski and K. Eder, ‘Energy Transparency for Deeply Embedded Programs’, *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 1, Mar. 2017, ISSN: 1544-3566. DOI: `10.1145/3046679` (cited on pp. 26, 27).

[113] C. Lattner and V. Adve, ‘LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation’, in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO ’04, Palo Alto, California: IEEE Computer Society, 2004, p. 75, ISBN: 0769521029 (cited on p. 26).

[114] Y. Ahn, ‘Real-Time Task Scheduling under Thermal Constraints’, PhD thesis, Texas A&M University, 2010 (cited on p. 27).

[115] R. T. Gran, J. Segarra, C. Rodriguez, L. Aparicio and V. Viñals, ‘Optimizing a combined WCET-WCEC problem in instruction fetching for real-time systems’, *Journal of Systems Architecture*, vol. 59, no. 9, pp. 667–678, 2013, ISSN: 1383-7621. DOI: `10.1016/j.sysarc.2013.07.012` (cited on p. 28).

[116] Y. De Bock, ‘Hard real-time scheduling on virtualized embedded multi-core systems’, PhD thesis, Universiteit Antwerpen, 2018 (cited on pp. 29, 30, 138, 147).

[117] L. Sha, T. Abdelzaher, K.-E. årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky and A. K. Mok, ‘Real Time Scheduling Theory: A Historical Perspective’, *Real-Time Systems*, vol. 28, no. 2/3, pp. 101–155, 2004. DOI: `10.1023/b:time.0000045315.61234.1e` (cited on p. 29).

[118] J. A. Carbone. (19th Jan. 2016). Reduce preemption overhead in real-time embedded systems. A. Kalnoskas, Ed., Microcontroller Tips, [Online]. Available: `https://www.microcontrollertips.com/1581-2/` (visited on 25/07/2020) (cited on p. 30).

[119] S. Hahn, J. Reineke and R. Wilhelm, ‘Towards compositionality in execution time analysis’, *ACM SIGBED Review*, vol. 12, no. 1, pp. 28–36, 2015. DOI: `10.1145/2752801.2752805` (cited on pp. 31, 34).

[120] S. M. Petters, ‘Worst case execution time estimation for advanced processor architectures’, PhD thesis, Technischen Universität München, 2002 (cited on p. 32).

[121] J.-F. Deverge and I. Puaut, 'Safe measurement-based WCET estimation', in *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 1, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2005. DOI: `10.4230/OASICS.WCET.2005.808` (cited on p. 32).

[122] A. Malik, B. Moyer and D. Cermak, 'Low power unified cache architecture providing power and performance flexibility', Feb. 2000, pp. 241–243, ISBN: 1-58113-190-9. DOI: `10.1109/LPE.2000.155290` (cited on p. 33).

[123] C. Zhang, F. Vahid and W. Najjar, 'A Highly Configurable Cache for Low Energy Embedded Systems', *ACM Transactions on Embedded Computing Systems*, vol. 4, no. 2, pp. 363–387, May 2005, ISSN: 1539-9087. DOI: `10.1145/1067915.1067921` (cited on p. 33).

[124] W. Zhang and A. Gordon-Ross, 'A Survey on Cache Tuning from a Power/Energy Perspective', *ACM Computing Surveys*, vol. 45, no. 3, Jul. 2013, ISSN: 0360-0300. DOI: `10.1145/2480741.2480749` (cited on p. 33).

[125] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan and P. Marwedel, 'Scratchpad memory', in *Proceedings of the tenth international symposium on Hardware/software codesign - CODES '02*, ACM Press, 2002. DOI: `10.1145/774789.774805` (cited on p. 33).

[126] I. Puaut and C. Pais, 'Scratchpad memories vs locked caches in hard real-time systems: A quantitative comparison', in *2007 Design, Automation & Test in Europe Conference & Exhibition*, IEEE, 2007. DOI: `10.1109/date.2007.364510` (cited on p. 33).

[127] H. Rihani, M. Moy, C. Maiza and S. Altmeyer, 'WCET analysis in shared resources real-time systems with TDMA buses', in *Proceedings of the 23rd International Conference on Real Time and Networks Systems - RTNS '15*, ACM Press, 2015. DOI: `10.1145/2834848.2834871` (cited on p. 33).

