Data-Driven Approach to Estimate WCET for Real-Time Systems

A THESIS
SUBMITTED FOR THE DEGREE OF
Doctor of Philosophy
IN THE
Faculty of Engineering

BY

Vikash Kumar

Department of Computational and Data Sciences
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2024

DEDICATED TO

my parents

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. S.K Nandy for his guidance and support throughout my Ph.D. His faith and trust in me have helped broaden my vision and grow as a researcher. It would not have been possible to complete this work without his knowledge, insight, feedback, encouragement, and support on both professional and personal fronts. I am forever indebted to him for giving me the opportunities that have made me who I am today.

The thesis work is a collaborative effort. I am grateful to Prof. Akash Kumar from TU Dresden for his invaluable guidance. I was fortunate to receive advice from him on the research direction. I am indebted to him, as I would have never learned how to write efficiently without his support. The one-to-one meeting on various research topics for hours during my stay in Germany helped me shape my thesis based on his constructive suggestions.

I am grateful to Dr. Rajani Narayan from Morphing Machine Pvt. Ltd for her support at the beginning of my Ph.D. I would like to thank Prof. Soumyendu Raha for his help at the end of my Ph.D. when my supervisor superannuated. I would like to thank Sumana and Kavitha for being the friendliest and most wonderful seniors and helping me throughout my stay at IISc. I would like to extend my thanks to my labmates Madhav, Ritika, Anoop, Pushpa, Harish, Prateek, Bikram, Vishwas, and Ravikiran for all the technical and non-technical discussions throughout this journey.

I would like to mention my dearest friends Sourav, Avneeth, Shubham, Ankur and Rohit for making my days at IISc more memorable and teaching a lesson not to take life too seriously, as ups and downs are always there. They always kept me pushing forward despite all the hurdles I have faced in IISc.

I want to express my thanks to Behnaz and Siddharth for the discussion on the research topic during my stay in Germany. I would also like to thank Traimbak from the Asian Institute of Technology, Thailand, for teaching me how to cook and enjoy life independently in Germany.

I would like to thank my course instructors for providing the skill to complete my research work. I am grateful to all my department staff, CDS, especially Mary Anitha, for helping me in all administrative matters. I also like to thank Conny from TU Dresden for providing me with all the help for getting my visa done on time.

I greatly appreciate the financial support provided by the Ministry of Human Resource Development (MHRD) of India. I feel honored to receive a scholarship from the Center for Advancing Electronics Dresden (CFAED) of Germany. Lastly, I would like to thank my family, especially my parents and elder brother, for all the support, encouragement, and love.

A new chapter of my life's journey is about to be written as my PhD voyage has come to an end.

Abstract

Estimating Worst-Case Execution Time (WCET) is paramount for developing Real-Time and Embedded systems. The operating system's scheduler uses the estimated WCET to schedule each task of these systems before the assigned deadline, and failure may lead to catastrophic events such as resource damage or even life loss. These systems must satisfy the timing constraints. For instance, it is essential to know that car airbags open fast enough to save lives. The major components required to estimate WCET are architecture or platform, application, and worst-case data. In this regard, we propose novel methods for these components using machine learning techniques to estimate WCET safely and precisely to make these systems more predictable and reliable than traditional approaches.

- **Estimation of WCET on GPU architecture:** With the advances in machine learning and artificial intelligence in every field of life, due to its tendency to solve many problems with accuracy, it requires Graphics Processing Units (GPUs) to provide massive parallelism for computation. GPUs are designed to provide high-performance throughput, but their integration into real-time systems focuses on predictability because most safety-critical applications have strict deadlines that need to be followed to avoid unwanted situations. We propose a Machine Learning approach to estimate the WCET of the GPU kernel from the binary of the applications. The approach helps reduce the significant design space exploration in a short time. We use a measurement-based approach to train the machine-learning model using different kernel instructions, which can predict the WCET of the GPU kernel to detect timing misconfiguration in the later development phase of the systems.

- **Estimation of WCET on Mixed-Criticality Systems:** In Mixed-Criticality (MC) Systems, there is a trend of having multiple functionalities upon a single shared computing platform for better cost and power efficiency. In this regard, estimating the suitable optimistic WCET based on the different system modes is essential to provide these functionalities. A single application has assigned multiple WCETs based on the criticality of the system, such as safety-critical, mission-critical, and non-critical. We propose ES-OMICS, a novel method to estimate suitable optimistic WCET using a Machine Learning model. Our approach is based on the application's source, and the model is trained based on the large data set. To prove the effectiveness of our approach, we evaluated it with a newly defined metric EDT using an analytical solution that allows us to compute the optimum value in a mixed-criticality system based on experimentation. Our experimental results outperform all the previous state-of-the-art approaches.

- **Estimation of Worst-Case Data for WCET:** Worst-Case Data which gives maximum execution time, plays a vital role in the estimation of WCET. An evolutionary algorithm, such as the Genetic Algorithm, has been employed to generate the Worst-Case Data. The complexity of an evolutionary algorithm requires the use of several computational resources. We propose a method to replace the hardware and simulator used in the evolution process with Machine Learning models. This method reduces the overall time required to generate Worst-Case Data. Different machine learning models are trained to integrate with genetic algorithms. The feasibility of the proposed approach is validated using benchmarks from different domains. The results show the speedup in the generation of Worst-Case Data.

- **Estimation of Early WCET:** WCET is available to us in the last stage of systems development when the hardware is available, and the application code is compiled. Different methodologies measure the WCET, but none give early insights into WCET, which is crucial for system development. If the system designers overestimate WCET in the early stage, then it would lead to an overqualified system, which will increase the cost of the final product, and if they underestimate WCET in the early stage, then it would lead to financial loss as the system would not perform as expected. We propose to estimate early WCET using Machine Learning and Deep Neural Networks as an approximate predictor model for hardware architecture and compiler. This model predicts the WCET based on the source code without compiling and running on the hardware architecture. The resulting WCET needs to be revised to be used as an upper bound on the WCET. However, getting these results in the early stages of system development is an essential prerequisite for the system's dimension's and configuration of the hardware setup.

Publications based on this Thesis

1. Vikash Kumar, Behnaz Ranjbar and Akash Kumar, “Utilizing Machine Learning Techniques for Worst-Case Execution Time Estimation on GPU Architectures,” in IEEE Access, vol. 12, pp. 41464-41478, 2024, doi: 10.1109/ACCESS.2024.3379018.

2. Vikash Kumar, Behnaz Ranjbar and Akash Kumar, “ESOMICS: ML-Based Timing Behavior Analysis for Efficient Mixed-Criticality System Design,” in IEEE Access, vol. 12, pp. 67013-67024, 2024, doi: 10.1109/ACCESS.2024.3396225.

3. Vikash Kumar, Behnaz Ranjbar and Akash Kumar, “Motivating the Use of Machine-Learning For Improving Timing Behaviour of Embedded Mixed-Criticality Systems,” In proceeding DATE 2024.

4. Vikash Kumar, “Estimation of an Early WCET Using Different Machine Learning Approaches,” In: Barolli, L. (eds) Advances on P2P, Parallel, Grid, Cloud and Internet Computing. 3PGCIC 2022. Lecture Notes in Networks and Systems, vol 571. Springer, Cham. https://doi.org/10.1007/978-3-031-19945-5_30.

5. Vikash Kumar, “An integrated approach of Genetic Algorithm and Machine Learning for generation of Worst-Case Data for Real-Time Systems,” 2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT). doi: 10.1109/DS-RT55542.2022.9932054.

6. Vikash Kumar, “Deep Neural Network Approach to Estimate Early Worst-Case Execution Time,” 2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC), San Antonio, TX, USA, 2021, pp. 1-8, doi: 10.1109/DASC52595.2021.9594326.

Contents

List of Figures

List of Tables

Keywords

Real-Time Systems, Worst-Case Execution Time Analysis, Machine Learning, General Processing Unit, Mixed-Criticality Systems, Worst Case-Data, Neural Networks

Notation and Abbreviations

ACC Adaptive Cruise Control.
ACET Average-Case Execution Time.
ADAS Advanced Driver Assistance System.
AI Artificial Intelligence.
ALU Arithmetic Logic Unit
BCET Best-Case Execution Time.
BLQ Bank Load Queue
CFG Control Flow Graph.
CUDA Compute Unified Device Architecture
DDP Design and Development Phases
DL Deep Learning
DNN Deep Neural Network.
DPU Double-Precision Unit
ECU Electronic Control Unit.
ERTS Embedded Real-Time Systems.
ESA European Space Agency.
FMA Fused Multiply-Add
GA Genetic Algorithm.
GPU Graphics Processing Unit
ILP Integer Linear Programming.
IoT Internet of Things.
LSU Load-Store Unit
MCS Mixed-Criticality System.
ML Machine Learning
MLP Multi-Layer Perceptron
MAE Mean Absolute Error
MOET Maximum Observed Execution Time

MPS Multi-Process Service
MRQ Memory-Request Queue
MSE Mean Squared Error
MSHR Miss Status Handling Register
NoC Network on Chip.
OLS Ordinary Least Squares
OS Operating System.
OTAWA Open Tool for Adaptative WCET Analysis.
QoS Quality of Service.
ReLU Rectified Linear Unit
RF Register File
RMSE Root Mean Square Error.
ROP Raster Operation Processor
RQ Research Question.
RTS Real-Time System.
SE Symbolic Execution
SFP Single Feasible Path.
SFU Special-Function Unit
SIMD Single Instruction Multiple Data
SIMT Single Instruction Multiple Thread
SM Streaming Multiprocessor
SMK Simultaneous Multi-Kernel
SP Spatial Partitioning
SPs Scalar Processors
SVM Support Vector Machine
SVR Support Vector Regression
SWEET SWEdish Execution Time Analysis Tool.
WCET Worst-Case Execution Time.

Chapter 1

Introduction and Motivation

Our society has increasingly depended on Embedded Real-Time Systems (ERTS) over the last few decades. These systems are now essential to our lives, bringing convenience and improving several facets of our everyday activities. Mobile phones, for instance, are now commonplace and enable us to contact anybody from any part of the world at any time. Asking for directions has been supplanted with navigation systems, allowing us to go between places without difficulty. Watching live sports and news broadcasts on television has allowed us to be entertained and informed while lounging on our couches or sofas. In order to enable efficient and safe travel to any location globally, ERTS has been installed into vehicles such as cars, trains, and planes. The internet connection that each of these gadgets has connects them all. These ERTS can effortlessly communicate with one another and deliver their respective services without any hiccups or delays, thanks to the internet. The Internet of Things (IoT) ecosystem, where billions of connected and communicative gadgets are found, results from this interconnection. In 2019, there were an estimated 9 billion Internet of Things (IoT) devices linked to the internet, and it is predicted that this number will increase to 30 billion by 2030 [139], as shown in Figure 1.1. The explosive growth of IoT devices is evidence of society's rising need for embedded real-time solutions. Given these technologies' significant impact on our lives, designers must prioritize dependability and safety when designing and producing such items. Our lives increasingly depend on the smooth operation of these ERTS, whether it is the dependability of a navigation system directing someone to their destination or the safety measures in a car.

Nowadays, almost zero percent of microprocessors are used in computers. Industries are inclined towards ERTS because they cover nearly 99% of the market [37]. Less than 1% of all microprocessors sold yearly are used in general-purpose computers, including PCs, Macs, engineering workstations, Cray supercomputers, and other general-purpose computers. The remaining are used in ERTS, such as Refrigerators, Dishwashers, Coffee makers, Washers,

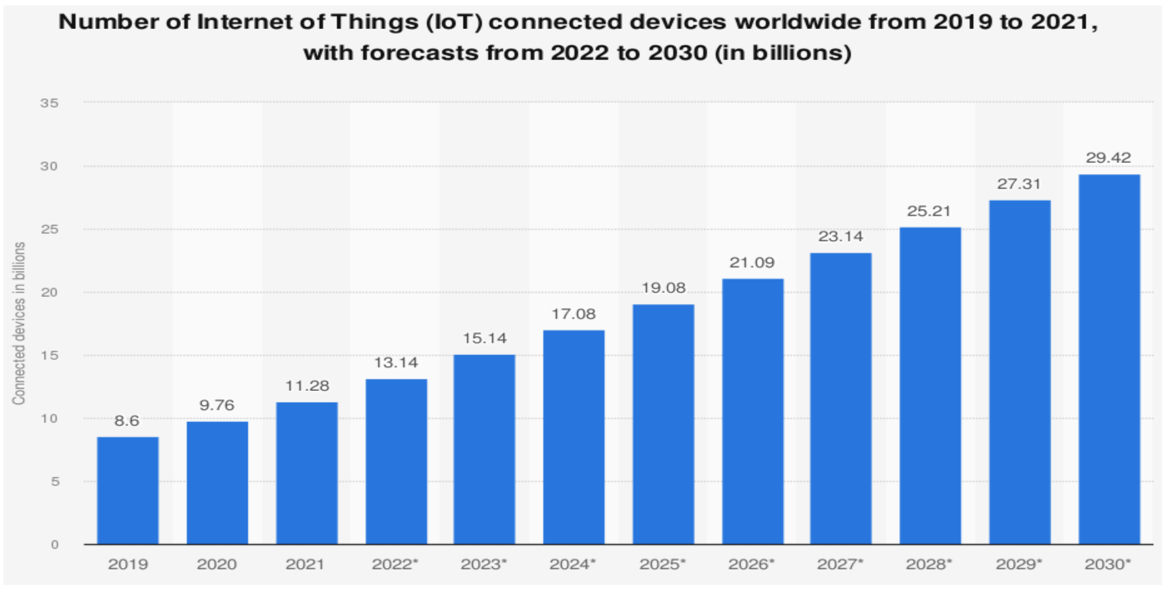


Figure 1.1: Number of connected devices increasing rapidly [139]

Dryers, Microwave ovens, Printers, Fax, Augmented/Virtual Reality, and so on. All these ERTS need to satisfy some requirements such as low cost, constraint to size, less energy consumption, and real-time behavior to create affordable, reliable, and safe systems.

Since ERTS are so ubiquitous, they are commonly used in safety-critical application domains such as automatic braking systems in automotive [103], insulin pumps in health care [138], and fly-by-wire systems used in avionics [114]. The correctness of these systems depends on the logical function and the time at which results are produced. In such systems, timing deadlines must be satisfied. Otherwise, it could lead to environmental resource damage or loss of life. For example, an airbag in the car must open on time if there is any collision to save lives. If the airbag opens a little early or late, then it would be of no use because, in both cases, timing deadlines are missed. The main research question in designing these systems is ensuring safety and real-time constraints.

Worst-Case Execution Time (WCET) analysis aims to notify users of the program's longest execution time before the program is used in the final product. WCET estimates are a crucial component in providing guaranteed service of system behavior because this value is used by the Operating System (OS) scheduler to schedule each of the tasks/applications before their deadline to prevent any catastrophic consequences. The estimation of WCET needs three major components: a Hardware architecture/platform, an Application, and Worst-Case Data that leads to WCET. In general, the actual WCET of an application is unknown. Therefore, it is necessary to derive an estimate of the WCET. Because a WCET estimation must securely upper bound the application's execution time, it inevitably overestimates the actual WCET. The unpredictable behavior of the systems due to hardware architecture and application complexity is the main reason for overestimation.

Several issues emerge in the design of a safety-critical system to satisfy the timing constraints. Designing such systems to ensure timeliness, dependability, and safety criteria is one of the crucial concerns that must be addressed [145]. An autonomous car is a perfect example of a safety-critical system because these systems must respond and behave correctly. For instance, a vehicle contains hundreds of Electronic Control Units (ECU) whose task is to detect pedestrians crossing the roads, maintain the lane, and use the automatic emergency brake if there is any collision. The hardware platform must be planned so that the applications/tasks meet their deadlines to construct a safety-critical system. However, to determine the schedulability of each task, we need to know the WCET required to execute each task. The different components involved in WCET estimation lead to enhanced complexity. The modern architecture/platform with advanced micro-architectures such as deep pipelines, branch predictions, and multi-level caches complex the WCET estimation due to variable execution time. Accelerators with thousands of cores and shared resources such as Network on Chip (NoC), memory hierarchy, and controller to improve average performance, highly affecting the timing behavior of applications [112]. The application code may follow several separate execution pathways, each with a unique set of instructions and memory accesses, resulting in a unique execution time that makes WCET analysis more difficult. The data that leads to WCET is also unknown in the initial stage, and applying all the inputs is not feasible.

Artificial Intelligence (AI) [99] is an emerging technology in every field, with a better solution to complex problems than traditional software algorithms that describe a set of rules for the given input and output. Different popular AI approaches, such as Machine Learning (ML) [100] and Deep Neural Networks (DNN) [125], outperform prior hand-crafted algorithms with better accuracy. The popularity of AI makes it a perfect choice to integrate it with safety-critical systems.

This thesis addresses the challenges associated with the design of Real-Time Systems (RTSs). In this thesis, the term ERTS and RTS are used interchangeably. We propose an ML approach for estimating WCET from an architectural point of view. We also present an ML approach to estimate the optimistic WCET when an application has more than one WCETs, which is very popular in Mixed-Criticality Systems (MCS). We introduced a theoretical-based scheme to improve the Quality of Service (QoS) of the MCS. We propose the combination of Genetic Algorithm (GA) and ML to determine the worst case data [81]. Finally, we present a novel approach using ML and DNN to estimate early WCET for design space exploration [79],[83].

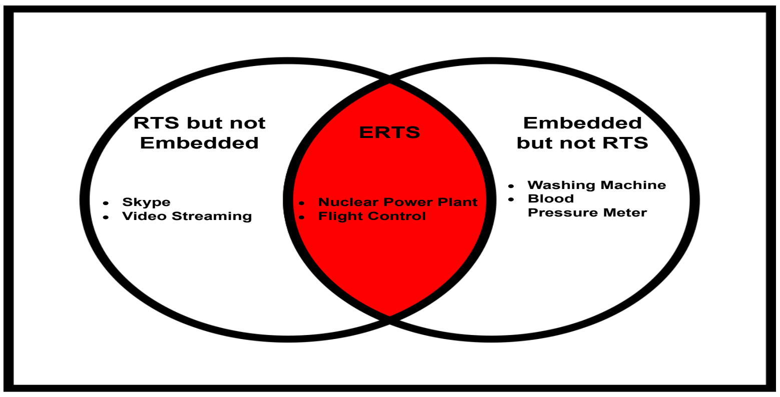


Figure 1.2: Difference between Embedded system and RTS

The rest of this chapter is structured as follows. In Section 1.1, we summarize the RTS definition, properties, and its different kinds. Section 1.2 presents the need for WCET analysis. We introduced the AI definition and its usefulness in RTS in Section 1.3. Then, in Section 1.4, we present the research questions and summarize the research challenges that need to be solved. Finally, Section 1.5 and Section 1.6 describe this thesis's key contributions and present the organization of the remaining chapter, respectively.

1.1 Real-Time System (RTS)

RTSs are computer systems designed to respond to events or inputs within a specified time frame, known as a deadline [75]. The correct behavior of the RTS depends not only on the correct logical output but also on the time frame at which output is produced. Depending on the specific application, this timeframe can range from microseconds to seconds. A wide range of applications use RTS where timing is critical, and they need specialized hardware and software to ensure they can meet their timing requirements. There is a misconception among people regarding the Embedded system and RTS. They assume both are the same, but this is not the case. An embedded system is a computer system embedded in a physical or mechanical system that usually performs a set of instructions and often interacts with its environment. In RTS, the system must react on time; otherwise, something may go wrong. Figure 1.2 shows the difference between them and their applications. Among the various systems worldwide, there exist two distinct categories. The first category includes embedded systems but not RTS, while the second category comprises RTS but not embedded systems. However, a subset of systems, ERTS, which are safety-critical systems where missing deadlines are not allowed, falls into the intersection of both categories. There are two different kinds of RTS, which are defined below.

1.1.1 Hard RTS

Hard RTS is a system where responses must occur within the required deadline [28]. Any wrong or late decision would endanger human life or the environment's safety. Such systems are also known as safety-critical systems, where failure to meet the deadline could cause a catastrophic outcome. One example of a hard RTS is a control system installed in a Nuclear Power Plant to detect leakage. Suppose the leakage isn't detected on time, and the leakage signal isn't sent to the control station. In that case, nuclear gas will spread into our environment, which could pollute our environment and lead to the deaths of many people, birds, and animals.

Another example of a system with a strict timing requirement is Adaptive Cruise Control (ACC), an Advanced Driver Assistance System (ADAS) that uses sensors and control algorithms to maintain a safe and consistent distance between the driver's vehicle and the vehicle in front of it. ACC measures the distance and speed of the automobile in front of the driver's car using sensors like radar, lidar, or cameras. The technology calculates the necessary acceleration or deceleration to maintain a safe following distance by continually measuring the distance and speed of the lead vehicle. By keeping a safe distance from the car in front of them, minimizing driver tiredness and tension, and enhancing the overall driving experience, ACC systems are intended to increase safety and comfort for drivers.

1.1.2 Soft RTS

Soft RTS is a system where deadlines are important but will still function if deadlines are occasionally missed [27]. Any wrong or late decision would reduce user satisfaction or QoS. Examples of soft real-time systems include multimedia systems, video games, and web servers. A missed frame using a Skype call to friends or colleagues would be annoying, but no one is killed or injured because of the interruption. Usually, in such a system, failure to meet deadlines means that the QoS is reduced, but the systems will still provide service.

Another example of Soft RTS is a web server. It is a piece of software that provides web pages and other resources in response to requests from clients (like web browsers). In order to provide a decent user experience, a soft real-time web server must be able to react to requests in an acceptable amount of time, but it is not subject to the same rigorous timing constraints as a hard real-time system. When a user requests a web page, the server is required to process the request and respond in a timely manner. Also, the server must be able to process several requests from several clients at once without sluggishness or failure. Nevertheless, if the server is overloaded, there are network or hardware problems, or there is a stringent deadline, the server might be unable to answer in time. The server may respond in these circumstances with reduced performance, such as sluggish response times or disconnected connections. Although these problems might harm the user experience, they are typically not severe and can be fixed by addressing the underlying problem.

1.2 The Need for WCET Analysis

WCET analysis [4] is an essential process in software engineering that involves analyzing and determining the maximum time it takes for a program or system to complete a particular task under certain conditions. When a system is in charge of carrying out numerous concurrent activities, it must be demonstrated that even in the worst-case situation, all of these tasks can achieve their deadlines. These guarantees are important to know before the system is out in the market. It's crucial to consider the execution time requirements of various tasks in order to generate such overall timing assurances. As a result, WCET analysis offers a strong foundation for creating RTS that are better and safer. This analysis is important for safety-critical systems, such as those used in aerospace, automotive, and medical industries, where any errors or delays can have catastrophic consequences. The need for WCET analysis can be summarized in the following points:

- **Predictability and Safety:** Systems that are safety-critical must be reliable and secure. Thus, they must always perform as planned and never endanger people or the environment. The identification and analysis of the worst-case situations that the system may experience are crucial in safety-critical systems. This is due to the potential influence that these circumstances may have on the system's safety and predictability. For instance, if the worst-case execution time of a brake system in an automobile system is incorrectly calculated, the braking distance may be greater than anticipated, increasing the risk of an accident. WCET analysis helps identify and quantify the worst-case scenarios for a system. This involves analyzing the program code to determine the maximum time it takes to execute a specific piece of code under a set of conditions. The conditions could be inputs that trigger specific code paths or the worst-case behavior of external devices like sensors or communication channels. By verifying that the system can handle these worst-case scenarios within the required time constraints, developers can ensure that the system is predictable and safe.

- **Compliance:** Safety-critical systems must undergo WCET analysis in accordance with a number of tight norms and laws. Safety-critical systems must adhere to these standards and laws' detailed specifications and high levels of quality. For instance, the automobile sector needs to comply with ISO 26262, while the aerospace industry needs to comply with DO-178B/C [124]. Adopting a multi-core processor in the avionics domain needs to follow CAST-32A for certification [30]. These standards include detailed instructions on how to conduct WCET analysis and what the outcomes should be.

- **Optimization:** WCET analysis can also be used to optimize the performance of a system. By identifying the parts of a program that take the most time to execute, developers can focus their optimization efforts on those areas to improve the overall performance of the system. Optimization efforts can include code refactoring, algorithmic improvements, or hardware upgrades. By optimizing the system's performance, developers can improve the system's reliability, reduce its power consumption, and extend its lifespan.

- **Cost and Time Savings:** Developers may address possible problems and delays in a system's execution using WCET analysis before they become more difficult and expensive to solve. Developers can locate possible bottlenecks and portions of the code that need to be optimized by examining the program code and determining the worst-case situations. By cutting down on the amount of time needed to build and test the system, can result in cost savings. Early problem detection might also lessen the likelihood of expensive redesign or rework later in the development cycle.

The different execution time estimates are shown in Figure 1.3. The curve represents the measured and possible execution time for some real-time tasks [145]. The WCET is the longest time the application takes to complete its execution on the target hardware. There are other execution time measures that can be used to describe the timing behavior of an application. The Best-Case Execution Time (BCET) is the shortest time the application takes to complete its execution on the target hardware. The Average-Case Execution Time (ACET) is defined as the time taken by the application between BCET and WCET. The minimal and maximal observed execution times are obtained using some measurement that is less than the actual BCET and WCET, respectively. The actual WCET must be found, or upper bounded so that the WCET estimate must be safe, i.e., guaranteed not to underestimate the real, and tight, i.e., provide an acceptable overestimation of WCET. It should be noted here that the definition of WCET is valid for the given hardware and application. Changing hardware and application results in different WCETs.

1.3 AI on Real-Time Systems

Writing the software algorithm that defines the issue and/or solution to compute the output based on the input and a set of rules is the traditional method of creating control logic for

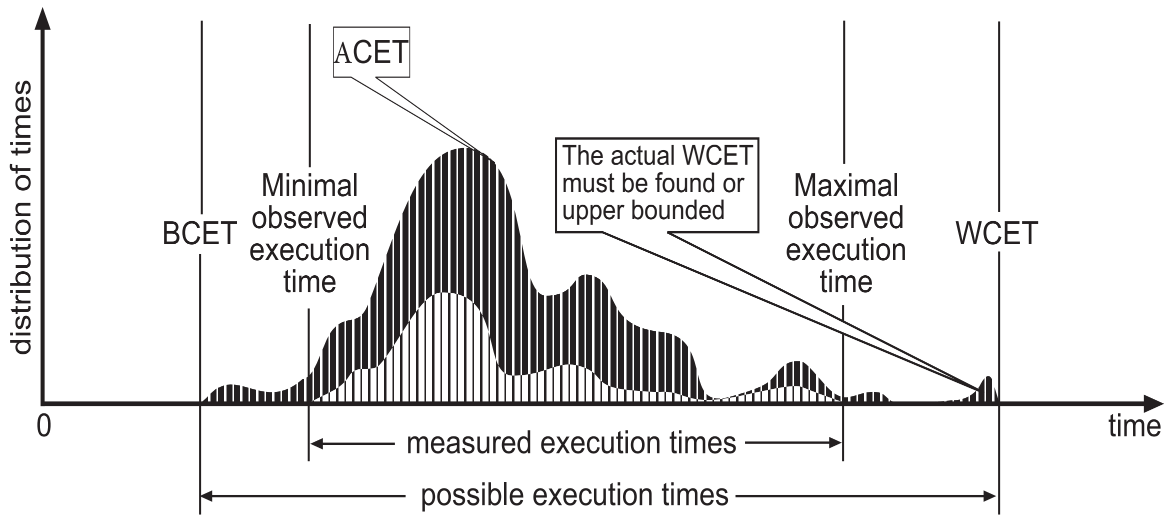


Figure 1.3: Distribution of execution time [145]

embedded systems. This method works incredibly well for systems that have simple, predictable behavior, like a decision tree or the fundamental principles of physics. Nonetheless, in order to build a set of rules that the control logic must adhere to, systems with complex interactions need in-depth data understanding. For instance, identifying unusual patterns in the way a factory assembly machine operates. The controller receives data streams from several sensors for processing. The developer must be aware of the data sets that hint at potential issues or anomalies. These sets will be made up of a complex fusion of several elements, including temperature, manufacturing rate, and vibrations. Identifying these sets is a time-consuming, error-prone, and challenging operation. It is more practical to let the algorithm figure out these data patterns and the accompanying rules on its own. ML is an AI idea where a prediction model is trained on data. On the basis of prior datasets viewed during the training phase, the final model is then capable of making a prediction for a particular input vector. Inference is the name for this procedure.

McCarthy et al. [99] initially proposed the idea of AI in 1955, but it wasn't until recent decades that the field made substantial strides thanks to the development of more potent computing technology, huge datasets, and novel architectures like deep learning. Several research and application fields have emerged as a result of the development of AI. The novel strategy of developing data-driven applications without fully comprehending the complexity of the system raises the issue of how to incorporate these prediction models in many areas, such as CPS and IoT. These interfaces have produced ground-breaking technologies, such as autonomous driving, anomaly detection, predictive maintenance, etc., that were thought to be impractical with traditional programming.

There are several fundamental distinctions between the creation of an ML model and conventional programming logic. The parts/components of a typical ML system are described below.

- **Data Collection:** Gathering relevant data and cleaning the data by handling missing values, removing duplicates, dealing with outliers, and splitting the data into training, validation, and testing sets as input for the model.
- **Model designing and selection:** Determining the appropriate model architecture that suits the best to perform the prediction for the problem dealing with (e.g., classification, regression, clustering).
- **Model Training and Evaluation:** Training the model with a subset of data and evaluating the model performance on the remaining data. Depending on the kind of problem, use the relevant assessment metrics (e.g., accuracy, precision, recall, F1 score, mean squared error).
- **Model Deployment and Monitoring:** Setting up the model for deployment and starting inference on the target platform. Maintaining track of the model's performance in use and, if necessary, modifying or retraining it to improve performance.

TensorFlow [2] and PyTorch [109] are two specialized frameworks that are frequently used while designing ML models. These frameworks provide specialized tools for developing and evaluating ML models as well as optimized implementations for ML architecture operators. To train and execute these models, sophisticated server infrastructure with GPU capabilities is required. We can divide the computational burden and reduce the bandwidth requirement of IoT applications by deploying these models on edge devices, such as IoT sensors, smartphones, etc. [36]. To make the models executable on these devices, however, significant optimization work will be needed. For example, the models must be small enough to fit on the constrained resources of these embedded edge devices or perform quantization on the model's weights and biases to fit them into 8-bit integers rather than 32-bit floats. For instance, Several AR/VR activities, such as hand detection, eye tracking, and digital people, require AI approaches to deliver high-quality interaction in order to bridge the gap between the real world and the virtual one. AI is used in IoT applications for autonomous sensing and reasoning, such as when a "smart home" monitor system is used to identify intruders. Similarly, Individual users can communicate with mobile assistants through voice, pictures, or text. AI is used to identify, comprehend, and interact with users based on their inputs.

1.4 Research Challenges and Questions

Variation in execution time is the major challenge that needs to be addressed by WCET analysis. It occurs because of the characteristic of work the application has to perform on the given hardware. Applications are typically sensitive to their inputs. Consider a railway system, where a safety-critical system controls the trains and ensures they operate safely. The system takes inputs from various sources, such as train schedules, track layouts, and signals. Depending on the input, the system may need to adjust the train's speed or direction to avoid collisions or derailments. For example, suppose the train schedule shows that two trains are approaching a single-track section at the same time. In that case, the safety-critical system must detect the potential collision and assign the trains to slow down or stop to avoid the collision. Similarly, if a signal indicates that a section of track is blocked, the system must instruct the train to stop and wait until the track is clear before proceeding. Thus, the same application can take a different execution time depending on the situation.

Equally crucial is the target hardware that the program runs on. Evidently, a new PC or accelerator runs applications significantly faster than a previous machine. The timing characteristics of the specific hardware on which the program runs must be taken into account by the WCET analysis. In modern processors, throughput is optimized by performance-improving technologies like caches, pipelines, speculation, etc. While these characteristics aim to improve performance on average, they also add execution time unpredictability and make it considerably more challenging to determine a reliable WCET estimate.

The worst-case data which the application will take as input to estimate WCET is just as important. Obviously, knowing the worst-case input for the application is not easy, and generally, people use a heuristic approach. In conclusion, both the hardware and software properties must be considered to understand and predict the WCET of an application. As well as Worst-case data must be known for an application.

According to the discussions in the previous sections and the challenges faced by this thesis, the following research objective is addressed in this thesis:

To ensure the guaranteed timing constraints of RTS, we must thoroughly explore each WCET analysis component.

The following Research Questions (RQ) must be addressed while developing RTS systems, examining RTS applications, and deploying RTS applications to accomplish the above objective.

- **RQ1:** How can we estimate the safe and tight WCET on GPUs architecture that is computational acceptable to perform and still provide a sound bound?

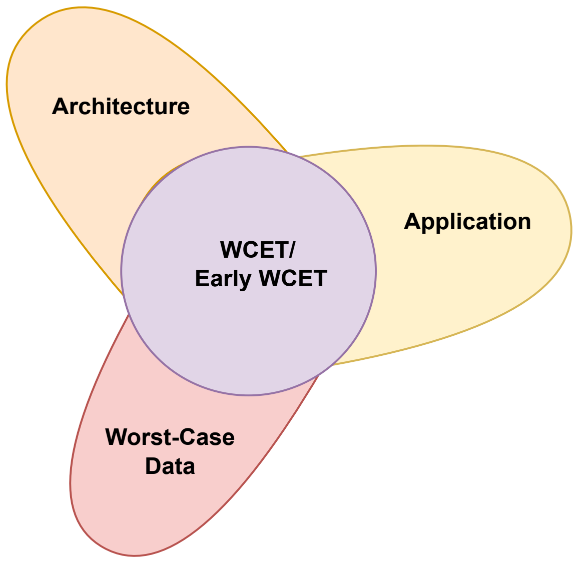


Figure 1.4: Contribution of different component in WCET estimation

- **RQ2:** How can we estimate the optimistic WCET for an application in MC systems having more than two execution time?
- **RQ3:** How can we determine or estimate the safe and tight low WCET for HC jobs in order to enhance QoS (i.e., minimize the amount of LC tasks that are discarded) and control the likelihood of mode switches?
- **RQ4:** How can we estimate the worst-case data of an application which leads to WCET?
- **RQ5:** How can the WCET analysis be used to provide early predictions in the absence of actual platform measurements?

1.5 Contribution to this thesis

The aim of this thesis is to enable AI on RTS in different scenarios. Deploying AI on RTS always targets a trade-off between predictability and correctness. Our goal is to identify and reduce the trade-off in different scenarios. Figure 1.4 shows the major components required for WCET estimation: architecture or platform, application, and worst-case data. In this thesis, we try to improve all these components to make RTS safer and more predictable. The main contributions of this thesis are as follows:

1. First, this thesis proposed a novel ML-based approach to estimate the WCET of the GPU architecture. The model is trained on a large dataset from different benchmarks. The trained model is then used in the testing phase for validation. Our approach shows better results than the state-of-the-art approaches. Our approach also eliminates the code coverage issues that exist in prior techniques. Content of this work has been published in the following publication: [85]

2. Secondly, we focused on estimating the WCET of an application having more than two execution times. This part of the thesis is prevalent in the MC system to find the optimistic WCET of high criticality tasks in lower mode. We propose ESOMICS, a novel method to estimate suitable optimistic WCET using a ML model. Our approach is based on the application's source, and the model is trained based on the large data set. To prove the effectiveness of our approach, we evaluated it with a newly defined metric LTM using an analytical solution that allows us to compute the optimum value in a mixed-criticality system based on experimentation. Our experimental results outperform all the previous state-of-the-art approaches. Content of this work has been published in the following publications: [86, 87]

3. Thirdly, We proposed a method to estimate the worst-case data. Worst-Case Data, which provides the longest execution time, is crucial for WCET estimate. The Worst-Case Data was produced using an evolutionary process, such as the Genetic Algorithm. An evolutionary algorithm involves the utilization of numerous computer resources due to its complexity. With our approach, ML models take the role of the hardware and simulator employed in the evolution process. The amount of time needed to create worst-case data is decreased with this technique. Genetic algorithms are trained to combine with various ML models. Using benchmarks from several areas, the suggested approach's viability is verified. The results show the speedup in the generation of Worst-Case Data. Content of this work has been published in the following publication: [82]

4. Finally, this thesis work also proposed an ML and DNN approach to estimate early WCET. WCET is available to us in the last stage of systems development when the hardware is available and the application code is compiled. Different methodologies measure the WCET, but none give early insights into WCET, which is crucial for system development. Our method using ML and DNNs acts as an approximate predictor model for hardware architecture and compiler. This model predicts the WCET based on the source code without compiling and running on the hardware architecture. The resulting WCET

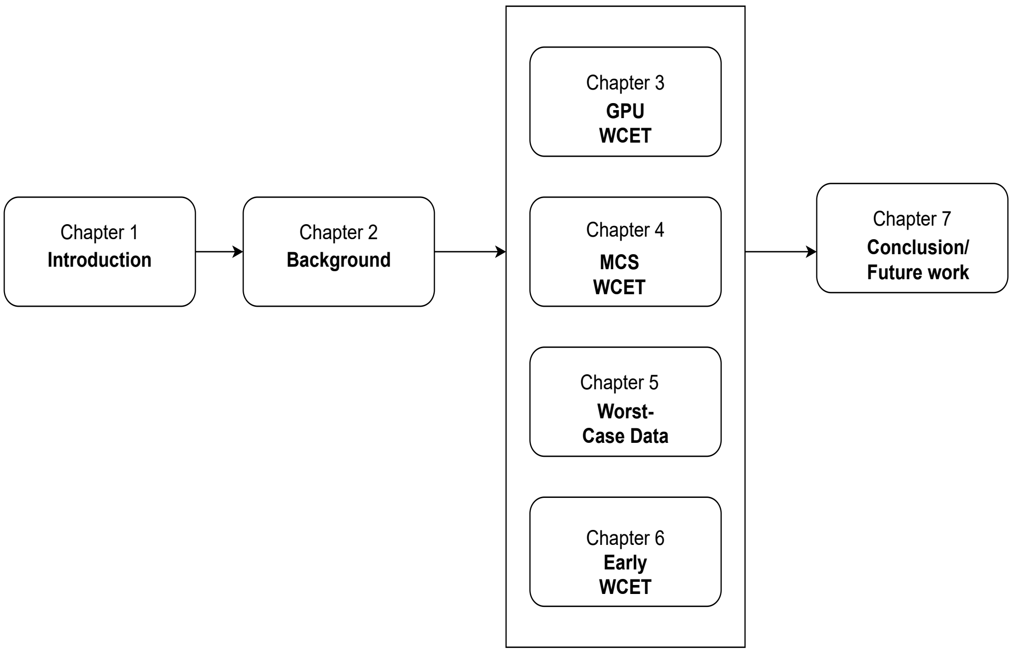


Figure 1.5: Thesis Structure

needs to be revised to be used as an upper bound on the WCET. Content of this work has been published in the following publications: [79, 84]

1.6 Thesis Outline

As shown in Figure 1.5, the thesis is organized as follows: Chapter 1 discusses the introduction and motivation. Chapter 2 gives the background knowledge about different WCET techniques, uses of WCET analysis, description of various WCET tools and AI overview. Chapter 3 presents the ML model for GPU architecture. Chapter 4 discusses the estimation of optimistic WCET in MCS. Chapter 5 talks about the Worst-Case Data estimation techniques. Chapter 6 describes ML and DNN approaches to estimate early WCET. Finally, Chapter 7 summarizes the thesis and discusses future work.