[128] P. Bajaj and D. Padole, 'Arbitration Schemes for Multiprocessor Shared Bus', in *New Trends and Developments in Automotive System Engineering*, InTech, 2011. DOI: `10.5772/16197` (cited on p. 33).

[129] S. Mercelis, 'A systematic multi-layered approach for optimizing and parallelizing real-time media and audio applications', PhD thesis, Universiteit Antwerpen, 2016, 220 pp. (cited on pp. 33, 34, 48, 127).

[130] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø and A. Tocchi, 'T-CREST: Time-predictable multi-core architecture for embedded systems', *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015. DOI: `10.1016/j.sysarc.2015.04.002` (cited on p. 34).

[131] T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. Zaykov, Z. Petrov, B. Boddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Casse, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quinones, M. Panic, J. Abella, F. Cazorla, S. Uhrig, M. Rohde and A. Pyka,

'parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability', in *2013 Euromicro Conference on Digital System Design*, IEEE, 2013. DOI: 10.1109/dsd.2013.46 (cited on p. 34).

[132] J. R. C. Patterson. (Aug. 2016). Modern Microprocessors: A 90-Minute Guide!, [Online]. Available: http://www.lighterra.com/papers/modernmicroprocessors/ (visited on 25/07/2020) (cited on pp. 34–36, 44).

[133] D. Harvey and J. Van Der Hoeven, 'Integer multiplication in time O(n log n)', 2019, [Online]. Available: https://hal.archives-ouvertes.fr/hal-02070778 (cited on p. 38).

[134] Silicon Labs, *EFM32GG Reference Manual: Giant Gecko Series*, West Cesar Chavez, Austin, 2016 (cited on p. 38).

[135] R. Barnes, R. Zwetsloot, L. Clay, P. King and L. Lynch, 'Raspberry Pi 3', *The MagPi*, vol. 43, pp. 6–18, 2016 (cited on p. 38).

[136] J. M. Calandrino, H. Leontyev, A. Block, U. Devi and J. H. Anderson, 'LITMUS^{RT} : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers', *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pp. 111–126, 2006 (cited on p. 38).

[137] ARM, *ARM Cortex-A Series - Programmer's Guide*, Version: 4.0, 2013 (cited on p. 38).

[138] ARM Limited, *Cortex-M3 - Technical Reference Manual*, Revision r2p0, 2010 (cited on pp. 38, 55).

[139] J. Pallister, S. Kerrison, J. Morse and K. Eder, 'Data Dependent Energy Modeling for Worst Case Energy Consumption Analysis', in *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '17, Sankt Goar, Germany: Association for Computing Machinery, 2017, pp. 51–59, ISBN: 9781450350396. DOI: 10.1145/3078659.3078666 (cited on p. 39).

[140] P. Burden, A. Burnard, M. Hennell, C. Hills, G. McCall, S. Montgomery, C. Tapp and L. Whiting, *MISRA-C:2004, Guidelines for the use of the C language in critical systems*, ed. by G. McCall, 2nd ed., 2008 (cited on pp. 40, 46, 52).

[141] J. L. Bentley, D. Haken and J. B. Saxe, 'A General Method for Solving Divide-and-Conquer Recurrences', *ACM SIGACT News*, vol. 12, no. 3, pp. 36–44, 1980, ISSN: 0163-5700. DOI: 10.1145/1008861.1008865 (cited on p. 40).

[142] M. Akra and L. Bazzi, 'On the Solution of Linear Recurrence Equations', *Computational Optimization and Applications*, vol. 10, pp. 195–210, 1998. DOI: 10.1023/A:1018373005182 (cited on p. 41).

[143] J. Knoop, L. Kovács and J. Zwirchmayr, 'Symbolic loop bound computation for WCET analysis', in *Perspectives of Systems Informatics. PSI 2011*, ser. Lecture Notes in Computer Science, vol. 7162, Springer Berlin Heidelberg, 2012, pp. 227–242. DOI: 10.1007/978-3-642-29709-0_20 (cited on p. 42).