Chapter 2

Background

The previous chapter introduced the trends and issues in RTS design. In designing such systems, all timing constraints need to be satisfied. First, the architecture/platforms, application, and worst data need to be appropriately known to accomplish this work. This chapter mainly introduces the relevant preliminaries and background required to understand the remaining chapter.. This chapter is organized as follows. Section 2.1 presents the different techniques available for WCET estimation, in which a brief introduction of available techniques, such as Measurement-Based, Static, and Hybrid approaches, is presented. Section 2.2 describes the use of WCET analysis in detail. Section 2.3 provides detail about the various WCET tools available in academia and industries. Section 2.4 overview the different AI algorithms briefly, which are used in this thesis. Finally, we summarise the chapter in Section 2.5.

2.1 Different Techniques for WCET Estimation

The distinction between whether the job under consideration is executed or statically evaluated serves as the basis for the general classification of the methodologies for a WCET calculation. The three significant approaches to estimating WCET are explained below.

2.1.1 Measurement-Based Approach

The Measurement-based method [81], [143] executes the task on the given hardware or the simulator for the architecture's different inputs and states (initial and intermediate) to measure the execution time. Different input data sets are applied to measure the maximal program execution time. It is an empirical technique to estimate WCET, which doesn't require complete knowledge of architecture. Figure 2.1 shows the pictorial representation of WCET estimation using a Measurement-based approach. Once the hardware under analysis or a simulator with

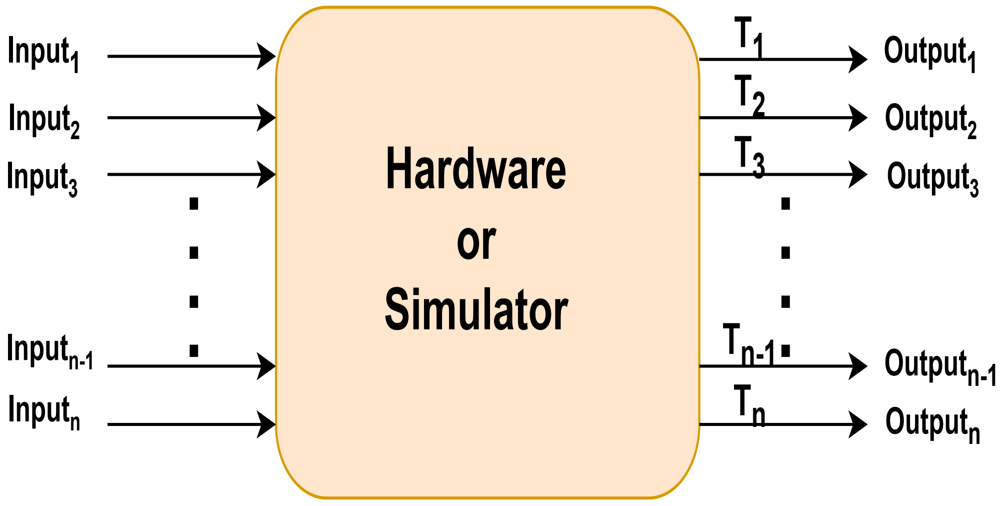


Figure 2.1: Measurement-Based Approach

the application is finalized, various inputs, denoted as Input$_1$, Input$_2$, ..., and Input$_n$, are applied to the system to observe their corresponding execution times, represented as T$_1$, T$_2$, ..., T$_n$. Each input triggers the execution of the application, resulting in the generation of an output. The time taken for the application to process an input and produce the output is recorded. These execution times can vary depending on factors such as input data, hardware characteristics, and system load. In the context of WCET estimation, the input that leads to the longest execution time is of particular interest. This input, denoted as Input$_{n-1}$, takes the maximum time to execute among all inputs. Consequently, the corresponding execution time, T$_{n-1}$, is identified as the WCET for the given application.

The measurement-based approach is the most common technique in the industry because hardware and simulators are usually available. Although there is no assurance that the maximum program run time was recorded, a safety margin is added to the measured execution durations, which frequently results in timing findings that are significantly overestimated.

The main disadvantage of this method is that determining the inputs to be considered for the WCET is not obvious, and running the WCET analysis over the entire set of possible inputs is not feasible. Also, it is frequently necessary to instrument the code for the measurement, for example, by adding instructions to manage hardware timers. However, the validation of safety-critical systems frequently requires that the same code used in the finished product be used.

2.1.2 Static Approach

In the Static approach [39], [81] the application is not executed on the hardware or the simulator. The results from the abstract timing model of the hardware are added to the structural representation of the application program. The primary activities in static methods are creating Control Flow graphs (CFGs), analyzing CFGs, combining CFGs with some abstract models of the target hardware architecture, and estimating upper bounds for WCET. The static technique strongly emphasizes safety and generates execution time boundaries that can be relied upon to never be exceeded by the program being under analysis. To scale down the complexity of an exhaustive analysis of all values, the large number of possible input data is reduced using a safe abstraction. The static approach to WCET estimation, as detailed in Figure 2.2, involves a comprehensive analysis of the program's CFG and micro-architectural characteristics to derive a precise WCET prediction. This approach leverages various techniques such as value analysis, loop-bound analysis, low-level analysis (including cache and pipeline analysis), and path analysis using Integer Linear Programming (ILP) to estimate the WCET accurately.

- CFG generation: The process begins with generating a CFG from the executable code. The CFG represents the program's control flow structure, illustrating how control flows through the program's basic blocks and branches. This representation serves as the basis for subsequent analysis steps.

- Micro-architectural Analysis: It involves three key steps:

  – Value Analyzer: The value analyzer computes value ranges for registers and address ranges for instructions accessing memory. The Value Analyzer identifies infeasible paths by analyzing the possible values that variables can take and the memory accesses performed by instructions. Infeasible paths, where certain conditions or constraints cannot be satisfied, can be excluded from further analysis, focusing efforts on feasible paths that contribute to the WCET.

  – Loop Bound Analysis: Loop bound analysis aims to determine upper bounds for the number of iterations of simple loops in the program. The analysis identifies the maximum number of iterations a loop can execute under any input scenario by analyzing loop structures and loop-invariant properties. This information is crucial for bounding the execution time of loops within the program.

  – Low-Level Analysis/Processor Analysis: The low-level analysis focuses on understanding the micro-architectural effects of the program's execution, particularly on

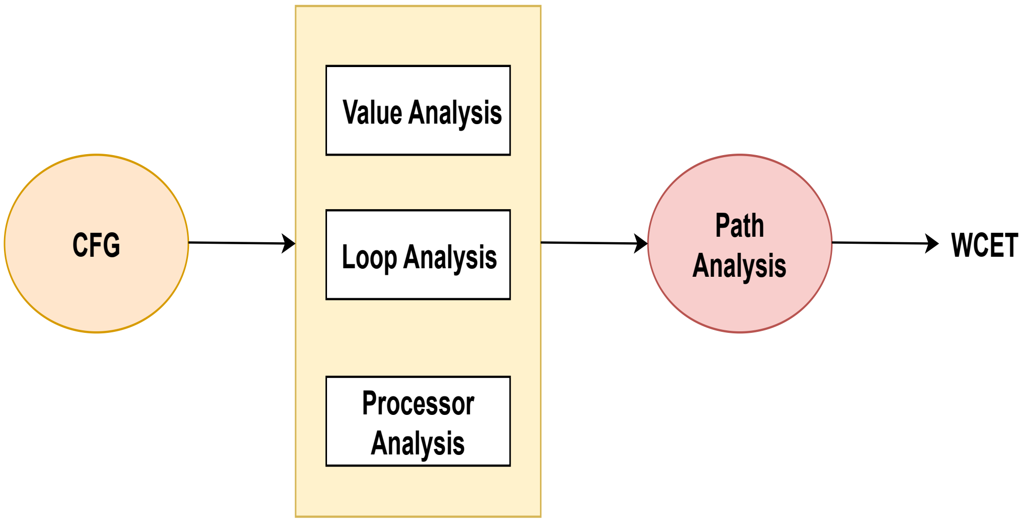


Figure 2.2: Static Approach

processor features like caches and pipelines. This analysis classifies memory references as cache hits or misses and predicts the program's behavior on the processor pipeline. The analysis provides insights into the program's low-level performance characteristics by considering factors such as cache behavior and pipeline stalls due to cache misses.

- Path Analysis: Path analysis involves using ILP to formulate and solve equations representing the program's execution paths. ILP is a mathematical optimization technique for solving linear equations under certain constraints. In the context of WCET estimation, ILP is employed to model the program's execution paths and derive an optimal solution for the WCET. Tools such as CPLEX [9] are utilized to solve the formulated ILP equations and determine the WCET of the program. CPLEX is a widely used ILP-solving tool capable of efficiently handling complex optimization problems. By solving the ILP equations, the tool identifies the worst-case execution path(s) and computes the corresponding WCET value.

Hardware vendors expose the success of the static approach, but in recent years, hardware vendors do not reveal their system features anymore. Additionally, this method becomes complex as the size of the application increases. Another problem of static analysis is that it may result in overestimated outcomes if prudent judgments are made as a result of a lack of knowledge at the time of the study. The scalability and complexity of the approach to the increase of code size are significant issues in static analysis.

2.1.3 Hybrid Approach

Combining ideas from measurement-based and static techniques is the rationale behind hybrid approaches [113]. The hybrid technique finds what is known as Single Feasible Pathways (SFPs), or program routes made up of a series of fundamental building pieces where the execution is independent of the input data. Using symbolic analysis on abstract syntax trees, SFPs may be found at the source code level. The SFPs' execution times are then measured on actual hardware or using cycle-accurate simulators in the following stage. Data input for a full branch coverage must be provided for input-dependent branches. Measurements are also used to estimate the execution duration of these components. An extra safety margin is provided to the measured execution time to account for any measurement underestimate. In order to find the longest path, the data from the SFPs are merged with methods from the static approach. FIGURE 2.3 shows the component involved in the Hybrid approach. It is a sophisticated approach that blends three distinct techniques to provide a balanced and accurate assessment of a program's WCET. Firstly, recognizing the inherent limitations of purely static analysis, hybrid approaches incorporate online testing to measure the execution time of short sub-paths within the code directly. By leveraging the actual processor's behavior through online testing, these techniques capture real-world execution dynamics, especially between decision points in the code. This approach ensures a more realistic understanding of execution times, mitigating the overly pessimistic estimates often associated with static analysis alone.

Secondly, hybrid methodologies enhance offline analysis by incorporating insights obtained from testing sessions. This integration enables the extraction of valuable information, such as the number of loop iterations and execution frequencies during modal operation. By incorporating this empirical data, hybrid approaches construct a nuanced model of the code's structure, identifying feasible paths through the program. This hybridization allows for a more comprehensive analysis, enabling the determination of complete and viable execution paths, which may not be fully captured through static analysis alone.

Finally, hybrid WCET estimation combines measurement and path analysis data to compute WCET that accurately reflects the variability introduced by hardware effects. By integrating information from both measurement and analysis phases, this approach captures the nuances of execution time variation across individual paths due to hardware intricacies. This holistic perspective ensures that the resulting WCET values balance the overly pessimistic estimates of static analysis and the potentially optimistic values obtained solely through measurement.

This method does not require an abstract architecture model compared to the static method. However, instrumented code is needed, which may not be allowed in all cases, and correct WCET

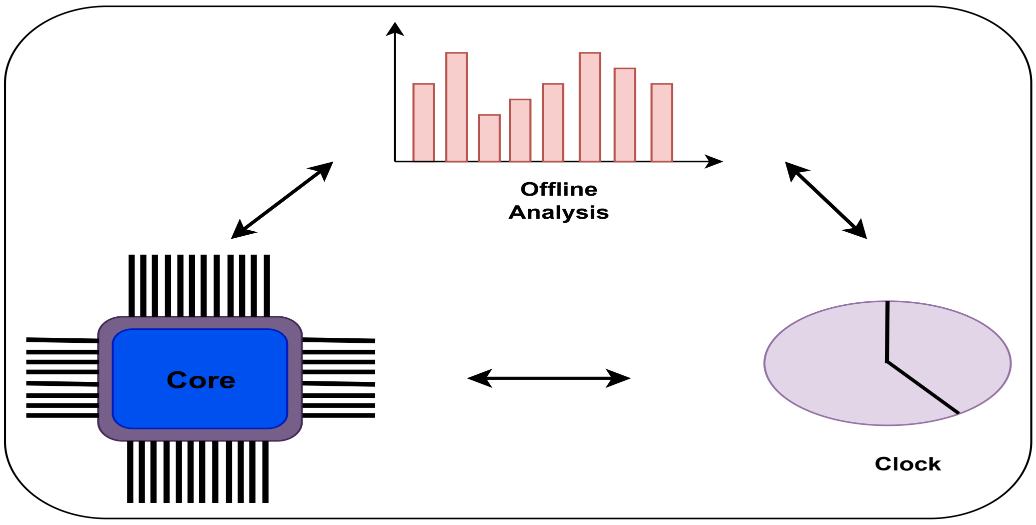


Figure 2.3: Hybrid Approach

is not possible because a safe initial state and worst-case input can not be assumed.

2.2 Uses of WCET Analysis

The major use of WCET is in the development and analysis of RTS. In such systems, scheduling, and schedulability analysis are performed using estimated WCET, giving timing guarantees for the system's behavior as a whole and deciding whether or not time restrictions can be satisfied for specific activities. WCET analysis is a natural method to use in any product development when timeliness is critical, but it has a much wider application scope. Some of the main use of WCET analysis in detail:

- **Timing analysis:** For RTSs, timing analysis is essential. It needs to perform tasks within strict timing constraints. For example, a medical device may need to deliver medication within a specific time window, or an automotive system may need to brake within a certain distance. WCET analysis provides a precise measurement of the execution time of a program, which can be used to determine if the program meets its timing requirements. This analysis involves measuring the execution time of each instruction and identifying the longest execution path, which represents the worst-case scenario.

- **Code optimization:** WCET analysis can help identify performance bottlenecks in a program. By analyzing the execution time of each instruction, developers can identify which parts of the code take the most time to execute. This information can be used to optimize the code, either by re-writing it to be more efficient or by using compiler optimizations. By optimizing the code, the execution time of a program can be reduced, which can increase system performance.

- **Hardware design:** WCET analysis can be used to design hardware platforms that meet the timing requirements of a given program. By analyzing the WCET of a program on different hardware platforms, designers can select the optimal platform that meets the timing requirements. This analysis can also be used to design custom hardware accelerators that can perform specific tasks more efficiently than a general-purpose CPU.

- **System verification:** WCET analysis can be used to verify the correctness of a real-time system. By verifying that a system meets its timing requirements, developers can ensure that the system operates correctly under all possible scenarios. This analysis involves running the system under different scenarios and measuring the execution time of each task. If the WCET of each task is within the timing constraints, then the system is considered to be correct.

- **Safety-critical systems:** In safety-critical systems, failure can result in catastrophic consequences. WCET analysis is essential in these systems to ensure that the system meets its timing requirements and operates safely under all possible scenarios. For example, in an automotive system, if the braking system does not engage within a certain distance, the consequences could be severe. By analyzing the WCET of the braking system, developers can ensure that it meets the timing requirements and operates safely.

Up till now, only robotics, avionics, automotive, space, and the automotive industry have used WCET analysis in practice. As many of their products contain resource-constrained embedded safety-critical RTSs, it is likely that these sectors will lead all other sectors in embracing WCET analysis.

2.3 WCET Tools

The research in timinig analysis has been done for the last two to three decades, which led to the development of numerous WCET tools both in academia and industries because of their vast application in many domains. Some of the tools for WCET analysis have been described below:

- **SWEET:** SWEdish Execution Time Analysis Tool is the abbreviation for the term " SWEET" [95]. In ALF format, SWEET analyzes programs. ALF stands for Artist Flow Analysis language; it is a general intermediate program language format, especially developed for flow analysis. A number of translators are available, and ALF code may be created from a variety of sources, including C and assembly code. The main function of SWEET is flow analysis. The result of flow analysis is flow facts, i.e., information about loop bounds and infeasible paths in the program. Flow facts are necessary for finding a safe and tight WCET for the analyzed program. SWEET may also immediately provide BCET/WCET estimations if a timing model is present, eliminating the requirement to compute flow facts. SWEET has been developed by the WCET research team in Västerås, Sweden, since 2001.

- **OTAWA:** OTAWA [13] stands for Open Tool for Adaptative WCET Analysis. It is an open framework created specifically for the experimental static analysis-based WCET calculation. To fulfill aims of simple extensibility and unrestricted openness, its design has taken advantage of several potent characteristics from successful generic tools already in existence, such as Salto or SUIF. As a consequence, we have a tool that represents the program using an abstract architecture layer, with various analyses being carried out using so-called code processors that are arranged in chains and store and apply annotations on the abstract architecture layer. It includes several analysis tools, such as WCET Analyzer and Timing Analyzer, that can be used to generate upper bounds on the WCET.

- **Heptane:** Open source software called Heptane [57] is distributed with the GNU General Public License version 32. The C++-based research prototype Heptane currently supports the MIPS and ARM v7 instruction sets and was constructed with about 13,000 lines of code. To ensure that the tool builds correctly and runs non-regression tests for the supported target processors and host operating systems, we use a continuous integration framework in particular. During the first tutorial on tools for real-time systems, heptane was used as an example. Heptane's goal is to provide upper limits for application execution times. It aims at applications with strict real-time demands (automotive, railway, and aerospace domains). Static analysis at the binary code level is used by Heptane to calculate WCETs. Micro-architectural features like caches and cache hierarchies are subject to static assessments. The first version of Heptane was developed in the late nineties.

- **Chronos:** Chronos [93] is an open-source WCET analysis tool for real-time embedded software. It takes as input the program binary, disassembles it, and performs static analysis on the assembly code. Static analysis involves program flow analysis as well as micro-architectural modeling. The user can change the micro-architectural analysis routines to model new processor platforms. It is built on top of the popular Simplescalar architectural simulator. Simplescalar allows the user to flexibly model different architectures for simulation.

- **aiT:** To determine upper boundaries for the execution timings of code fragments (such as those provided as subroutines) in executables, AbsInt created the timing-analysis tool aiT [39]. These code fragments may be scheduler-called tasks in a real-time application, where each job has a deadline. Because the source code lacks information on register utilization, instruction, and data advertisements, aiT only works on executables. When there are several memory locations with various timing characteristics, such addresses are crucial for cache analysis and the timing of memory accesses. aiT relies on the standard calling convention. If some code does not adhere to the calling convention, the user might need to supply additional annotations describing the control-flow properties of the task.

- **Bound-T:** The European Space Agency (ESA) originally contracted Space Systems Finland Ltd. to build the Bound-T [61] tool, which was designed to verify onboard software in spacecraft. The tool determines an upper bound on the execution time of a subroutine, including called functions. Optionally, the tool can also determine an upper bound on the stack usage of the subroutine, including called functions. The input is a binary executable program that typically has a symbol table contained in it (debug information). In some counter-based loops, the tool can compute upper limits. The user enters annotations for additional loops, known as assertions in Bound-T. Variable values may also be annotated in order to enable the automated loop bounds. Annotations are written in a separate text file, not embedded in the source code. The implicit path enumeration technique applied to the CFG of the subroutine finds the worst-case path and the upper bound for one subroutine.

- **RapiTime:** RapiTime is a robust timing and WCET analysis tool created especially for embedded targets and to meet certification standards. You may utilize the timing measurements generated by RapiTime to show that you've met the DO-178B/C objectives. RapiTime combines static and dynamic analysis of your code on target to provide detailed information on its timing behavior. RapiTime enables you to automate timing analysis on your embedded system to spot timing problems early in the development process and improve your code, saving you money down the road. The input of RapiTime is either a set of source files (C or Ada) or an executable. The user must also provide test data from which measurements will be taken. The output is a browsable HTML report describing the WCET prediction and actual measured execution times, split for each function and subfunction. RapiTime's minimal overhead means you can perform timing analysis in every test run, making timing information available throughout your development process. This information will help you identify timing issues early in development and minimize WCET.

As we can realize, a number of tools have been designed in the last two decades in both academia and industries to estimate WCET safely and tightly. We also observed from the structure of each tool that the estimation of WCET requires target architecture or hardware, the compiled binary, and the worst-case data. Most of the tools described above need the executable of the application to estimate WCET, and none of the above tools provide WCET in the early stage. As well as, setting the environment, which leads to WCET for the particular application also not feasible.

To overcome all these, we proposed an AI-based WCET estimation in the thesis that doesn't require the need for the target architecture or platform, the compiled binary, and worst-case data. Our approach also doesn't need to set the environment all the time. The thesis describes the proposed method using AI based on two phases. Firstly, Our approach needs to be trained using the dataset of the particular architecture to learn the timing behavior of the architecture. During the training, some of the tools mentioned above were used to verify our approaches' correctness and result. Heavy and intensive computations are involved in this part and are done once for each architecture. Secondly, the approaches are tested to evaluate their accuracy using different benchmarks.

2.4 Brief Overview of AI Approaches

ML is a branch of AI that gives computers the ability to learn and develop on their own, without having to be explicitly programmed for each job. It is predicated on the notion that computers can recognize patterns and insights in data and apply these discoveries to generate predictions or conduct actions based on brand-new, unforeseen data. By the use of ML, computers are now able to solve challenging issues, spot patterns, and make precise predictions. The reason for using ML models rather than Deep Learning (DL) models is interpretability, a significant concern in Safety-Critical Systems. The DL models are like black boxes, and we do not know what is happening in each layer, so we restrict ourselves to using only those models that are interpretable. ML algorithms can be categorized into three broad types: supervised learning, unsupervised learning, and reinforcement learning.

- **Supervised Learning:** The most prevalent kind of ML is supervised learning [21].

With this method, a dataset made up of input data and related output data is sent to the computer. The output data is referred to as the label, whereas the input data is referred to as features. The goal of supervised learning is to develop a mapping function that can forecast the intended result from fresh input information. It can be further divided into regression and classification. Regression is used when the output variable is continuous, such as predicting a stock price. Classification is used when the output variable is categorical, such as predicting whether an email is a spam or not. Some popular supervised learning algorithms include linear regression, logistic regression, decision trees, support vector machines, and neural networks.

- **Unsupervised Learning:** When there are no labeled outputs in the data, unsupervised learning is performed [59]. With this method, a dataset is provided to the computer, which then analyzes the dataset in search of patterns and linkages. Clustering, Anomaly Detection, and Dimensionality Reduction may all be accomplished with unsupervised learning. Clustering is the process of grouping similar data points together, while anomaly detection is identifying data points that are significantly different from the rest. Dimensionality reduction is reducing the number of features in a dataset while retaining as much useful information as possible. Some popular unsupervised learning algorithms include k-means clustering, hierarchical clustering, principal component analysis, and autoencoders.

- **Reinforcement Learning:** Reinforcement learning [134] is a type of ML that involves an agent interacting with an environment to learn how to make decisions that maximize a reward signal. The agent receives feedback in the form of a reward or penalty for each action it takes, and its goal is to learn the optimal policy that maximizes the total reward over time. It can be used in robotics, game-playing, and autonomous driving. Some popular reinforcement learning algorithms include Q-learning, policy gradient methods, and actor-critic methods. In addition to the three main types of ML, there are also other types, such as semi-supervised, transfer, and ensemble learning which is described below:

  - **Semi-supervised:** It is a machine learning paradigm where the model is trained on a dataset that contains both labeled and unlabeled data. In contrast to supervised learning, where every data point in the training set has a corresponding label, semi-supervised learning leverages the additional information provided by the unlabeled data to improve the model's performance. It leverages the abundance of unlabeled data, which may be readily available or cheaper to acquire compared to labeled data.

  - **Transfer Learning:** It is a machine learning technique where a model trained on one task is reused or adapted as the starting point for a model on a second related task. It leverages the knowledge gained from the source task to improve learning on the target task, particularly when the target task has limited labeled data or computational resources. Transfer learning has become increasingly popular, especially in deep learning, due to its ability to boost performance and reduce training time on new tasks. It enables faster convergence and reduces the amount of data required for training on the target task, as the model starts with pre-learned features or knowledge.

- **Ensemble Learning:** It is a machine learning technique that combines the predictions from multiple individual models to improve the overall performance. The basic idea is that by aggregating the predictions of multiple models, you can often achieve better results than any single model could achieve on its own. The two most widely used ensemble learning are Boosting and Bagging. This method helps to reduce the overfitting and is more robust to outliers and noise in the data. It can also be used to detect anomalies in data.

In this thesis supervised learning has been used. In-depth knowledge of the prediction model of supervised learning is not required to understand the experiments presented. However, for the sake of completeness, a brief description of each ML model is given in the following.

- **Linear Regression:** A straightforward and widely used approach for regression tasks—tasks that require predicting a continuous output variable based on one or more input features—is linear regression [58]. Finding a linear connection between the input characteristics and the output variable is the aim of linear regression. The equation for a simple linear regression model with one input feature can be written as follows:

$$y = \beta_0 + \beta_1 * x + \epsilon \tag{2.1}$$

  where :

  - y is the dependent variable or output variable that we want to predict
  - x is the independent variable or input feature that we use to make predictions
  - β_0 is the intercept, which represents the predicted value of y when x is equal to zero
  - β_1 is the slope or coefficient, which represents the change in y for a unit change in x
  - ϵ is the error term or residual, which represents the difference between the predicted value of y and the actual value of y

The objective of linear regression is to find the values of β_0 and β_1 that minimize the sum of squared errors between the predicted values and the actual values. This is commonly referred to as the ordinary least squares method. To find the values of β_0 and β_1, we use a training dataset that consists of pairs of input features and output values. We then calculate the values of β_0 and β_1 that minimize the sum of squared errors between the predicted values and the actual values. This can be done using a variety of optimization techniques, such as gradient descent. Once we have trained the linear regression model, we can use it to make predictions on new, unseen data by plugging in the values of the input features into the equation and calculating the predicted value of y. Linear regression can be extended to include multiple input features by using a multiple linear regression model. The equation for a multiple linear regression model with n input features can be written as follows:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n * x_n + \epsilon \tag{2.2}$$

Where $x_1, x_2, ..., x_n$ are the n input features and $\beta_1, \beta_2, ..., \beta_n$ are the coefficients or slopes for each input feature.

- **Polynomial Regression:** The link between the independent variable x and the dependent variable y is modelled as an nth-degree polynomial function of x in polynomial regression [111], a kind of regression analysis. In comparison to a linear function, the polynomial function may capture nonlinear interactions between the variables and offer a better fit to the data.

  The general form of the polynomial regression model with degree n can be written as:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + ... + \beta_n x^n + \epsilon \tag{2.3}$$

  where y is the dependent variable, x is the independent variable, β_0, β_1, β_2, ..., β_n are the coefficients to be estimated, ϵ is the error term, and n is the degree of the polynomial. The polynomial function can be expressed as a linear combination of powers of x, and the coefficients can be estimated using linear regression.

  It is crucial to remember that if the degree of the polynomial is too high, the polynomial function may overfit the data and perform poorly on fresh data. To balance the bias-variance trade-off, it is crucial to select a polynomial degree that is adequate. The choice of the degree of the polynomial can be based on visual inspection of the data, or by using statistical measures such as the adjusted R-squared, the Root Mean Square Error (RMSE), or cross-validation. In general, a lower degree polynomial is preferred to prevent overfitting, unless there is strong evidence that a higher degree polynomial is necessary to capture the underlying relationship between the variables.

- **Ridge Regression:** In order to avoid overfitting, ridge regression [33], a sort of regularised linear regression, adds a penalty term to the Ordinary Least Squares (OLS) objective function. By adding a restriction on the size of the coefficients, the penalty term decreases the complexity of the model and may enhance generalisation ability. The ridge regression objective function with an L^2 penalty term can be written as:

  $$\min_{\beta} ||y - X\beta||^2 + \alpha||\beta||^2 \tag{2.4}$$

  Where y is the vector of the dependent variable, X is the matrix of the independent variables, β is the vector of the coefficients to be estimated, and α is the regularization parameter that controls the strength of the penalty term. The first term is the OLS objective function, which minimizes the sum of squared errors between the predicted and actual values. The second term is the L^2 penalty term, which adds a penalty proportional to the square of the magnitude of the coefficients. When there is multicollinearity between the features, which can result in unstable and overfitting models in OLS regression, or when the number of features exceeds the number of data, ridge regression is very helpful. In ridge regression, the L^2 penalty term stabilises the estimates and lowers the variance of the coefficients, which can enhance the model's performance. The regularization parameter α controls the trade-off between the fit to the training data and the complexity of the model.

- **Decision Tree:** Decision trees [118] are a popular algorithm for both classification and regression tasks. The main idea behind decision trees is to recursively split the data into smaller and smaller subsets based on the values of the input features until a stopping criterion is met. The result is a tree-like structure where the internal nodes represent tests on the input features, and the leaf nodes represent the predicted output values. The algorithm selects the best feature to split the data on at each node. This is done by calculating a metric such as Information Gain or Gini impurity, which measures the homogeneity of the target variable within each subset. The feature with the highest metric is chosen to split the data, and the process is repeated recursively for each subset until the stopping criterion is met.

  The stopping criterion can be based on various factors, such as the maximum depth of the tree, the minimum number of samples required to split a node or the minimum improvement in the metric. These parameters can be tuned to control the complexity of the tree and avoid overfitting. Decision trees can handle both categorical and numerical input features, and they can also handle missing data by imputing the missing values or treating them as a separate category. One disadvantage of decision trees is that they can be sensitive to small changes in the data and tend to overfit when the tree becomes too complex.

- **Support Vector Regression :** Support Vector Regression (SVR) [130] is a variant of Support Vector Machine (SVM) that is used for regression tasks. The main idea behind SVR is to find a hyperplane that best fits the data while minimizing the margin violations. Given a training set of n data points with m input features and corresponding target values, SVR tries to find the optimal hyperplane that best fits the data subject to the constraint that the deviations from the target values are within a certain margin.

$$\min_{w,b,\epsilon} 1/2||w||^2 + C\sum_{i=1}^{n}(\epsilon_i + \epsilon_i^*) \tag{2.5}$$

subject to the constraints:

$$y_i - \langle w, \phi(x_i)\rangle - b \leq \epsilon_i \langle w, \phi(x_i)\rangle + b - y_i \qquad \epsilon_i, \epsilon_i^* \geq 0$$

where y_i is the target value for the i-th data point, x_i is the i-th input value, w and b are the coefficients of the hyperplane, ϕ is the non-linear mapping function, C is a regularization parameter that controls the trade-off between the margin and the deviation, and ϵ_i and ϵ_i^* are slack variables that measure the deviation from the margin on either side of the hyperplane.

- **Random Forest :** Several decision trees are combined in Random Forest [24], an ensemble learning approach, to increase the predictors' robustness and accuracy. The method builds distinct decision trees using subsets of the data and characteristics that are randomly chosen, then aggregates their forecasts to provide a final prediction. Both classification and regression tasks, high-dimensional data, non-linearly separable data, missing values, and outliers may be handled using the technique. The main parameters of the algorithm are n_estimators (the number of decision trees in the forest), max_depth (the maximum depth of each decision tree), min_samples_split (the minimum number of samples required to split an internal node), min_samples_leaf (the minimum number of samples required to be at a leaf node), and max_features (the maximum number of features to consider when splitting a node). The decision tree in Random Forest is a binary tree that partitions the data into subsets based on the values of the features, and the prediction is obtained by aggregating the predictions of the decision trees.

The main advantages of Random Forest are:

- It can handle both classification and regression tasks.
- It can handle high-dimensional data with many features.
- It can handle non-linearly separable data and interactions between features.

- **Gradient Boosting :** An ensemble learning approach called gradient boosting [42] combines a number of weak prediction models to create a strong prediction model. The approach is based on the boosting principle, which entails adding weak models to the ensemble successively in order to increase its accuracy. In order to gradually enhance the model's predictions, gradient boosting works by fitting a succession of decision trees to the residuals (i.e., the difference between the real values and the forecasts) of the prior trees. The algorithm can handle both regression and classification tasks, non-linearly separable data, missing values, and outliers. The main parameters of the algorithm are n estimators (the number of trees in the ensemble), max depth (the maximum depth of each tree), learning rate (the step size of the gradient descent), subsample (the fraction of the training data to use for each tree), and loss function(the function to optimize during training, such as mean squared error or cross-entropy).

  The prediction of gradient boosting is obtained by aggregating the predictions of the decision trees, weighted by the learning rate and the performance of the trees. Specifically, the final prediction is given by:

$$y = \sum (\alpha + f(x)) \tag{2.6}$$

  where y is the predicted value, α is the weight of the tree, f(x) is the prediction of the tree for the input x, and $\sum$ is the sum of all the trees in the ensemble. The strength of gradient boosting comes from the ability to iteratively improve the model by fitting trees to the residuals of the previous trees. This allows the GB to gradually capture the complex interactions between features and the target variable. The learning rate controls the step size of the gradient descent, and the subsampling control the variance of the model. The choice of loss function depends on the task and the type of data.

- **Adaptive Boosting :** Adaptive Boosting (AdaBoost) [41] is a boosting algorithm that adapts the weights of the training samples based on their previous performance. The basic idea behind AdaBoost is to train a series of weak models on the weighted samples and then combine their predictions to form a strong model. At each iteration, the algorithm increases the weight of the misclassified samples and decreases the weight of the correctly classified samples in order to focus on the difficult samples. The prediction of AdaBoost is obtained by aggregating the predictions of the weak models, weighted by their respective weights. Specifically, the final prediction is given by:

$$y = sign \sum (\alpha + f(x)) \tag{2.7}$$

Where y is the predicted value, α is the weight of the weak model, f(x) is the prediction of the weak model for the input x, and $\sum$ is the sum of all the weak models in the ensemble. The sign function is used for classification tasks to convert the weighted sum of the predictions into a binary decision. The strength of AdaBoost comes from the adaptive weight update, which focuses on the difficult samples and gradually improves the performance of the model. The learning rate controls the step size of the weight update, and the choice of loss function depends on the task and the type of data.

The difference between Gradient and Adaptive Boosting is using different loss functions (Exponential in GB and MSE or Squared loss in AdaBoost) and learning rates (variable in AdaBoost and fixed in GB).

- **Elastic Net :** It is a linear regression model that combines the L1 and L2 regularization techniques to overcome their limitations and improve the performance of the model [151]. The L1 regularization (Lasso penalty) encourages sparse solutions by penalizing the absolute values of the coefficients, while the L2 regularization (Ridge penalty) encourages smooth solutions by penalizing the squared values of the coefficients. The Elastic Net combines both penalties by introducing a hyperparameter alpha that controls the trade-off between the two penalties. The Elastic Net model can be formulated as follows:

$$min(1/2 * ||y - X\beta||^2 + alpha * ||\beta||_1 + (1 - alpha) * ||\beta||_2^2) \tag{2.8}$$

where:

  - y is the dependent variable
  - X is the design matrix

  - β is the coefficient vector
  - α is the elastic net mixing parameter, which controls the relative weight of the L^1 and L^2 penalties
  - $||\beta||_1$ is the L^1 norm of the coefficient vector
  - $||\beta||_2^2$ is the L^2 norm of the coefficient vector

  It is noteworthy that Elastic Net also has certain disadvantages, including possibly higher computing costs and the requirement to adjust more hyperparameters compared to Ridge or Lasso regression.

- **Multi-Layer Perceptron (MLP):** MLP [125] is a type of artificial neural network that consists of multiple layers of interconnected nodes or neurons, each layer performing a non-linear transformation of its inputs. MLP is a feedforward neural network, meaning that the information flows only in one direction, from the input layer to the output layer. The MLP model can be represented by the following equation:

$$y = f(W2 * g(W1 * x + b1) + b2) + \epsilon \tag{2.9}$$

  where y is the dependent variable to be predicted, x is a vector of independent variables or features, f is the activation function of the output layer, g is the activation function of the hidden layer, $W1$ and $W2$ are the weight matrices that determine the strengths of the connections between the neurons, $b1$, and $b2$ are the bias vectors that determine the offsets of the activation functions, and ϵ is the error term or residual that captures the unexplained variation in y. The hidden layer is the layer of neurons that is sandwiched between the input layer and the output layer, and its purpose is to extract and transform the features of the input data into a new representation that is more suitable for the prediction task. The number of neurons in the hidden layer is a hyperparameter that needs to be chosen based on the complexity of the problem and the size of the dataset. The activation functions f, and g are nonlinear functions that introduce non-linearity into the MLP model, allowing it to capture complex and nonlinear relationships between the inputs and outputs. Some commonly used activation functions are the Sigmoid function, the Hyperbolic Tangent function, and the Rectified Linear Unit (ReLU) function. The weight matrices and bias vectors are the learnable parameters of the MLP model, which are updated during the training process using an optimization algorithm such as back-propagation. The goal of the training process is to minimize the loss function, which measures the discrepancy between the predicted values and the actual values of the dependent variable. The loss function can be chosen based on the regression metric and the nature of the data, such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and log-likelihood. The MLP model can be further improved by regularization techniques, such as L1 or L2 regularization, which penalize the magnitude of the weight matrices and bias vectors for preventing overfitting and improving generalization performance. Dropout [131] is another technique that randomly drops out some neurons during training to reduce co-adaptation and increase robustness.

2.5 Summary

In this chapter, we take a closer look at three different techniques for WCET estimation: Measurement, Static and Hybrid approaches. The choice of technique is a trade-off between tight and safe WCET. The use of WCET analysis is explained in detail to understand its various application domains. We discussed the importance of various WCET tools available in academia and industries. None of the tools mentioned in the previous section is capable of providing insights into early WCET. To determine the WCET with precision in this thesis, we proposed AI approaches and different AI approaches were explained in detail to understand their use in the remaining chapters. The above-mentioned WCET analysis techniques, WCET tools, and AI approaches will be used widely in the following chapters.

Chapter 3

Estimation of WCET on GPU

This chapter summarizes our contributions, emphasizing real-time systems on GPU architectures with strict timing guarantees. As a result, system execution must ensure that tasks are finished before deadlines to meet each task's latency needs. More precisely, Section 3.1 presents the introduction and context of GPU architecture in real-time systems and its increasing popularity in safety-critical real-time systems such as autonomous vehicles to provide massive parallelism for workloads. Section 3.2 describes the GPU background in detail with its hardware, execution, and programming model. Section 3.3 presents our contributions regarding the timing model that allows multiple kernel interference to occur and estimates the WCET of GPU kernels using the ML model. The experiment setup and its evaluation with real-life benchmarks and synthetic task sets are shown in Sections 3.4 and 3.5, respectively. Section 3.6 summarises the contributions mentioned above.

3.1 Introduction

With the advances in Machine Learning (ML) and Artificial Intelligence (AI) in every field of life [64], due to their tendency to solve many problems with accuracy, it requires Graphics Processing Units (GPUs) to provide massive parallelism. GPUs are commonly used as an accelerator to exploit compute-intensive workloads in various application domains, such as autonomous vehicles, industrial robotics, and avionics. For instance, the autonomous driving domain uses ML and CV workloads to process visual data about the surroundings to find pedestrians and objects [141] and to carry out driving safety protocols [104],[137]. GPUs are designed to provide high-performance throughput, but their integration into real-time systems focuses on predictability because most safety-critical applications have strict deadlines that need to be followed to avoid unwanted situations. The increasing use of GPUs [112] in real-time embedded systems provides severe challenges in developing methods to find the Worst-Case Execution Time (WCET) of the GPU kernel.

Traditional WCET analysis techniques [145] do not apply to GPUs directly because of the multiple threads competing for the same resources. As in the CPU case, multi-core timing analysis is challenging to estimate with all the known methods and approaches. These challenges gradually increase when switching to GPUs, considering the interference from different Streaming Multiprocessors (SMs) and Communication Channels. Previous research works have been done to guarantee the timing constraints for GPU applications using measurement and static approaches. However, these approaches have their known issues. In the measurement-based approach, the availability of the simulator or hardware to estimate WCET is not guaranteed because of not knowing the initial architecture state and input. The scalability and complexity of the approach to the increase of code size are significant issues in static analysis.