[144] M. Michiel, A. Bonenfant, H. Cassé and P. Sainrat, 'Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation', in *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008, pp. 161–166. DOI: 10.1109/RTCSA.2008.53 (cited on p. 42).

[145] J. E. Smith, 'A study of branch prediction strategies', in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ser. ISCA '81, Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, pp. 135–148 (cited on p. 45).

[146] R. R. Schaller, 'Moore's law: Past, present and future', *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, 1997. DOI: `10.1109/6.591665` (cited on p. 47).

[147] K. Rogozhin. (Mar. 2017). Vectorization, intel, [Online]. Available: `https://www.hpc.kaust.edu.sa/sites/default/files/files/public/HPCSAUDI17/1_Vectorization_Intro.pdf` (visited on 25/07/2020) (cited on p. 47).

[148] K. Diefendorff, 'Pentium III = Pentium II + SSE, Internet SSE Architecture Boosts Multimedia Performance', *Microprocessor Report*, Microprocessor Resources, vol. 13, no. 3, p. 7, 8th Mar. 1999 (cited on p. 47).

[149] ARM Limited, *Introducing NEON, Development article*, ARM Limited, 2nd Jun. 2009, p. 18 (cited on p. 47).

[150] intel. (3rd Apr. 2019). Requirements for vectorizable loops, Intel Corporation, [Online]. Available: `https://software.intel.com/content/www/us/en/develop/articles/requirements-for-vectorizable-loops.html` (visited on 25/07/2020) (cited on p. 48).

[151] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, Second Edition. Pearson Education, 2007, ISBN: 978-0321486813 (cited on p. 50).

[152] M. J. Schulte, N. Lindberg and A. Laxminarain, 'Performance evaluation of decimal floating-point arithmetic', in *Proceedings of the 6th IBM Austin Center for Advanced Studies Conference*, 2005 (cited on p. 50).

[153] A. Betts, N. Merriam and G. Bernat, 'Hybrid measurement-based WCET analysis at the source level using object-level traces', in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, B. Lisper, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 15, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 54–63, ISBN: 978-3-939897-21-7. DOI: `10.4230/OASIcs.WCET.2010.54` (cited on pp. 53, 55).

[154] ATMEL Corporation, *8-bit AVR Microcontroller with 32K/64K/128K Bytes of ISP Flash and CAN Contoller*, 2325 Orchard Parkway, San Jose, CA 95131, USA, 2008 (cited on p. 55).

[155] Digilent, Inc., *Nexys4 DDR FPGA Board Reference Manual*, 1300 Henley Court, Pullman, WA 99163, USA, 2016 (cited on p. 56).

[156] Keysight Technologies, *Keysight N6700 Modular Power System Family - Specifications Guide*, 10th ed., Keysight Technologies, Feb. 2019 (cited on p. 58).

[157] Nordic Semiconductor. (2021). Power Profiler Kit, Nordic Semiconductor, [Online]. Available: `https://www.nordicsemi.com/Software-and-tools/Development-Tools/Power-Profiler-Kit` (visited on 23/06/2021) (cited on pp. 58, 59).

[158] Silicon Labs, *UG197: EZR32 Leopard Gecko 915 MHz Wireless Starter Kit*, 2.00, Silicon Laboratories Inc., May 2015 (cited on pp. 58, 59, 78, 83).

[159] Jetperch, *Joulescope JS110 User's Guide - Precision DC Energy Analyzer*, 1.0, Jetperch LLC, Jun. 2020 (cited on pp. 58, 59, 62).

[160] LabNation. (2021). Smartscope, LabNation, [Online]. Available: `https://www.lab-nation.com` (visited on 19/11/2021) (cited on p. 59).

[161] RIGOL, *DP800 Series Programmable Linear DC Power Supply User's Guide*, RIGOL Technologies, Inc., Jun. 2016 (cited on p. 62).

[162] P. P. Puschner and A. V. Schedl, 'Computing Maximum Task Execution Times — A Graph-Based Approach', *Real-Time Systems*, vol. 13, no. 1, pp. 67–91, Jul. 1997, ISSN: 0922-6443. DOI: `10.1023/A:1007905003094` (cited on pp. 63, 138).