With each new generation and architecture, GPUs are becoming powerful and have thousands of cores. Usually, one real-time application can not use an entire GPU, while multiple applications can be benefited from it. GPUs must deliver predictable application performance even in worst-case circumstances for real-time applications since they must meet rigorous deadlines, especially for safety-critical workloads. To overcome this, Nvidia provides Multi-Process Service (MPS) [105], allowing multiple applications to co-run on GPU. Previous works [69, 126, 128] have demonstrated that conflicts in the memory hierarchy (mostly in shared cache and DRAM) may nevertheless affect the runtime of two applications executing on different GPU cores due to the following reasons: (i) Cache set conflicts in the shared cache, (ii) DRAM bus contention, and (iii) Row buffer conflicts in DRAM.

Integrating GPUs into real-time embedded systems is a complex process that involves many steps. The selection and configuration of hardware and the identification and mapping of kernel functions to SMs are important decisions that must be made during the software Design and Development Phases (DDP), which come before the validation phase. The timing dimension influences a significant portion of those decisions because the final system's accuracy ultimately depends on the ability of the implemented functions to complete within the specified time budgets. To avoid timing misbehavior being caught at the timing analysis stage in the advanced development phases, where changing design decisions is very difficult, early execution time numbers already in the DDP are required. The variation in execution times due to multiple factors, such as thousands of threads competing for the same computational resources and contention from different memory hierarchies, compounds this situation. However, all the state-of-the-art techniques cannot precisely provide WCET for the GPU kernel.

In this chapter, we propose an ML approach to estimate the WCET of the GPU kernel from the binary of the applications. The approach operates in two phases. In the first phase, the timing model of GPU is learned using the ML model. Our approach comes with five different ML models that are the best prediction of WCET. Learning is performed using comprehensive measurement of kernel code. The WCET of each kernel is learned in the context of interference through the enemy kernel. One benefit of our approach is that it avoids the problem of the lack of public knowledge about the properties of the memory subsystem. The training phase is executed only once per architecture. In the second phase, the WCET of the unseen kernel is estimated using the timing model learned in the first phase for the interference suffered by the kernel. This phase is executed once for each target program.

We believe our approach can be used for software in the safety-critical system, which uses GPU and accelerator to accelerate the compute-intensive work and to guarantee timing predictability. As WCETs are required for these systems, some pessimism in WCET estimation is acceptable, but missing a deadline, if it happens infrequently enough, can be tolerated. We have evaluated our approach on the NVIDIA GPU RTX3060, for which some information, such as scheduling policy, is not publicly available due to intellectual property. Our experiments evaluate the WCET of 10 kernel programs from different benchmarks, such as Basic Compute Unified Device Architecture (CUDA) Samples, CUDA Imaging Samples, and the Tango DNN suite. Our approach is an empirical technique that doesn't always guarantee a safe WCET estimate. However, for the tested kernel, the estimated WCET is always larger than the actual WCET. Our approach has two advantages over the existing hybrid technique. First, measurements, which are time-consuming, are performed only once per architecture to train the ML algorithms. Estimating the WCET of a target kernel is fast based on our experimental evaluation. Second, since the WCETs of the kernel of a program are estimated using an ML algorithm instead of measurements, Our approach eliminates the code coverage issue that exists in related hybrid techniques. Also, it allows the users to find the WCET of any GPU kernels without getting involved in the complexity of running two kernels simultaneously.

To our best knowledge, nobody has used ML approaches to estimate the WCET of the GPU kernel. The NVIDIA CUDA programming approach is discussed in this work, although the approach also applies to other analogous technologies, such as OpenCL.

Contributions: The contributions of the proposed approach are:

- We propose a novel approach to estimate WCET of the GPU kernel using ML based timing model of the GPU. This is to analyze the timing properties of GPU programs.
- We implement our approach for commodity GPU hardware (i.e. NVIDIA RTX 3060).
- A selection of best prediction model based on R^2 learning score obtained after experiments.

- We evaluate our approach with several GPU kernel benchmarks. Our result shows promising WCET estimation because we observed that predicted WCETs are always higher than actual WCET for all benchmarks and all ML algorithms. Our implementation and all experimental data are publicly available[1].

3.2 Background

To make an easier description of the GPU analytic model and optimization techniques in the next sections, we describe some background knowledge about modern GPUs in this section. The structure and programming model of CUDA are explained in Section II-A. The architecture and features of Ampere GPU are detailed in Section II-B.

3.2.1 Terminology

Although GPUs have existed in some form for many years, NVIDIA coined the term ”GPU” in 1999 with the introduction of the GeForce 256 [1]. Many of the terms used by GPU researchers today were developed by NVIDIA. The phrases, however, pertain to private components, of which only certain information is made available to the general public. The main rival of NVIDIA, AMD provides consumers with access to greater information regarding GPU implementation specifics. Unfortunately, two distinct sets of terminology and definitions for the GPU's components have emerged as a result of two manufacturers separately creating GPUs that are not entirely open-source. Although each company's GPUs have some distinctive features, most of these terms refer to similar ideas.

3.2.2 GPU Hardware Model

A GPU is composed of multiple SMs sharing an L2 cache and DRAM controllers via a crossbar interconnection network. The SMs are the central parts of the GPU architecture, which perform all the vertex/geometry/pixel-fragment shader programs and GPGPU- programs. As shown in Figure 3.1, an SM features a number of Scalar Processor cores (SPs) and two other types of function units — the Double-Precision Units (DPUs) for Double-Precision (DP), floating-point calculations and the Special-Function Units (SFUs) for processing transcendental functions and texture-fetching interpolations. Other components, such as the Register Files (RFs), Load-Store Units (LSUs), scratchpad memory (i.e., shared memory), and various caches (i.e., instruction cache, constant cache, texture/read-only cache, L1 cache) for on-chip data caching also reside in the SMs. Each of these GPU components has been explained briefly below.

[1]The source code is available at https://cfaed.tu-dresden.de/pd-downloads

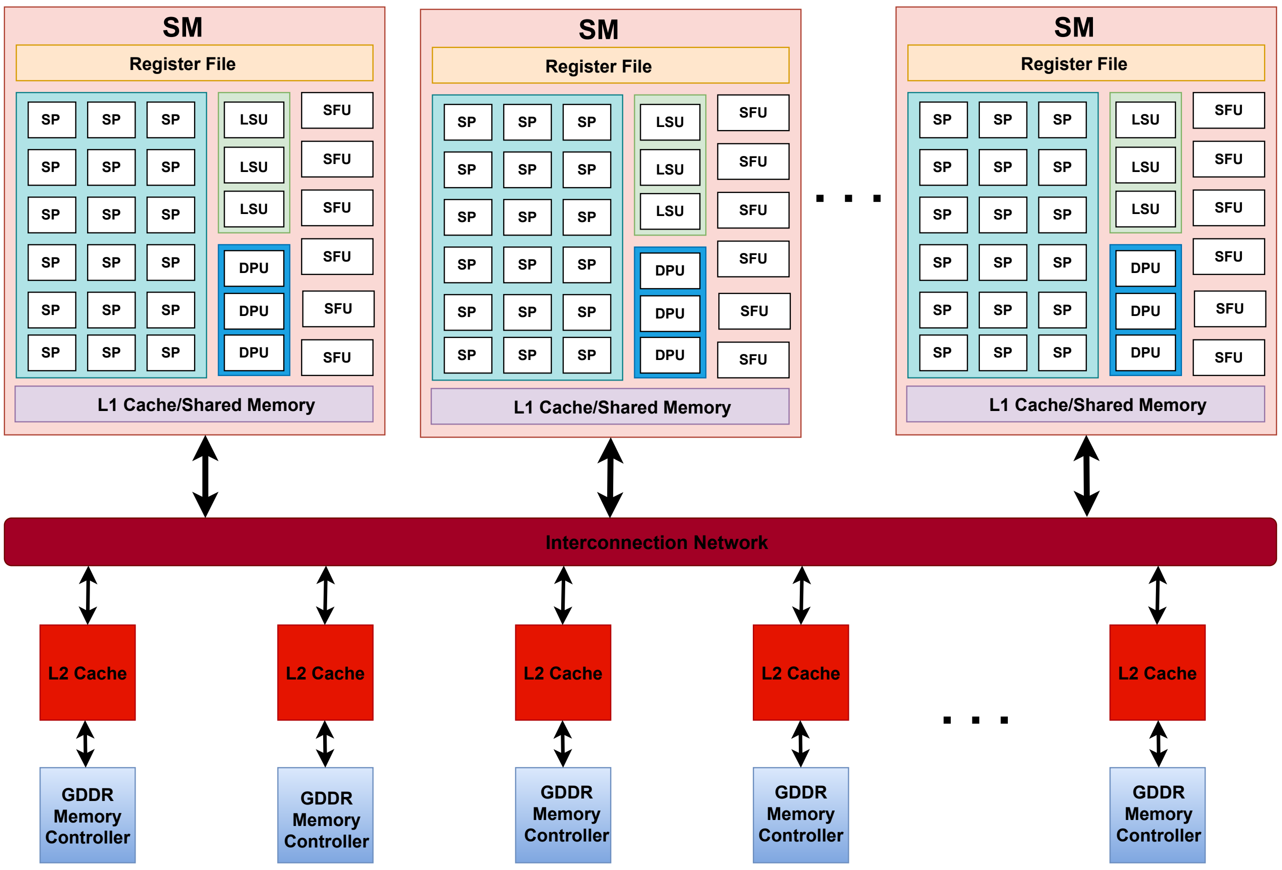


Figure 3.1: General Architecture for Modern GPUs.

- **Scalar Processor (SP):**The major basic processors of an SM are the scalar processors, sometimes referred to as CUDA cores, which carry out the essential integer, floating-point, comparison, and type conversion operations. Both the single-precision Floating-Point Unit (FPU) and the integer Arithmetic Logic Unit (ALU) are fully pipelined components of each SP.

- **Special-Function-Unit (SFU):** For quick transcendental function computations (such as sine, cosine, reciprocal, square root, etc.) and planar attribute interpolations, the SFUs are integrated. In addition to the SPs, each SFU has four floating-point multipliers that can increase FP throughput. The SP pipelines and the SFU pipelines operate separately.

- **Double-Precision-Unit (DPU):** Particularly for DP computations, the units are the DPUs. They employ extremely effective deep pipelines to carry out Fused Multiply-Add (FMA) DP operations. The amount of DPUs in an SM determines the GPU device's DP performance; for example, the Maxwell GPUs have only 4 DPUs in the SMs and give only

1/32 of their SP performance as DP performance.

- **Load-Store-Unit (LSU):** The LSUs are used to fetch and save data to memory, as suggested by their name. They have specialized processing power to quickly determine the source and destination addresses for memory requests.

- **Register Files:** In general, GPUs contain a huge number of registers. Because of their size, GPU registers are implemented by SRAMs that are divided into banks in order to maximize throughput. As a result, GPU registers have a high access latency relative to CPU registers and may encounter bank conflicts [146].

- **Local Memory:** The local memory is a piece of the global memory rather than a physical memory area. Similar to RFs, its scope is thread-private. When there aren't enough registers to carry all the necessary variables (i.e., register overflowing) or when arrays are specified in the kernel but the compiler is unable to determine the precise indexing to access them, it is typically utilized for temporal spilling. In Fermi and Kepler, both L1 and L2 cache the local memory, but in Maxwell and Pascal, only L2 cache the local memory. Register spilling in local memory hurts the performance as it introduces extra instructions and memory traffic, especially when there is a cache miss (so the register value has to be fetched from off-chip global memory).

- **Shared Memory:** The on-chip storage known as shared memory or scratchpad memory is distributed among all of the units of an SM. It functions as a communication channel for quick data transfer between various threads in a thread block (also known as a Cooperative Thread Array or CTA). Compared to local memory or global memory, shared memory offers a substantially larger bandwidth and lower access latency since it is on-chip. The CUDA programming guide strongly advises optimizations that can move global/local memory access to shared memory [49]. Similar to register files and L2 cache, shared memory may be accessed in parallel since it has been divided into banks to increase bandwidth. However, in case two addresses from the same memory request fall into the same bank, a bank conflict occurs, and the accesses have to be serialized, which seriously degrades the performance of the shared memory.

- **Global Memory:** The device memory, also referred to as GPU off-chip memory or GPU main memory, is the global memory. Since it is the memory that GPUs use the most frequently, its throughput often determines the maximum performance that GPUs can achieve in memory-bound applications. The attainable global memory throughput, or sustainable throughput [11], is mainly constrained by two factors: raw memory bandwidth and coalescing degree. (1) The raw memory bandwidth is limited by the pin number, wire length, and the physical property of DRAM; therefore, it has been increasing slowly since Kepler. However, the 3D-stack memory technique that Pascal recently used completely transforms such an unchanging situation [144]. (2) Memory access coalescing, a method, is used to benefit from sending huge data blocks at once. Each warp lane's target addresses are first calculated separately by the LSUs. Prior to memory retrieval, specialized Address-Coalescing hardware [32] will verify that addresses from the same warp are constantly spread, which is the typical scenario for global memory access. Following one or more aggregated block transfers from the cache or main memory, it notifies the memory interface units [32]. The CUDA programming guide provides a detailed discussion about the identification of memory coalescing [50].

- **L1 Data Cache:** In Fermi, the GPU's L1 data cache made its debut. The shared memory of an SM and the SM-private L1 cache both share the same on-chip storage. (16/48 or 48/16 KB in Fermi and 16/48, 32/32, or 48/16 KB in Kepler) Their relative sizes are reconfigurable. 128 B makes up the L1 cache line. It is non-coherent and caches both read and write operations to local and global memory [91]. For register spills, function calls, and automatic variables, local memory is often used [49]. When caching access to local memory instead of global memory, the L1 cache is writable rather than read-only. The standard L1 cache and the texture cache are combined from Maxwell.

- **L2 Cache:** Fermi also served as the foundation for the unified L2 cache. All memory access types, including global, local, constant, and texture, are supported, and it is consistent with the host CPU memory. The L2 cache has a write-back replacement strategy and is readable and writable [91]. It serves as the focal point for data unification [44] and provides a convenient setting for data transfer amongst SMs. In order to drastically lower the overall memory bandwidth need, the L2 cache is typically divided into banks, each of which serves as a buffer for an off-chip memory channel (GDDR or HBM2-DRAM).

- **Interconnection Network (NC):** A crossbar network serves as the SM and L2 banks' network of connections. It enables concurrent communication between many SMs and L2 banks, greatly increasing NC throughput. A conventional crossbar NC encompasses two data buses and an address bus, as described in [45]. While the two data buses form a bidirectional channel between SMs and L2 banks, the address bus is only ever going in one direction from SMs to L2 banks. Point-to-point communication is being used here [72]. Each SM and L2 bank has a Memory-Request Queue (MRQ) and a Bank

Load Queue (BLQ), respectively. An SM's LSUs will create load requests, which will initially be cached in the local MRQ before being sent to the destination BLQ through the crossbar NC. After some waiting time in BLQ, the request will be processed by the L2 banks. It is already known that the crossbar network comes at a high switching cost for simultaneous connections. Particularly, when the accessing requests are random and messy, interference will appear, which leads to the reduction of effective bandwidth [148].

3.2.3 GPU Execution Model

The GPU execution model refers to the way in which tasks are executed on a Graphics Processing Unit (GPU). The execution model of a GPU involves several key components and concepts:

1. Thread Hierarchy: At the core of the GPU execution model is the concept of threads. Threads are small units of work that are executed concurrently by the GPU. GPU threads are organized into a hierarchical structure that includes blocks and grids.
    - Thread: A thread represents the smallest unit of work that can be scheduled and executed independently on a GPU. Threads are typically used to process individual elements of data or perform specific calculations.
    - Block: A block is a group of threads that can cooperate and synchronize with each other through shared memory. Blocks are organized into a two or three dimensional grid structure.
    - Grid: A grid is a collection of blocks. It defines the overall structure and size of the parallel computation. Grids can also be organized in a two or three dimensional manner.
2. SIMT (Single Instruction, Multiple Thread): The GPU execution model employs a SIMT architecture, where multiple threads within a block are executed simultaneously in a SIMD (Single Instruction, Multiple Data) manners. SIMT allows the GPU to execute the same instruction across multiple threads simultaneously, which increases parallelism and computational efficiency.
3. Warp: A warp is a group of threads that are scheduled and executed together. In NVIDIA GPUs, a warp typically consists of 32 threads. All threads within a warp execute the same instruction but on different data elements.
4. Scheduling and Execution: The GPU scheduler manages the execution of blocks and grids on the GPU. It assigns blocks to available SMs for execution. Each SM contains multiple CUDA cores and executes the threads within a block in a SIMT fashion.

- Thread Scheduling: The GPU scheduler determines which blocks are scheduled for execution on each SM and manages the allocation and execution of threads within a block.
- Memory Coalescing: GPUs optimize memory access by coalescing memory transactions. Memory coalescing minimizes memory latency by combining multiple memory accesses into a single transaction whenever possible.

3.2.4 GPU Programming Model

GPU programs are typically written using a model framework on top of an established programming language. The CUDA programming model by Nvidia is the C/C++ interface for the GPUs operation implemented by a combination of hardware and device-driver software. Since the focus of this thesis is on NVIDIA GPUs, the programming model is described from the perspective of that.

The CPU and GPU components share many programming features. To avoid confusion between the two, the terms "host" and "device" are used as descriptors for data and code pertaining to the CPU and GPU, respectively. The code is organized into functions that are designed to be executed on the GPU, known as kernels. The program that executes on the GPU has at least one kernel but must start its execution on the CPU. The host code is responsible for preparing the data structures to be used by the GPU program, launching the program to the GPU, and collecting and cleaning up data once the GPU is done. A grid is associated with each kernel which contains a group of a thread blocks. Each thread block contains a maximum of 1024 threads. Within the thread block, a group of 32 threads termed warp execute in a lockstep manner. The thread from the same warp executes the same instruction in the same cycle, provided there is no divergence.

The kernel, such as the one found in Listing 3.1, falls into the category of device code. The __global__ keyword indicates to the compiler that the function is a kernel. In CUDA syntax, the threadblock index and thread index are represented by blockIdx.x and threadIdx.x, respectively. The first line of the vector addition kernel computes a globally unique thread index using the threadblock size (blockDim.x) and local offset. The suffix .x corresponds to one out of three possible dimensions; however, for the sake of simplicity, the threadblock considered is one-dimensional.

```
__global__ void vecAdd(double *a, double *b, double *c, int n){
  // Get our global thread ID
   int id = blockIdx.x*blockDim.x+threadIdx.x;

```

```
    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

Listing 3.1: Sample GPU kernel for vector addition

Listing 3.2 shows the simplified version of the host code for the sample vector addition kernel in Listing 3.1. As already discussed, the kernel is the function that runs on the GPU side by massive parallel GPU threads. The way to specify the number of threads to execute the kernel is via the <<< ... >>> configuration syntax. The CPU will

1. Allocate and initialize the host vector h_a,h_b,h_c
2. Allocate device vector d_a,d_b,d_c,
3. Copy the host data to the device (H2D),
4. Launch the kernel given a desired number of threadblocks and threads,
5. Copy the device data to the host (D2H),
6. If applicable, use the output data, and
7. Deallocate the host and device data.