[163] P. Altenbernd, 'The false path problem in hard real-time programs', Jul. 1996, pp. 102–107, ISBN: 0-8186-7496-2. DOI: `10.1109/EMWRTS.1996.557827` (cited on p. 63).

[164] J.-I. Lee, S.-H. Park, H. Bang, T. H. Kim and S. D. Cha, 'A Hybrid Framework of Worst-Case Execution Time Analysis for Real-Time Embedded System Software', *2005 IEEE Aerospace Conference*, pp. 1–10, 2005. DOI: `10.1109/AERO.2005.1559632` (cited on pp. 63, 138).

[165] M. Zolda and R. Kirner, 'Calculating WCET estimates from timed traces', *Real-Time Systems*, vol. 52, pp. 38–87, 2016, ISSN: 1573-1383. DOI: `10.1007/s11241-015-9240-1` (cited on p. 63).

[166] A. Makhorin, *GNU Linear Programming Kit - Reference Manual for GLPK Version 5.0*, Free Software Foundation, Inc., Dec. 2020 (cited on pp. 64, 138).

[167] K. Panetta. (23rd Aug. 2021). 3 Themes Surface in the 2021 Hype Cycle for Emerging Technologies. English, Gartner, [Online]. Available: `https://www.gartner.com/smarterwithgartner/3-themes-surface-in-the-2021-hype-cycle-for-emerging-technologies` (visited on 23/11/2021) (cited on p. 69).

[168] D. Griffin, B. Lesage, I. Bate, F. Soboczenski and R. I. Davis, 'Forecast-Based Interference: Modelling Multicore Interference from Observable Factors', in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS '17, Grenoble, France: Association for Computing Machinery, 2017, pp. 198–207, ISBN: 9781450352864. DOI: `10.1145/3139258.3139275` (cited on p. 70).

[169] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 2nd Edition. O'Reilly Media, Inc., 2019, ISBN: 9781492032649 (cited on pp. 73–76, 82, 90, 110, 124, 125, 134, 135, 137).

[170] I. Guyon and A. Elisseeff, 'An Introduction to Variable and Feature Selection', *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, Mar. 2003, ISSN: 1532-4435 (cited on pp. 73, 124, 135).

[171] M. A. Hall, 'Correlation-Based Feature Selection for Discrete and Numeric Class Machine Learning', in *Proceedings of the Seventeenth International Conference on Machine Learning*, ser. ICML '00, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 359–366, ISBN: 1558607072 (cited on pp. 73, 124, 135).

[172] M. S. Oyamada, F. Zschornack and F. R. Wagner, 'Accurate Software Performance Estimation Using Domain Classification and Neural Networks', in *Proceedings of the 17th Symposium on Integrated Circuits and System Design*, ser. SBCCI '04, Pernambuco, Brazil: Association for Computing Machinery, 2004, pp. 175–180, ISBN: 1581139470. DOI: `10.1145/1016568.1016617` (cited on p. 76).

[173] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and É. Duchesnay, 'Scikit-learn: Machine Learning in Python', *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011 (cited on pp. 78, 96, 101, 137).

[174] 'MEAN ABSOLUTE PERCENTAGE ERROR (MAPE)', in *Encyclopedia of Production and Manufacturing Management*, P. M. Swamidass, Ed. Boston, MA: Springer US, 2000, pp. 462–462, ISBN: 978-1-4020-0612-8. DOI: `10.1007/1-4020-0612-8_580` (cited on pp. 79, 84, 87, 101).

[175] S. A. B. Shah, M. Rashid and M. Arif, 'Estimating WCET using prediction models to compute fitness function of a genetic algorithm', *Real-Time Systems*, vol. 56, pp. 28–63, Jan. 2020, ISSN: 1573-1383. DOI: `10.1007/s11241-020-09343-2` (cited on pp. 79, 84, 87, 101).

[176] R. J. Hyndman and A. B. Koehler, 'Another look at measures of forecast accuracy', *International Journal of Forecasting*, vol. 22, no. 4, pp. 679–688, 2006, ISSN: 0169-2070. DOI: `10.1016/j.ijforecast.2006.03.001` (cited on p. 83).

[177] Nordic Semiconductor. (). nRF52 DK - Development kit for Bluetooth Low Energy and Bluetooth mesh, Nordic Semiconductor, [Online]. Available: `https://www.nordicsemi.com/Software-and-tools/Development-Kits/nRF52-DK` (visited on 23/06/2021) (cited on pp. 86, 100, 110, 115).

[178] F. Hutter, L. Kotthoff and J. Vanschoren, Eds., *Automated Machine Learning - Methods, Systems, Challenges*, ser. The Springer Series on Challenges in Machine Learning. Springer, Cham, 2019. DOI: `10.1007/978-3-030-05318-5` (cited on pp. 88, 124).

[179] R. S. Olson, N. Bartley, R. J. Urbanowicz and J. H. Moore, 'Evaluation of a Tree-Based Pipeline Optimization Tool for Automating Data Science', in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16, Denver, Colorado, USA: Association for Computing Machinery, 2016, pp. 485–492, ISBN: 9781450342063. DOI: `10.1145/2908812.2908918` (cited on pp. 88, 124, 137).

[180] C.-W. Hsu, C.-C. Chang and C.-J. Lin, 'A Practical Guide to Support Vector Classification', Department of Computer Science, National Taiwan University, Tech. Rep., 19th May 2016. [Online]. Available: `https://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf` (cited on p. 92).

[181] M. Cranmer, D. Ashok and J. Brehmer, *MilesCranmer/PySR: v0.6.0*, version v0.6.0, 2021. DOI: `10.5281/zenodo.4885457` (cited on p. 92).

[182] TuringBot Software. (2021). TuringBot, TuringBot Software, [Online]. Available: `https://turingbotsoftware.com/` (visited on 23/07/2021) (cited on p. 92).

[183] M. Schmidt and H. Lipson, 'Distilling Free-Form Natural Laws from Experimental Data', *Science*, vol. 324, no. 5923, pp. 81–85, 2009, ISSN: 0036-8075. DOI: `10.1126/science.1165893` (cited on p. 92).

[184] S. Grigorescu, B. Trasnea, T. Cocias and G. Macesanu, 'A survey of deep learning techniques for autonomous driving', *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, 2020. DOI: `10.1002/rob.21918` (cited on p. 95).

[185] R. Wang, K. Nie, T. Wang, Y. Yang and B. Long, 'Deep Learning for Anomaly Detection', in *Proceedings of the 13th International Conference on Web Search and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 894–896, ISBN: 9781450368223. DOI: `10.1145/3336191.3371876` (cited on p. 95).

[186] K.-H. Yu, A. L. Beam and I. S. Kohane, 'Artificial intelligence in healthcare', *Nature Biomedical Engineering*, vol. 2, pp. 719–731, 10 2018. DOI: `10.1038/s41551-018-0305-z` (cited on p. 95).

[187] I. Georgievski and M. Aiello, 'Automated Planning for Ubiquitous Computing', *ACM Computing Surveys*, vol. 49, no. 4, Dec. 2016, ISSN: 0360-0300. DOI: `10.1145/3004294` (cited on p. 95).

[188] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D. Patterson, D. Pau, J.-s. Seo, J. Sieracki, U. Thakker, M. Verhelst and P. Yadav, 'Benchmarking TinyML Systems: Challenges and Direction', *arXiv preprint arXiv:2003.04821*, Jan. 2021 (cited on pp. 95, 98).

[189] D. Castelvecchi, 'The Black Box of AI', *Nature*, vol. 538, pp. 20–23, Oct. 2016. DOI: `10.1038/538020a` (cited on p. 95).

[190] A. Rai, 'Explainable AI: from black box to glass box', *Journal of the Academy of Marketing Science*, vol. 48, pp. 137–141, 1 2020. DOI: `10.1007/s11747-019-00710-5` (cited on p. 95).