```
// Allocate and initialize host vector
h_a = (double*)malloc(bytes);
h_b = (double*)malloc(bytes);
h_c = (double*)malloc(bytes);

// Allocate memory for each vector on GPU
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);

// Copy host vectors to device
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

int blockSize, gridSize;

// Number of threads in each thread block
```

```
blockSize = 1024;

// Number of thread blocks in grid
gridSize = (int)ceil((float)n/blockSize);

// Execute the kernel
vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

// Copy array back to host
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

// Release device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

// Release host memory
free(h_a);
free(h_b);
free(h_c);
```

Listing 3.2: Sample Host code for vector addition

3.2.5 Related Works

In this subsection, we compare our work to the existing literature. Many works have attempted to make GPU computation predictable or to develop an appropriate WCET analyzer.

In [18], the authors proposed an approach of measurement in conjunction with statistical approaches such as Extreme Value Theory to estimate safe probabilistic WCET of GPU applications. The use of the measurement approach gives unsafe WCET for GPU kernels. Contrary to our work, the method does not model interference and delay in execution due to other kernels running simultaneously.

In [62], GDivAn was introduced to measure the WCET of arbitrary GPU kernel. It combined the strength of Symbolic Execution (SE) and Genetic Algorithm (GA) to converge toward the WCET. The problem with this approach is the complexity of the SE becomes intractable with the growing number of threads. In contrast to their work, we focus on estimating the WCET using ML models instead of generating the worst-case inputs.

In [70], the author introduced the Stargazer, an automated GPU design space exploration framework based on step-wise regression modeling to understand benchmark diversity, hardware bottlenecks, and trade-offs. In contrast, we use different ML models for the timing estimation of kernels and use the MPS service to execute both victim and enemy kernels concurrently to measure the maximum interference.

In [16], the ILP-based approach was introduced to find the worst-case makespan of a GPU kernel on a single SM. The approach was computationally intractable for many threads, and the absence of cache effects made the approach limited to use. We propose a first attempt to consider the interference suffered by the victim kernel and use that information to train an ML model.

The work done in [17] extends [16] and presents the meta-heuristic of the simulated annealing technique to find the better makespan. The methods were not safe and applicable to soft real-time systems. Compared to [16], [17], we perform analysis at the binary code level. As a result, predictions have a higher level of accuracy since the code being examined accurately represents what the hardware really executes.

The WCET analysis of GPU L1 data caches was done in [65] using abstract interpretation. The proposed WCET analyzer can attain a safe and very accurate estimation of the worst-case GPU L1 data cache miss rates.

There are also recent studies on GPU to concurrently run two kernels to contend for computational resources in order to enable more confident measurement-based WCET estimations [69], [149]. In contrast to their work, we focus on predicting WCET instead of generating powerful enemies for the victim kernel. The work done in [23] used a Neural network to predict the impact of multicore contention on task execution time. Their technique could complement ours for validating timing on GPUs.

3.3 Detailed Methodology

The proposed scheme to estimate the WCET of the GPU Kernel has two phases. In the first phase, described in Section 3.3.1, ML models are trained. In the second phase, presented in Section 3.3.2, the WCET of kernels is estimated using trained ML models. The generation of training data is described in detail in Section 3.3.3. The ML-Based algorithm is explained in Section 3.3.4.

3.3.1 Learning of the GPU timing model

The process of developing a timing model using a labeled dataset to discover patterns, correlations, and dependencies within the data is known as the training phase in machine learning. It entails several processes and strategies to optimize the model's structure or parameters to reduce mistakes and boost its capacity for precise prediction.

In the training phase for GPU architecture, the objective is to enable the model to com-

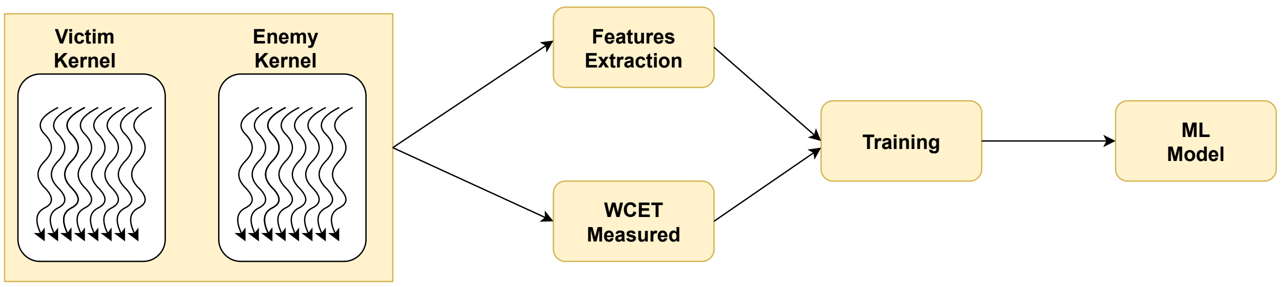


Figure 3.2: Training phase

prehend and predict the timing behavior of the GPU accurately. This process involves several steps, each aimed at gathering data and training machine learning models to understand and predict the WCET of specific kernels, as shown in Fig. 3.2. The first step involves the concurrent execution of both victim and enemy kernels using the Multi-Process Service (MPS). These terms, ”victim” and ”enemy,” originate from a paper [149], where the victim kernels represent those whose WCET we are interested in determining, while the enemy kernels deliberately contend for GPU resources. This concurrent execution scenario simulates real-world conditions where multiple kernels compete for GPU resources, enabling the model to capture the dynamic nature of GPU resource allocation and contention.

In the second step, profiling tools are employed to extract relevant features from the victim kernel. Profiling tools provide insights into various aspects of kernel execution, including memory access patterns, compute intensity, and resource utilization. By analyzing these features, the model gains a deeper understanding of the characteristics and behavior of the victim kernel, which is essential for accurate WCET estimation. The third step involves measuring the WCET of the victim kernel. This step serves as a label for the dataset used for training machine learning models. By directly measuring the WCET, the ground truth data is obtained, providing a reference for evaluating the performance of the trained models.

In the final step, the extracted features and corresponding WCET labels are utilized to train different machine-learning models. During the training process, the models learn to associate the extracted features with their corresponding WCET values, effectively capturing the complex relationships between kernel characteristics and execution times.

The ML models learn about the GPU using the numerical values called features. The features extracted for training are basically the instruction of the kernel and the metric, i.e., statements, operations, dram_read_bytes and dram_write_bytes. Some of the extracted features are shown in TABLE 3.1. Features and WCET estimates are both represented as floating-point values.

Table 3.1: Features extracted

# addition operations	# division operations	# subtraction operations
# logic operations	# function calls	# return statements
# load operations	# store operations	# multiplication operations
# shift operations	# jump statements	# comparison operations
dram_read_bytes	dram_write_bytes	l2_utilization
ram_read_throughput	dram_read_transactions	dram_utilization
dram_write_throughput	dram_write_transactions	inst_inter_thread_communication
eligible_warps_per_cycle	gld_throughput	gld_transactions
global_hit_rate	global_load_requests	global_store_requests
inst_executed_global_loads	inst_executed_global_stores	inst_executed_shared_stores
l2_utilization	l2_read_throughput	l2_write_throughput

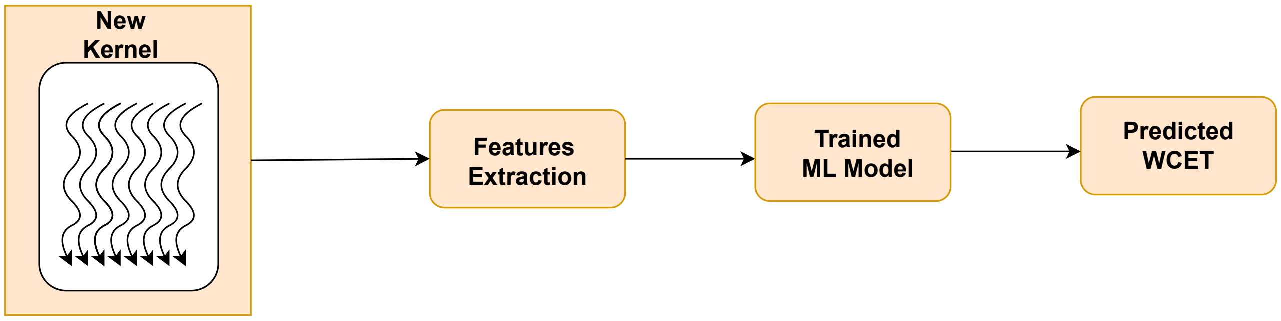


Figure 3.3: Testing Phase

3.3.2 Determining WCET of target kernel

FIGURE 3.3 illustrates the evaluation process of the model for estimating the WCET of new kernels from unseen programs. This evaluation is crucial for assessing the generalizability and effectiveness of the trained model beyond the training dataset. Firstly, the new kernel, which has yet to be encountered during the training phase, is selected to test the model's performance. Features pertinent to the kernel's execution characteristics are then extracted using the techniques established during the training phase. These features serve as inputs to the trained model. Subsequently, the model utilizes the learned weights (W) acquired during training to predict the estimated WCET for the given kernel. Through this process, the model leverages its learned understanding of the relationships between the extracted features and WCET to generate predictions. The evaluation process in FIGURE 3.3 demonstrates the model's capability to generalize its learned knowledge to unseen data, thereby providing valuable insights into its performance in real-world scenarios.

3.3.3 Generation of training data

The generation of the dataset requires following multiple steps as we need to set up the environment, which leads to maximum interference for our victim kernel. The first step is to choose the enemy kernel, which would provide more harm to the victim execution kernel than any actual concurrently running kernel. Secondly, we need to execute both kernels from the same CUDA context, i.e., non-time-sliced manner, because it allows both kernels to use GPU hardware at the same time. The two major environments for the concurrent execution of multiple kernels are Simultaneous Multi-Kernel (SMK) and Spatial Partitioning (SP), illustrated in [128], [149].

In SMK, the victim and enemy kernels contend for per-SM hardware, such as CUDA core, floating point unit, and load/store units. For instance, if all the load/store units are busy and there is a request for it from another kernel, it will have to wait, which will eventually delay its execution time. The local-shared memory, such as the L1 cache, is also a potential resource of contention due to warps within the SM. This can be achieved by incorporating inline assembly code into the kernel code. Listing 1.3 shows the CUDA code to implement this approach in the GPU. The actual code needs some additional to acquire the warps using techniques, defined in [147]. The major part is explained here to understand the partition of resources between victim and enemy kernels. The warpid of each warp is assigned using the special register. The conditional statement next in the figure represents that if the warp belongs to the enemy, then only execute the remaining code.

```
__global__ void SMK_Enemy_Kernel(unit64_t wpid){
    unit64_t wpid = 0;
    // inline assembly code to read special register
    asm("mov.u32 %0, %warpid;" : "r="(wpid));
    if(wpid > Total_number_warp/2){
        // Execute kernel code here
        ...
        ...
    }
}
```

Listing 3.3: CUDA code for SMK enemy kernel implementation

In SP, the victim and enemy kernels are assigned a different number of SMs. The L2 cache and DRAM is the primary source of contention between the kernels. For instance, the eviction of cache lines by the enemy kernel used by the victim kernel will increase the execution time for the victim kernel. Miss Status Handling Registers (MSHR) are also a potential resource of contention across the SMs. Listing 1.4 shows the CUDA code to implement this approach in the GPU. The actual code needs some additional to acquire the SMs using techniques defined in [147]. The major part is explained here, assigning different SMs to the victim and enemy kernel. Once the smid is known, it is checked whether it is assigned to the enemy or not. If it is not assigned, it's immediately exited because this SM is for the victim kernel.

```
__global__ void SP_Enemy_Kernel(unit64_t smid){
    unit64_t smid = 0;
    // inline assembly code to read special register
    asm("mov.u32 %0, %smid;" : "r="(smid));
    if(smid > Total_number_SM/2){
        // Execute kernel code here
        ...
        ...
    }
}
```

Listing 3.4: CUDA code for SP enemy kernel implementation

To allow the victim and enemy kernel to run concurrently, we have used NVIDIA's MPS. This service allows us to execute two different kernels with the same CUDA context. In this service, using the simple script, we allow the enemy kernel to launch first and then the victim kernel because we want the victim kernel to experience interference throughout its execution. Once this environment is set up, we extract the features from the victim kernel to form our dataset, which will be fed to ML models.

It is unjustified to assume that a kernel performing a concurrent task will experience worst-case interference on a consistent basis. But even with relatively simple code, a properly designed enemy program can continuously and successfully stress an interference channel. We recommend interested readers to [13], for methods designed to produce enemies that are more harmful to the victim's execution time.

3.3.4 Algorithm for ML-Based approach

The pseudo-code for the ML-Based approach, which includes code generation, training, and testing of the ML approach, is presented in Algorithm 1. As inputs, the algorithm takes the victim and enemy kernel, the GPU platform whose timing model to learn. On the other hand, the $WCET$ at the algorithm's termination is the application's output. The procedure begins with calling to function *DataGenerator*, which uses GPU MPS to concurrently execute two kernels to obtain the features and labels. Once the data is generated, it is fed to ML models for training. Once the training is done, the metric R^2 is used to choose the best ML model for the inference of the new kernel. The value obtained during inference/ testing is the desired output of the victim kernel.

Algorithm 1 ML-Based Approach

```
Input: Victim Kernel, Enemy Kernel, GPU
Output: $WCET$
1: Procedure ML_approach()
2: features, labels = $DataGenerator$(GPU_MPS(Victim Kernel, Enemy Kernel))
3: ML = $Training$(features,labels)
4: y = $Testing$(New/victim kernel)
5: $WCET$ = y
6: end Procedure
```

3.4 Experimental Setup

This Section presents the experimental setup used for the methodology discussed in Section 3.3. The hardware and software environments are first introduced in Section 3.4.1. The programs used for evaluating the quality of predictions are presented in Section 3.4.2. We then detail the learning and prediction phase of our approach Sections 3.4.3 and 3.4.4.

3.4.1 Hardware and Software Requirements

The NVIDIA GeForce RTX 3060 is used in our experiments running CUDA 12.1. Fig. 3.4 shows the layout of the NVIDIA GeForce RTX 3060 used in our work containing 28 SMs. Each SM contains multiple CUDA cores for integer and floating-point arithmetic and has an L1 cache of size 128KB shared among the threads. It also includes various load/store units that load data from/store data to cache or DRAM and multiple special function units implementing sine, cosine, square root, etc., in hardware. The smallest unit of scheduling in SM is warp. The warp scheduler dispatches the ready warp for execution on computing hardware. Each SMs share 3MB of L2 cache and a global memory of 12GB.

In this work, we use a measurement-based approach to train the ML model using different kernel instructions, which can predict the WCET of the GPU kernel to detect timing misconfiguration in the later development phase of the systems. We also largely avoid the problem that hindered earlier work, i.e., the lack of public knowledge about the properties of the memory subsystem, by employing measurements.

The profiling tools that are most frequently used for GPU programs on NVIDIA devices are Visual Profiler and Nsight Compute Command-line Profiler. The command-line profiler is heavily used in this work to measure various runtime events and performance indicators, such as kernel execution time and the number of instructions in the kernel.

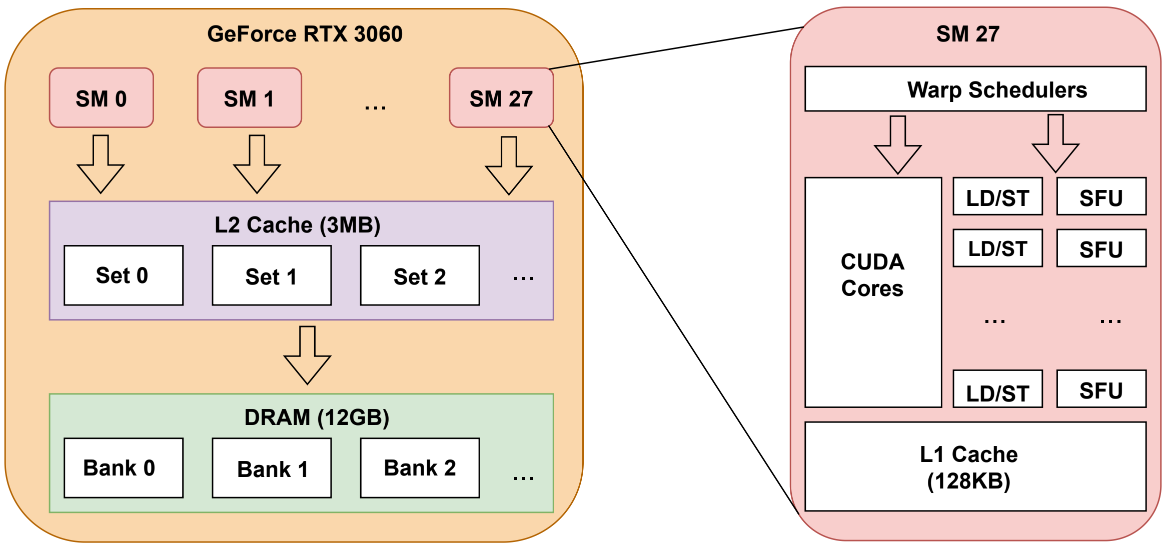


Figure 3.4: An NVIDIA Ampere GPU.

Table 3.2: Benchmarks for the evaluation

Application	Description	Domain	# kernel
2dconv	2D Convolution	Linear Algebra	1
Alexnet	Image classification	Neural Network	23
covar	Covariance computation	Linear Algebra	3
Eigenvalues	Scaled factor of eigenvector	Linear Algebra	1
fdt2d	2D finite difference time domain kernel	Simulation	3
gramschm	Gram-schmidt process	Linear Algebra	3
Histogram	Compute 2D saturating histogram with maximum 256 bins	Data Mining	2
MatrixMul	Dense matrix-matrix multiply	Linear Algebra	1
Mergesort	Merge sort Program	Data Mining	5
Vectoradd	Vector addition	Linear Algebra	1

3.4.2 Benchmarks

On ten benchmarks from the different benchmark suites, the accuracy of WCET predictions of the kernel was assessed. TABLE 3.2 shows the full description of the victim kernel used for the evaluation of the approach. The kernels are chosen such that it belongs to different domains to show the prospect of our approach. The dominance of applications from the Linear Algebra domain is more here because mostly GPU in Autonomous vehicles uses computer vision algorithms to detect pedestrians and objects, whereas all these applications run underlying these algorithms only. The number of kernels in each application varies depending on the domain of the application, which further leads to different numbers of operations and statements in each kernel and eventually having different execution times.

3.4.3 Training Phase

A total of 5000 kernels was generated for the training of the ML model from the different benchmark suites such as CUDA-SDK, FastHOG [117], GPGPU-sim [10], LonestarGPU [106], Mars [60], Maryland [56], Parboil [132], Polybench [46], Rodinia [31], Shoc [34], and Tango DNN Suite [71]. We have ensured that these suites' generated kernel covers all the instructions from the different domains using different inputs. The quality of the timing model depends on the input training data, which must be as realistic of the real world as feasible in order for the timing model to perform well. This is totally avoided in our case because we have chosen familiarly known benchmarks to create the dataset. To the greatest extent possible, bias in the training data was avoided because it led to the overfitting of the model. We found that some features have greater values, and some features have lower values. These differences in the range of values bias the model's prediction to be inclined towards values of features that are larger, and the features having lower values contribute much less to the prediction, i.e., low-value features have no significance in the prediction. To tackle this issue, we need to normalize our data value in the range of 0 to 1. Several frameworks, such as PyTorch [109] and Tensorflow [2], are available today. We have used the PyTorch framework in this experiment.

We used 80% of kernels for the training and cross-validation and the remaining 20% for the testing in both environments. Each kernel was executed ten times to set the Maximum Observed Execution Time (MOET). Executing the 5000 kernels to obtain the timing samples for the two environments required approximately seven days using an NVIDIA RTX 3060 GPU card. The length of the training phase is not a concern to us because it only needs to be done once for each architecture. Training, executed on a Linux machine running on a Victus HP R5-5600G with six-core AMD Ryzen processors, required around 40 minutes for the 5 ML algorithms.

We chose the 5 ML algorithms based on preliminary experiments that produced the best outcomes, such as Support Vector Regression (SVR), Random Forest (RF), ElasticNet (EN), Adaptive Boosting (AdaBoost), and Gradient Boosting (GB), has been made as shown in TABLE 3.3. The abbreviations SVR, RF, EN, AdaBoost, and GB will be used throughout the remainder of the chapter in place of the full names and explained in detail in Chapter 2.

The selection of the best models is made using their R^2 score. The R^2 score, also known as the coefficient of determination, is a statistical measure that indicates how well the predicted values of a model fit the actual values. The R^2 score ranges from 0 to 1, with a higher score indicating a better fit.

Table 3.3: Experimented ML Algorithm

Algorithm	**Description**
Support Vector Regression (SVR)	Type of support vector machine
Random Forest (RF)	A multitude of decision tree
Gradient Boosting (GB)	Boosting algorithm with complex dataset
Adaptive Boosting (AdaBoost)	Boosting algorithm with simpler dataset
ElasticNet (EN)	Combination of Lasso and Ridge Regularization

3.4.4 Testing Phase

The goal of the testing/inference phase is to quickly and effectively produce predictions or insights using the trained model. Compared to the training phase, it takes fewer computing resources and is frequently speed and efficiency-optimized. In situations where real-time reactions are necessary, the process might be interactive or automated and incorporated into production systems.

The testing phase is implemented by using the scikit library. The trained ML models with their weights and biases were used to predict the WCET of new kernels in the current state of the implementation. The WCET for each kernel is estimated for both environments, one for SMK and the other for SP.

3.5 Evaluation

The quality of our approach is evaluated from different points of view. First, we evaluate the quality of WCET predictions of entire programs in Section 3.5.1. Then, we evaluate the WCET of different benchmarks using SMK (Section 3.5.2) and SP (Section 3.5.3). The comparison with the other policies are then presented in Section 3.5.4.

3.5.1 Prediction of WCET of Kernel

The detailed analysis of benchmark atax, which is matrix transpose and vector multiplication from Polybench, is shown in Fig 3.5. This benchmark is straightforward enough to guarantee that our method is the only source of pessimism in WCET estimations. It depicts the MOET and ML models predicted WCET for all 5 ML algorithms. The estimated WCETs are compared with the MOET of the benchmark, obtained by taking the maximum timing of 1000 executions, all using the inputs that trigger the worst-case execution path. The SMK environment is used for this experiment. We randomly choose the 1000 instances to show the result here, and it has nothing to do with the Maximum Observed Execution Time (MOET), which periodically increases and then decreases as the number of runs progresses.

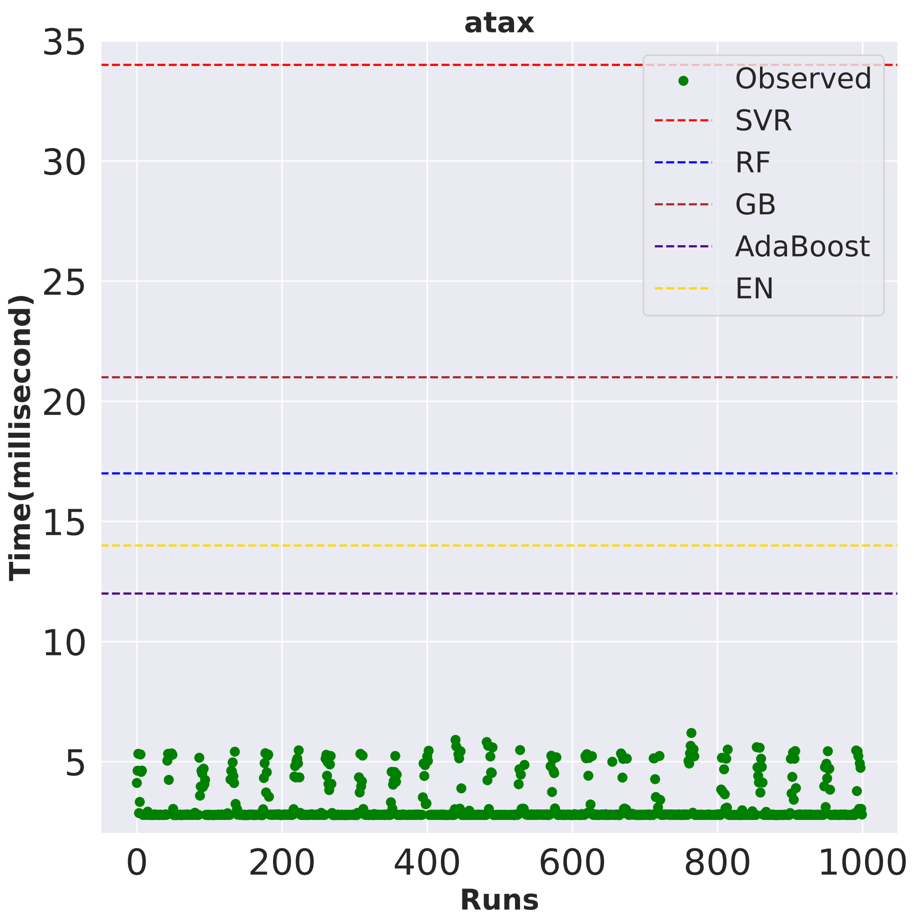


Figure 3.5: ML models WCET versus observed execution times for atax application

In terms of atax, SVR received the highest WCET estimate, followed closely by GB. AdaBoost obtained the smallest WCET estimate, followed tightly by EN. Although employing our ML models, we did not observe any WCET estimates that were understated. We found that most programs had WCET prediction duration's of around thirty seconds when it came to WCET estimation time. Similarly, the detailed analysis of benchmark kmeans from Rodina, is shown in Fig 3.6. In this case, SVR also received the highest WCET estimate, followed closely by GB. EN obtained the smallest WCET estimate, followed tightly by Adaboost. The SP environment is used for this experiment. The EN shows better results than other models because of its inherent property, which allows both L1 and L2 regularization to be used in the cost function to minimize it. The L2 regularization reduces the overfitting of the model to become a complex model for the given patterns. Similarly, L1 regularization not only reduces

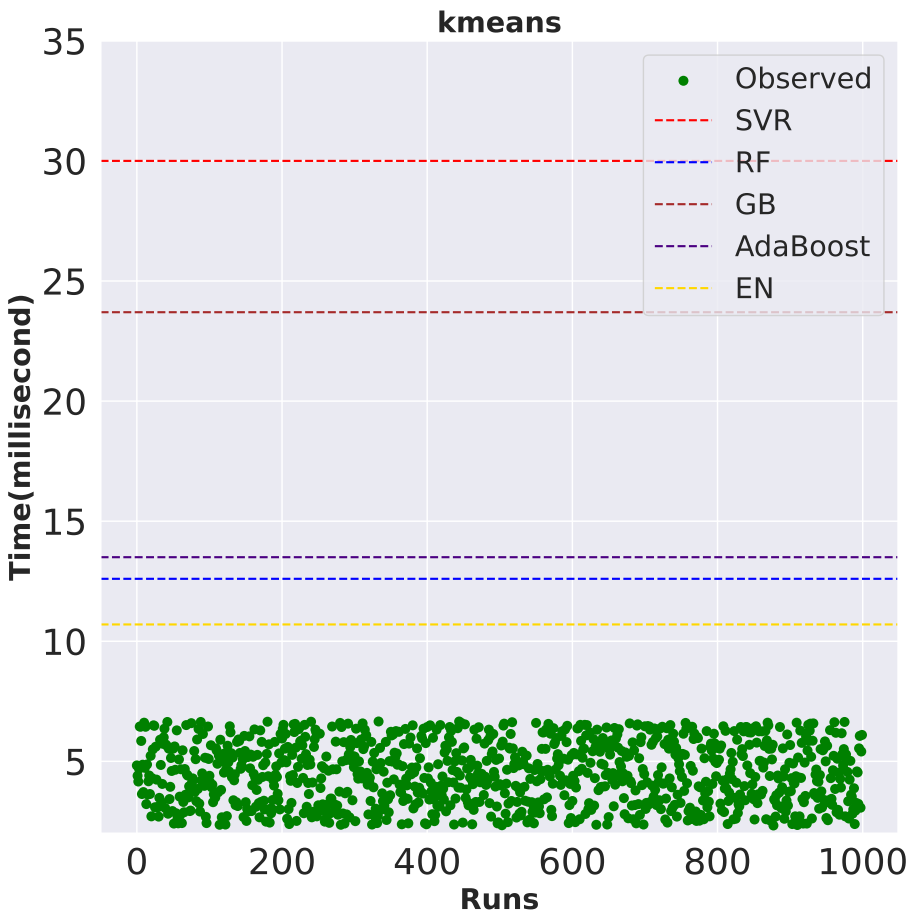


Figure 3.6: WCET obtained using ML models versus MOET for bfs application

the overfitting of the model but also helps with feature selection.

3.5.2 SMK Environment

Prior work has been done to choose the most promising enemy kernel capable of causing maximum interference, such as sacalrProd and stereoDisparity. We have used scalrProd as an enemy/stressor kernel in this environment from [149].

TABLE 3.4 reports the WCET estimated by our approach on the benchmarks using the five selected ML algorithms. The estimated WCETs are compared with the MOET of each benchmark, obtained by taking the maximum timing of 1000 executions, all using the inputs that trigger the worst-case execution path. The best-performing variant of ML models obtains the predicted WCET values in the TABLE regarding kernels' quality of learning: see Section

Table 3.4: ESTIMATED WCET OBTAINED BY OUR APPROACH VERSUS MOET FOR SMK ENVIRONMENT

Application	**SVR** (ms)	**RF** (ms)	**GB** (ms)	**AdaBoost** (ms)	**EN** (ms)	**MOET** (ms)	**Overestimation factor**
2dconv	5.12	19.61	2.07	7.7	**1.47**	0.282	5.25
Alexnet	81229.63	46053.49	20730.16	42781.85	**39901.50**	7224.10	5.52
covar	75020.81	**5122.96**	14765.19	5794.17	14862.7	1578.07	3.24
Eigenvalues	51.22	9.42	45.30	**5.80**	16.7	4.67	1.24
fdt2d	10441.69	2461.17	1843.67	1676.84	**1650.55**	158.929	10.38
gramschm	51229.6	87430.01	**35655.23**	41624.42	41456.58	2417.16	14.75
Histogram	2.07	117.75	**0.61**	3.9	1.49	0.11	5.45
MatrixMul	10.34	3.41	4.15	4.90	**2.14**	0.52	4.11
Mergesort	34.12	24.31	14.23	28.26	**15.8**	2.15	7.34
Vectoradd	5.12	5.67	3.07	5.88	**2.47**	0.24	10.29

Table 3.5: ESTIMATED WCET OBTAINED BY OUR APPROACH VERSUS MOET FOR SP ENVIRONMENT

Application	**SVR** (ms)	**RF** (ms)	**GB** (ms)	**AdaBoost** (ms)	**EN** (ms)	**MOET** (ms)	**Overestimation factor**
2dconv	5.12	19.61	2.07	7.7	**1.47**	0.385	3.81
Alexnet	81229.63	46053.49	20730.16	42781.85	**39901.50**	7975.966	5.0
covar	75020.81	**5122.96**	14765.19	5794.17	14862.7	2083.491	2.45
Eigenvalues	51.22	9.42	45.30	**5.80**	16.7	4.68	1.23
fdt2d	10441.69	2461.17	1843.67	1676.84	**1650.55**	159.01	10.38
gramschm	51229.6	87430.01	**35655.23**	41624.42	41456.58	7004.977	5.08
Histogram	2.07	117.75	**0.61**	3.9	1.49	0.11	5.45
MatrixMul	10.34	3.41	4.15	4.90	**2.14**	0.97	2.20
Mergesort	34.12	24.31	14.23	28.26	**15.8**	2.87	5.50
Vectoradd	5.12	5.67	3.07	5.88	**2.47**	0.34	7.26

IV-D for more details. The rightmost column gives the overestimation factor, calculated as the ratio between the estimated WCET and the MOET. The estimated WCET used to calculate the overestimation factor is the one depicted in boldface in the TABLE, calculated by the less pessimistic ML technique. It describes how much the predicted WCET deviates from the MOET.

The calculated WCETs are consistently higher than MOETs, as we have observed. No ML model consistently outperforms the others on all benchmarks. The lowest estimated WCETs are most of the times computed by EN (6 times out of 10) and GB (2 times). The ML algorithm that computes the largest WCET estimates the most often is SVR. fdt2c is by far the overestimated benchmark, for the rest the overestimation factor varies between 1.24 and 10.38.

Table 3.6: R^2 score of ML Models for SMK

ML Models	**R^2 score**
SVR	0.45
RF	0.57
GB	0.60
AdaBoost	0.65
EN	0.77

Table 3.7: Comparison of approaches to estimate WCET using [19]

Application	[19]			**SMK**			SP		
	Actual (Cycles)	Predicted (Cycles)	Difference	Actual (ms)	Predicted (ms)	Difference	Actual (ms)	Predicted (ms)	Difference
Eigenvalues	1,143,429	2,801,330	145%	4.67	5.80	24%	4.68	5.80	24%
Histogram	1,811,709	1,274,469	603%	0.11	0.61	455%	0.11	0.61	455%
MatrixMul	3642	4680	29%	0.52	2.14	389%	0.97	2.14	120%
Vectoradd	656	659	1%	0.24	2.47	930%	0.34	2.47	626%

3.5.3 SP Environment

To evaluate GPU hardware contention in the SP environment, the previous work chose the ExecuteFirstLayer kernel from SqueezeNet, fwtBatch2Kernel from fastWalshTransform, and vectorAdd. In this work, we chose vectorAdd as an enemy kernel from [149].

TABLE 3.5 shows the estimated WCET of the benchmark versus the MOET for the SP environment. We observed from the TABLE that the prediction of WCET from different ML models are same as in TABLE 3.4 because the features fed to ML models in this experiment are the same as SMK. Only the labels change. The overestimation factor varies between 1.23 and 10.38.

The best ML model is chosen based on the R^2 score. TABLE 3.6 shows the R^2 score of each ML model. A score close to one is the best fit for the model. We observed that EN has the maximum value, followed by AdaBoost. The lowest is shown by the SVR, which represents the worst model out of all. We observed that EN has the maximum value, followed by AdaBoost. The lowest is shown by the SVR, which represents the worst model out of all. EN is the best ML model in both SMK and SP environments, which performs better than others because it has both the L1 and L2 regularization terms in the ints equation, as defined in chapter 2. The L2 regularization helps to reduce the overfitting of the EN, and the L1 regularization helps to select the features.

3.5.4 Comparison with other policies

The two techniques, named dynamic using measurement based on high-water mark measurements and hybrid through a static analytical model based on instrumentation point graphs annotated with execution time, were presented in [19]. The approaches assumed that the thread blocks were coming in waves and scheduled using a round-robin. The pessimistic result on the GPU kernels using the approaches didn't scale it for further use. In contrast to their work, our approach doesn't suffer from code coverage issues.

In TABLE 3.7, we present a comparison of the percentage differences in WCET estimation between two different approaches: one presented in [19], termed as the "hybrid approach," and our approach using two different environments, SMK and SP, as detailed in TABLE 3.4 and 3.5. This comparison aims to evaluate the effectiveness of our proposed method against an existing approach, shedding light on its strengths and weaknesses. The first column of TABLE 3.7 lists the application names, while the second column displays the percentage difference using the hybrid approach from [19], showcasing the disparity between actual and predicted WCET. The third and fourth columns exhibit the percentage difference using the SMK and SP environments, respectively, obtained from the data provided in TABLE 3.4 and 3.5. Our analysis reveals that our approach outperforms the hybrid approach in some applications, while the latter proves to be more effective in others. This observation underscores the importance of considering various factors and methodologies in WCET estimation, as different applications may exhibit varying characteristics that impact prediction accuracy differently. Notably, the percentage differences obtained are considerable, suggesting they might not be suitable for direct use as upper bounds. However, we emphasize the value of these numbers in the early stages of system development. Despite their limitations, these estimates serve as valuable insights that can guide system designers, preventing them from making overly pessimistic assumptions about WCET. Understanding the limitations of WCET estimation methods is crucial for system designers to make informed decisions about resource allocation, scheduling, and overall system performance. While the presented percentage differences may not directly translate into precise upper bounds, they offer valuable early-stage guidance, allowing designers to refine their strategies and uncover areas for optimization.

TABLE 3.8 compares the estimated WCET using [62] and our approaches within both environments against the MOET. The applications examined include LBM and BFS from the Rodina benchmark and NSICHNEU from the Mälardalen benchmark [52]. The [62] proposed three distinct approaches for WCET estimation of GPU kernels, with GDivAn identified as the most effective method. Our comparison with these approaches reveals that our methods

Table 3.8: Comparison of approaches to estimate WCET using [62] with MOET

Application	[62]			**SMK** (ms)	**SP** (ms)	**MOET** (ms)
	GDivAn (ms)	random+ga (ms)	random (ms)			
LBM	12028	1514	7255	1100	1300	150
BFS	1411	911	1128	500	300	80
NSICHNEU	12669	13333	2702	800	1600	210

consistently demonstrate a significantly reduced deviation from MOET. Our approaches are remarkably close to MOET across all examined applications and environments. This suggests a higher level of accuracy and reliability in our WCET predictions compared to the methodologies proposed in [62]. This notable improvement could signify various factors, such as the sophistication of our modeling techniques, the quality of the data used for training, or the robustness of our evaluation methodologies. Regardless of the specific reasons, the findings underscore the efficacy and superiority of our approaches in estimating WCET, paving the way for more dependable performance analysis and resource allocation in GPU kernel contexts.

Our approaches demonstrate significant promise in accurately estimating the WCET of GPU kernels, particularly when compared to state-of-the-art methods. Notably, our techniques achieve estimations that closely align with the MOET, indicating their efficacy in reliably capturing worst-case scenarios. One of the key advantages of our approaches lies in their ability to streamline the WCET estimation process, particularly in GPU kernel contexts. Unlike traditional methods that often involve complex setup procedures for worst-case environments, our approaches leverage Machine Learning (ML) models to simplify and expedite the estimation process. Our approaches can rapidly provide WCET estimations for any application within seconds by training ML models on relevant data. This efficiency is particularly valuable when time constraints are critical, enabling developers and system designers to make informed decisions without investing extensive resources into repetitive environment setup procedures. Overall, our approaches represent a significant advancement in WCET estimation methodologies, offering a compelling alternative to traditional techniques. Their ability to achieve close alignment with MOET while simplifying the estimation process underscores their practicality and suitability for real-world applications. By empowering developers with rapid and accurate WCET estimations, our approaches facilitate better decision-making and enhance the reliability and efficiency of systems utilizing GPU kernels.

3.6 Summary