[191] P. Gembaczka. (Dec. 2021). AIfES – an open-source standalone AI framework for almost any hardware, [Online]. Available: `https://cms.tinyml.org/wp-content/uploads/talks2022/tinyML_Talks_Pierre_Gembaczka_211201.pdf` (visited on 09/02/2022) (cited on p. 96).

[192] J. Lin, W.-M. Chen, Y. Lin, j. cohn john, C. Gan and S. Han, 'MCUNet: Tiny Deep Learning on IoT Devices', in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 11 711–11 722 (cited on p. 96).

[193] H. Ren, D. Anicic and T. A. Runkler, 'TinyOL: TinyML with Online-Learning on Microcontrollers', in *2021 International Joint Conference on Neural Networks (IJCNN)*, 2021, pp. 1–8. DOI: `10.1109/IJCNN52387.2021.9533927` (cited on p. 96).

[194] B. Sudharsan, P. Yadav, J. G. Breslin and M. Intizar Ali, 'Train++: An Incremental ML Model Training Algorithm to Create Self-Learning IoT Devices', in *2021 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/IOP/SCI)*, 2021, pp. 97–106. DOI: `10.1109/SWC50871.2021.00023` (cited on p. 96).

[195] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, P. Warden and R. Rhodes, 'TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems', in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis and I. Stoica, Eds., vol. 3, 2021, pp. 800–811 (cited on p. 96).

[196] N. Jouppi, C. Young, N. Patil and D. Patterson, 'Motivation for and Evaluation of the First Tensor Processing Unit', *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018. DOI: `10.1109/MM.2018.032271057` (cited on p. 97).

[197] P. Torelli and M. Bangale, 'Measuring Inference Performance of Machine-Learning Frameworks on Edge-class Devices with the MLMark™ Benchmark', EEMBC, Tech. Rep., 2019. [Online]. Available: `https://www.eembc.org/techlit/articles/MLMARK-WHITEPAPER-FINAL-1.pdf` (cited on p. 98).

[198] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang and Y. Zhou, 'MLPerf Inference Benchmark', *arXiv:1911.02549v2*, May 2020 (cited on p. 98).

[199] T. Cassimon, S. Vanneste, S. Bosmans, S. Mercelis and P. Hellinckx, 'Designing resource-constrained neural networks using neural architecture search targeting embedded devices', *Internet of Things*, vol. 12, p. 100 234, 2020, ISSN: 2542-6605. DOI: `10.1016/j.iot.2020.100234` (cited on pp. 98, 142).

[200] Y. Yang, A. Nam, M. Nasr-Azadani and T. Tung, 'Resource-Aware Pareto-Optimal Automated Machine Learning Platform', in *2020 3rd International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, 2020, pp. 1–6. DOI: `10.1109/ISRITI51436.2020.9315336` (cited on p. 98).

[201] J. Kim, Y. Park, G. Kim and S. J. Hwang, 'SplitNet: Learning to Semantically Split Deep Networks for Parameter Reduction and Model Parallelization', in *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, Eds., ser. Proceedings of Machine Learning Research, vol. 70, PMLR, 2017, pp. 1866–1874 (cited on p. 98).

[202] E. Malekhosseini, M. Hajabdollahi, N. Karimi, S. Samavi and S. Shirani, 'Splitting Convolutional Neural Network Structures for Efficient Inference', *CoRR*, Feb. 2020. arXiv: `2002.03302` (cited on p. 98).

[203] I. Prakash, A. Bansal, R. Verma and R. Shorey, 'SmartSplit: Latency-Energy-Memory Optimisation for CNN Splitting on Smartphone Environment', in *2022 14th International Conference on COMmunication Systems NETworkS (COMSNETS)*, 2022, pp. 549–557. DOI: `10.1109/COMSNETS53615.2022.9668610` (cited on p. 98).

[204] J. Bai, F. Lu, K. Zhang *et al.*, *ONNX: Open Neural Network Exchange*, `https://github.com/onnx`, 2019 (cited on p. 99).

[205] L. Deng, 'The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]', *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012. DOI: `10.1109/MSP.2012.2211477` (cited on p. 101).

[206] D. Goldberg, 'What Every Computer Scientist Should Know about Floating-Point Arithmetic', *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, Mar. 1991, ISSN: 0360-0300. DOI: `10.1145/103162.103163` (cited on p. 108).

[207] P. Wägemann, T. Distler, T. Hönig, V. Sieh and W. Schröder-Preikschat, 'GenE: A Benchmark Generator for WCET Analysis', in *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, F. J. Cazorla, Ed., ser. Open-Access Series in Informatics (OASIcs), vol. 47, Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 33–43, ISBN: 978-3-939897-95-8. DOI: `10.4230/OASIcs.WCET.2015.33` (cited on p. 124).

[208] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd. Pragmatic Bookshelf, 2013, ISBN: 1934356999 (cited on pp. 130, 136).

[209] T. Parr and S. Harwell, *Antlr Project*, `https://github.com/antlr`, 2014 (cited on p. 130).

[210] A. Ermedahl, J. Fredriksson, J. Gustafsson and P. Altenbernd, 'Deriving the Worst-Case Execution Time Input Values', Jul. 2009, pp. 45–54. DOI: `10.1109/ECRTS.2009.32` (cited on p. 133).

[211] I. Wenzel, B. Rieder, R. Kirner and P. Puschner, 'Automatic Timing Model Generation by CFG Partitioning and Model Checking', in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE '05, USA: IEEE Computer Society, 2005, pp. 606–611, ISBN: 0769522882. DOI: `10.1109/DATE.2005.76` (cited on p. 133).

[212] J. Fredriksson, T. Nolte, A. Ermedahl and M. Nolin, 'Clustering Worst-Case Execution Times for Software Components', in *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, C. Rochange, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 6, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007, ISBN: 978-3-939897-05-7. DOI: `10.4230/OASIcs.WCET.2007.1185` (cited on p. 133).

[213] C. Delgado and J. Famaey, 'Optimal energy-aware task scheduling for batteryless IoT devices', *IEEE Transactions on Emerging Topics in Computing*, 2021. DOI: `10.1109/TETC.2021.3086144` (cited on p. 138).

[214] H. Li, P. De Meulenaere and P. Hellinckx, 'Powerwindow: a Multi-component TACLeBench Benchmark for Timing Analysis', in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, vol. 1, Springer Nature, Oct. 2016, pp. 779–788. DOI: `10.1007/978-3-319-49109-7_75` (cited on p. 139).

[215] H. Li, 'Hybrid Timing Analysis Based on a Block-Isolation Technique', PhD thesis, University of Antwerp, 2021 (cited on p. 139).

[216] G. Behrmann, A. David and K. G. Larsen, 'A Tutorial on UPPAAL', in *Formal Methods for the Design of Real-Time Systems*, M. Bernardo and F. Corradini, Eds., ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, vol. 3185, pp. 200–236, ISBN: 978-3-540-30080-9. DOI: `10.1007/978-3-540-30080-9_7` (cited on p. 139).

[217] B. Koolmees, 'Validation of modeled behavior using UPPAAL', Final Bachelor Project, University of Technology Eindhoven, 2011 (cited on p. 139).

[218] D. Balemans, W. Casteels, S. Vanneste, J. de Hoog, S. Mercelis and P. Hellinckx, 'Resource efficient sensor fusion by knowledge-based network pruning', *Internet of Things*, vol. 11, p. 100 231, 2020, ISSN: 2542-6605. DOI: `10.1016/j.iot.2020.100231` (cited on p. 142).

[219] A. Gilchrist, 'Introducing Industry 4.0', in *Industry 4.0: The Industrial Internet of Things*. Berkeley, CA: Apress, 2016, pp. 195–215, ISBN: 978-1-4842-2047-4. DOI: `10.1007/978-1-4842-2047-4_13` (cited on p. 144).

[220] A. Schenck, W. Daems and J. Steckel, 'AirleakSlam: Detection of Pressurized Air Leaks Using Passive Ultrasonic Sensors', in *2019 IEEE Sensors*, 2019, pp. 1–4. DOI: `10.1109/SENSORS43011.2019.8956631` (cited on p. 144).

[221] C. Kyrkou, S. Timotheou, P. Kolios, T. Theocharides and C. Panayiotou, 'Drones: Augmenting Our Quality of Life', *IEEE Potentials*, vol. 38, no. 1, pp. 30–36, 2019. DOI: `10.1109/MPOT.2018.2850386` (cited on p. 144).

[222] D. Yanguas-Rojas, G. A. Cardona, J. Ramirez-Rugeles and E. Mojica-Nava, 'Victims search, identification, and evacuation with heterogeneous robot networks for search and rescue', in *2017 IEEE 3rd Colombian Conference on Automatic Control (CCAC)*, 2017, pp. 1–6. DOI: `10.1109/CCAC.2017.8276486` (cited on p. 144).

[223] M. Milford, S. Anthony and W. Scheirer, 'Self-Driving Vehicles: Key Technical Challenges and Progress Off the Road', *IEEE Potentials*, vol. 39, no. 1, pp. 37–45, 2020. DOI: `10.1109/MPOT.2019.2939376` (cited on p. 144).

[224] N. Balemans, P. Hellinckx and J. Steckel, 'Predicting LiDAR Data From Sonar Images', *IEEE Access*, vol. 9, pp. 57 897–57 906, 2021. DOI: `10.1109/ACCESS.2021.3072551` (cited on p. 144).

[225] S. Vanneste, J. de Hoog, T. Huybrechts, S. Bosmans, R. Eyckerman, M. Sharif, S. Mercelis and P. Hellinckx, 'Distributed Uniform Streaming Framework: An Elastic Fog Computing Platform for Event Stream Processing and Platform Transparency', *Future Internet*, vol. 11, no. 7, 2019, ISSN: 1999-5903. DOI: `10.3390/fi11070158` (cited on pp. 145, 146).

[226] K. Winter. (Apr. 2017). For Self-Driving Cars, There's Big Meaning Behind One Big Number: 4 Terabytes. English, intel, [Online]. Available: `https://newsroom.intel.com/editorials/self-driving-cars-big-meaning-behind-one-number-4-terabytes` (visited on 27/05/2020) (cited on p. 145).

[227] S. Bosmans, S. Mercelis, J. Denil and P. Hellinckx, 'Testing IoT systems using a hybrid simulation based testing approach', vol. 101, no. 7, pp. 857–872, 2019. DOI: `10.1007/S00607-018-0650-5` (cited on p. 145).

[228] R. Eyckerman, M. Sharif, S. Mercelis and P. Hellinckx, 'Context-Aware Distribution In Constrained IoT Environments', en, in *Proceedings of the 13th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2018)*, F. Xhafa, F.-Y. Leu, M. Ficco and C.-T. Yang, Eds., Springer International Publishing, 2019, pp. 437–446, ISBN: 978-3-030-02607-3. DOI: `10.1007/978-3-030-02607-3_40` (cited on p. 146).

[229] P. Bellavista, A. Corradi, L. Foschini and A. Pernafini, 'Data Distribution Service (DDS): A performance comparison of OpenSplice and RTI implementations', in *2013 IEEE Symposium on Computers and Communications (ISCC)*, 2013, pp. 000 377–000 383. DOI: `10.1109/ISCC.2013.6754976` (cited on p. 150).

[230] T. Koponen and T. Virtanen, 'A Service Discovery: A Service Broker Approach', in *37th Hawaii International Conference on System Sciences (HICSS-37 2004)*, IEEE Computer Society, 2004. DOI: `10.1109/HICSS.2004.1265669` (cited on p. 150).

[231] P. T. Eugster, P. A. Felber, R. Guerraoui and A.-M. Kermarrec, 'The Many Faces of Publish/Subscribe', *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, Jun. 2003, ISSN: 0360-0300. DOI: 10.1145/857076.857078 (cited on p. 151).

[232] C. Delgado, J. M. Sanz and J. Famaey, 'On the feasibility of battery-less LoRaWAN communications using energy harvesting', *2019 IEEE Global Communications Conference (GLOBECOM), 9-13 December, 2019, Waikoloa, Hawaii, USA*, pp. 1–6, 2019. DOI: 10.1109/GLOBECOM38437.2019.9013638 (cited on p. 152).