This chapter presented an approach to determine the WCET of the GPU applications using the ML approach. The approach allows the victim and enemy kernel to execute concurrently, which attempts to produce worst-case contention on shared GPU resources. The features are the victim kernel instruction, and the label is WCET which is fed to our model to train it. The model performance is evaluated on real benchmarks, and our approach result showed that none of the ML models consistently outperforms the others on all benchmarks. Although our approach does not offer safety guarantees, we observed that predicted WCETs are always higher than any observed execution times for all benchmarks. Compared with traditional approaches, our ML methodology minimizes resource consumption to estimate WCET, saving hours of GPU execution. In most circumstances, our ML strategy decreases the time by 99% because inferences only take seconds to forecast WCET. Although our approach does not offer safety guarantees because of its empirical nature, we observed that predicted WCETs are always higher than any observed execution times for all benchmarks, and the maximum overestimation factor observed is 11x. Finally, WCET estimates for all benchmarks were calculated in seconds for most benchmarks.

Although the benchmark results demonstrate the efficacy of our methods, they also open up the possibility of investigating related problem paradigms. Could an application having more than two execution times based on the critical level affect the WCET estimation on the uniprocessor or multi/many-core processors? We would like to answer these questions in the later part of this thesis.

Chapter 4

Estimation of WCET on MCS

In Mixed-Criticality (MC) Systems, there is a trend of having multiple functionalities upon a single shared computing platform for better cost and power efficiency. In this regard, estimating the suitable optimistic WCET based on the different system modes is essential to provide these functionalities. A single application is assigned multiple WCETs based on the criticality of the system, such as safety-critical, mission-critical, and non-critical. Determining the appropriate WCET in low critical mode is challenging and has been done in a few research works due to its complexity.

We propose ESOMICS, a novel scheme, to obtain appropriate WCET for LO modes, in which we first propose an analytical approach. This approach introduces a newly defined metric, Lowest cycle Time for Majority of samples (LTM) for WCET determination. We then propose an ML-based approach for WCET estimation based on the application's source code analysis and the model training using a comprehensive data set. Our experimental results show that the estimated WCET is close to the analytical approach for all the benchmarks. On average, it exceeds 25% and 15% on the prediction cycle and the metric, respectively. Our experimental findings also demonstrate that our approach improves the utilization of cutting-edge MC systems by 14% and 7%, while the percentage of task overrunning in a worst-case mode is 4% and 3% in both approaches, respectively.

In the rest of this chapter, the problem statement, motivational example and contributions are presented in Section 4.1. Section 4.2 describes the related work in estimating WCET with and without ML approaches. The application and system models are presented in Section 4.3. In Section 4.4, we describe ESOMICS in detail, while our experimental results have been described in Section 4.5. Finally, we conclude the chapter in Section 4.6.

4.1 Introduction

A Mixed-Criticality (MC) system is commonly used in embedded systems to meet the cost, space, and energy efficiency requirements of various applications, such as automotive, medical devices, and avionics [14, 26, 51, 97]. These MC systems are designed to perform multiple tasks with different criticality levels while ensuring the correct execution of these tasks. In order to prevent catastrophic damages, the system should ensure that all High-Criticality (HC) tasks are guaranteed to execute successfully before their deadlines while scheduling a large number of Low-Criticality (LC) tasks to maximize the processor utilization and Quality-of-Service (QoS) [26, 51, 119]. In conventional real-time systems, the tasks are scheduled based on their pessimistic Worst-Case Execution Time (WCET) [145], which can be estimated by the measurement-based, static analysis, and the hybrid approaches. Many tools and frameworks are available as open source, and commercials, such as aiT [39], Chronos [93], Heptane [57], and OTAWA [13], are used to determine the pessimistic WCET when the hardware architecture and compiled binary code are available. However, most execution times of task samples are shorter than conservative WCET, which leads to poor processor utilization and QoS in conventional real-time systems [145].

In this regard, multiple WCETs are defined in MC systems corresponding to the different criticality levels and the ongoing mode of operation [98]. Since there are different MC system operational modes, it ensures that QoS and processor utilization are maximized in the low-criticality mode (LO mode) while the constraints are preserved in the high-criticality mode (HI mode). Initially, an MC system starts its operation in the LO mode while executing the tasks based on the optimistic WCET. If the execution time of at least one HC task exceeds its optimistic WCET, the system mode changes from LO to HI mode. In such a scenario, all or some LC tasks must be dropped/degraded to provide the processor computation capacity for running the HC tasks and guarantee their correct execution before their deadlines. However, it can drastically affect the service and cause significant performance loss of LC tasks. When the gap between the pessimistic and optimistic WCETs is large, more tasks, such as LC tasks, are scheduled at design-time. However, this can cause system mode switches to occur frequently and, consequently, drop more tasks at run-time. When this gap is small, the overall processor utilization decreases due to scheduling fewer tasks at design-time [120]. As can be realized, the optimistic WCET is an essential factor in the design of MC systems. However, the major problem of designing such MC systems is to guarantee the timing constraints of executing only high-criticality tasks under very pessimistic assumptions and all tasks, including low-critical ones, under less pessimistic assumptions. The suitable optimistic WCET of the given application is defined as the WCET, which maximizes the processor utilization and minimizes the mode switches, which is observed by us through experimentation of the application distribution graph. Hence, for the given application, if this WCET is known, we can define it as a suitable optimistic WCET for the given application.

Motivational Example: Most previous research works set $WCET^{opt}$ based on the Average (AVG) execution cycle of the application [120] or fraction of the WCET^{pess} [14, 51]. These approaches may lead to poor processor utilization (while $WCET^{opt}$ is close to $WCET^{pess}$), or more mode switches (when there is a high gap between $WCET^{opt}$ and $WCET^{pess}$). As an example, Figure 4.1 shows the time distribution of a real application, *ns* from Mälardalen benchmark [52], running on the Raspberry Pi 4 board. The X-axis represents the processor clock cycles. For this application, the $WCET^{pess}$ is estimated to be 52531 cycles, using the SWEET tool [95]. With setting the $WCET^{opt}$ as a percentage of $WCET^{pess}$, many system mode switches occur if we set WCET^{opt} to $\frac{WCET^{pess}}{32}$ and $\frac{WCET^{pess}}{64}$, but more LC tasks can be scheduled in the system. On the other hand, if $WCET^{opt}$ is set to $\frac{WCET^{pess}}{4}$ and $\frac{WCET^{pess}}{8}$, or based on the average execution cycle, system mode switches are reduced, but poor utilization occurs due to scheduling fewer LC tasks. However, if $WCET^{opt}$ is set to close the indigo line (shown by Optimal $WCET^{opt}$ in the figure), both processor utilization and mode switches may be improved compared to other approaches.

To address such a problem, we propose a novel scheme, ESOMICS (**E**stimation of **S**uitable **O**ptimistic WCET for **MI**xed-**C**riticality **S**ystems), to obtain a suitable $WCET^{opt}$ for HC tasks in order to improve QoS and utilization and reduce the number of mode switches. In this scheme, an analytical approach is proposed to analyze the application time distributions, and then a new metric is defined to determine the appropriate value of $WCET^{opt}$ while reducing the probability of task overrunning. Based on the analysis, this chapter also proposes a Machine-Learning (ML) approach, which can then be generalized to any embedded application. In this approach, the model functionality and performance are evaluated based on the generated data sets to train and validate different prediction techniques. To the best of our knowledge, this is the first work that obtains $WCET^{opt}$ for MC tasks using the ML models while guaranteeing the real-time constraints, improving the QoS and reducing the number of mode switches. Although we focus on MC systems with two criticality levels, our scheme can be applied to an MC system with several criticality levels and different level of task criticality are mentioned in D0-178C [124].

Contributions: The main contributions of this chapter are following:

- Introducing a new metric based on the $WCET^{pess}$ and execution time distribution and conducting an extensive analysis of different applications through the new metric.

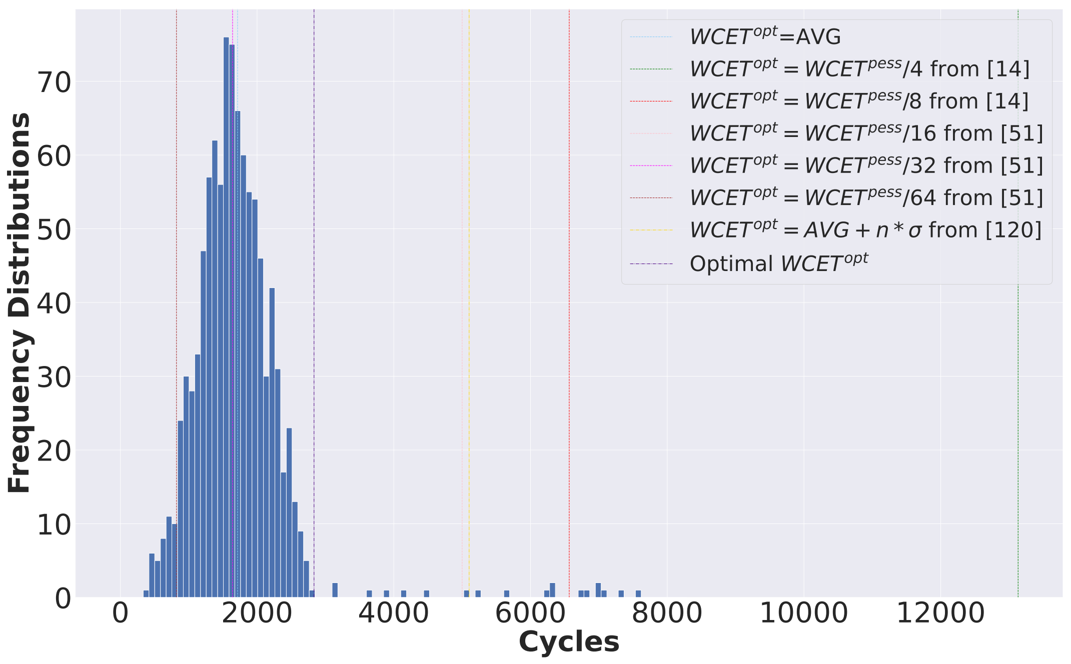


Figure 4.1: Execution time distribution for an application from Mälardalen benchmark, running on the Raspberry Pi 4 board.

- Proposing a data-driven approach to obtain $WCET^{opt}$ of MC tasks based on the ML model and the newly defined metric.
- Improving the QoS and resource utilization by scheduling more LC tasks in the system while reducing the number of task overruns.
- We implement ESOMICS for uni-processors using Raspberry Pi. Such an implementation can easily be integrated with any measurement-based timing analysis for uni-processors.
- Our proposed scheme, with extensive experiments, outperforms the previous research regarding the number of schedulable task sets and improving resource utilization. Our approach is evaluated for various state-of-the-art MC systems to show the prospect of our approach. Our implementation and all experimental data are publicly available.

Two fundamental concepts in MCS are low and high-criticality modes, as well as high and low-criticality tasks.

- **Low-Criticality Mode:**

  - In low-criticality mode, the system operates with relaxed constraints and can provide best-effort execution for low criticality tasks.
  - Low-criticality tasks are executed with reduced priority or resources compared to high criticality tasks.
  - The goal of low-criticality mode is to maximize system utilization and efficiency while meeting the safety requirements of high criticality tasks.

- **High-Criticality Mode:**
  - In high-criticality mode, the system operates under strict constraints to ensure the correct and timely execution of high- criticality tasks.
  - High-criticality tasks are allocated sufficient resources and given higher priority to meet their stringent safety requirements.
  - The transition to high-criticality mode may occur in response to changes in the system's operating conditions or triggered by specific events or deadlines.

- **High-Criticality Tasks:**
  - High-criticality tasks are characterized by stringent safety requirements, where failure to meet deadlines or correct execution can lead to catastrophic consequences.
  - Examples of high-criticality tasks include control tasks in autonomous vehicles, flight control systems in aircraft, medical monitoring systems, and emergency shutdown systems in industrial plants.
  - High-criticality tasks are allocated higher priority, resources, and redundancy to ensure their correct and timely execution, even under adverse conditions.

- **Low-Criticality Tasks:**
  - Tasks are less critical for system operation or safety and can tolerate longer response times or occasional delays without causing system failure or hazards.
  - User interface tasks such as displaying information on a screen or responding to user inputs. Background tasks like system maintenance, logging, or non-critical data processing.
  - Low-criticality tasks are scheduled around high criticality tasks, utilizing leftover system resources. They may be preempted or delayed to accommodate high-criticality tasks when necessary.

4.2 Related Work

There are various research works in the embedded real-time systems area that have focused on application timing analysis, like estimating WCETs. In order to explain the approaches briefly, the estimation of WCET using the implicit path enumeration techniques by modeling the control flow and the architecture using integer linear programming has been done in [12, 25, 94]. The different components of the architecture vary the execution time a lot, and the system's cache [6], branch prediction [15], and pipeline [92] effects have been studied in detail. The analysis of different WCET tools with their advantages and disadvantages has been presented in [145]. Once you have fixed the architecture and application, the next important factor is worst-case data which triggers WCET, as discussed in [82]. The authors presented an approach using a GA and ML model to estimate worst-case data for the application efficiently.

Besides, applying ML and artificial intelligence to estimate WCET is also prevalent in the research, as these are emerging technology in every field of our lives because of their ability to solve complicated problems with better accuracy. Altenbernd P. et al. [7] proposed a method to estimate the WCET of the application using source-level timing analysis. Most of the approaches, such as [22, 66, 96], used an ML model to estimate WCET from the application's source or assembly level—the work using the neural networks approach presented in [80]. The authors in [8] proposed WE-HML, a hybrid WCET estimation technique for architecture with caches in which the longest path is estimated using static techniques, whereas the ML model is used to determine the WCET of basic blocks. The contention prediction using Quantile Neural Networks in multiprocessor proposed in [23]. The proposed approach reduces the risk of detecting timing misconfiguration in late phases of the development process that would result in costly changes to the system design. The author in [35] proposed ACETONE which is a predictable programming framework for safety-critical systems using ML models. The proposed approach is capable of generating readable and traceable code.

From the MC system design perspective, as mentioned in the Introduction section of this chapter, different levels of WCETs are defined for each application to guarantee the real-time constraints of tasks with different criticality levels. In order to estimate the pessimistic WCETs, most of the above approaches can be employed. However, few works have addressed determining the WCET for lower criticality levels, like determining the $WCET^{opt}$ in dual-criticality systems. As an example, [14, 51, 97] have determined the $WCET^{opt}$ as a percentage of $WCET^{pess}$. In addition, in [120], researchers have claimed that obtaining the $WCET^{opt}$ as a percentage of $WCET^{pess}$ is not an efficient method; therefore, they determined the $WCET^{opt}$ based on Average-Case Execution Time (ACET) by using the Chebyshev's theorem. Their approach estimates the mode switching probability as very pessimistic about being applied to any application. However, since real-time applications are known in advance in embedded systems, they can be analyzed at design-time for $WCET^{opt}$ determination. Besides, some approaches [48, 63] determined the $WCET^{opt}$ at run-time based on the application behavior. However, these methods' goal is to postpone the mode switches for a long time to guarantee the minimum QoS, not improving the QoS by scheduling more LC tasks in the system.

As a result, we propose a novel approach to first analyze the application's timing distribution at design-time and then determine an appropriate value of $WCET^{opt}$ based on the analysis through ML techniques to improve the QoS and reduce the probability of mode switching.

4.3 Application and System Model

We consider a dual-criticality system with two distinct levels of criticality, referred to as LC denoting low criticality, and HC denoting high criticality, respectively. The terms applications and tasks have been used interchangeably in work. There are n MC tasks τ=$\{\tau_1, ..., \tau_n\}$ running on uniprocessor systems, and other than the processor, the tasks are independent and do not share any resources. Each task τ_i has a period P_i and a relative deadline D_i. A pair of WCETs ($WCET_i^{LO}$ and $WCET_i^{HI}$) is used to indicate its worst-case computation requirements in the LO and HI running modes, just like in the traditional MC model. We can represent each task as a tuple (P_i, D_i, ζ_i, $WCET_i^{opt}$, $WCET_i^{pess}$), where:

- P_i is the period of the τ_i.
- D_i is the relative deadline of task τ_i, D_i=P_i.
- ζ_i is the set of criticality level.
- $WCET_i^{opt}$($WCET_i^{pess}$) denotes the WCET of the tasks τ_i in LO(HI) mode.

In this system model, we have two WCETs for each HC task τ_i where $WCET_i^{pess} \geq WCET_i^{opt}$. The utilization of HC and LC tasks is defined as $u_i^{LO} = \frac{WCET_i^{opt}}{P_i}$ and $u_i^{HI} = \frac{WCET_i^{pess}}{P_i}$, respectively. The system begins with operating in LO mode, with all tasks (LC and HC) being executed before their deadlines. Whenever an HC task exceeds its $WCET_i^{opt}$, the system switches from LO to HI mode immediately to provide all the compute resources to HC tasks to guarantee their execution to prevent any damage, and all the LC tasks are possibly guaranteed with degraded resources.

4.4 Proposed Approach

As mentioned earlier, estimating the suitable WCET^{opt} for HC tasks is a significant challenge. To address it, the proposed approach designs the MC systems and analyzes the tasks of the application to choose the suitable WCET^{opt} based on the ML model, which improves the task overrun and executes more LC tasks. Our data-driven and analytical approaches are divided into many steps. The analytical phase is presented in Subsection 4.4.1. The first step in the data-driven approach is the training phase, explained in Subsection 4.4.2.1. The second step is the inference phase described in Subsection 4.4.2.2. The generation of training data is explained in Subsection 4.4.3. The ESOMICS Algorithm is presented in the Subsection 4.4.4. The system objective analysis is discussed in Subsection 4.4.5.

4.4.1 Analytical Approach

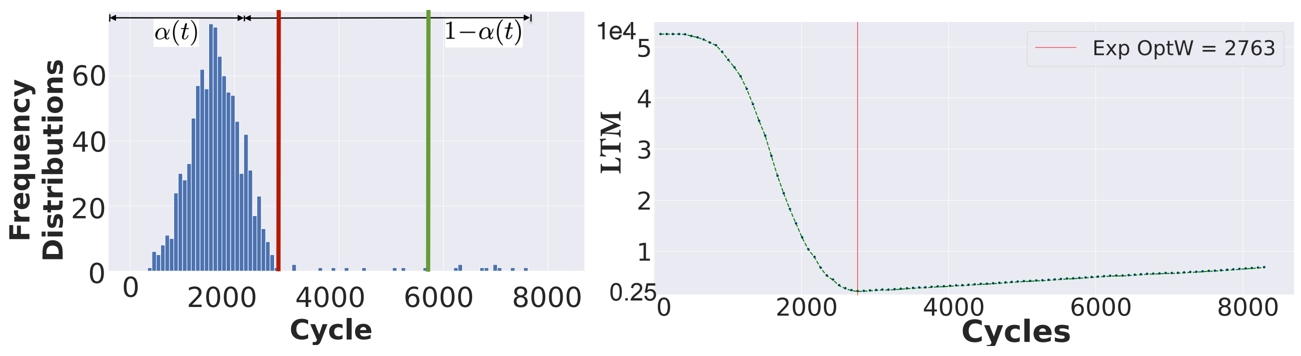
Determining the appropriate value of $WCET^{opt}$ is essential in improving the timing behavior of MC systems, mode switching probability, and utilization. Therefore, supposing the application function is known, we propose an analytical approach that relies on an extensive experiment-driven method to establish a new metric for determining the appropriate $WCET^{opt}$ for each HC task. Fig. 4.2a shows the execution distribution of ns application from the Mälardalen benchmark, with 1000 instances, presented in the motivational example of Section 4. The suitable $WCET^{opt}$ corresponds to the minimum cycle in which the majority of sample executions are less than it. In Fig. 4.2a, $\alpha(t)$ is the ratio of frequency of number of samples executed at time t to the total execution frequency/Sample (presented in Eq. (4.1)). To this end, we introduce a new metric, Lowest cycle Time for Majority of samples (LTM), in Eq. (4.2) to obtain $WCET^{opt}$.

$$\alpha(t) = \frac{\text{Frequency of \# of Samples Executed at Time}(t)}{\text{Total Sample}} \tag{4.1}$$

$$LTM(t) = \alpha(t) * WCET^{curr}(t) + (1 - \alpha(t)) * WCET^{pess} \tag{4.2}$$

where WCET^{curr} is the execution cycle of the task at time t. The WCET^{pess} of the application is constant and obtained from SWEET tool [95]. Eq. (4.2) illustrates that LTM at time t equals the scenario in which α percent of samples complete execution before $WCET^{curr}$, while the remaining $(1 - \alpha)$ complete before $WCET^{pess}$. We focus on finding the minimum time cycle, which covers the maximum execution distribution to improve processor utilization and reduce the mode switches.

To provide a clear demonstration, in Fig. 4.2, when considering $\alpha(t)$=0.98, then WCET^{curr} =



(a) Frequency distribution (b) LTM metric

Figure 4.2: An application analysis

2850 cycles and LTM=1×10^3. If $\alpha(t)$=0.99, then WCET^{curr}=5705 cycles and LTM=6×10^3. Indeed, in this case, the WCET^{curr} appears to be more pessimistic, and higher LTM value, compared to the previous scenario, and leads to poor processor utilization and scheduling fewer LC tasks in the system. Avoiding such pessimism is crucial when designing systems, as even 1% increase in α can result in nearly doubling the execution cycles. Any values of α below 98% result in an increase in mode switches, which is undesirable as it hinders the desired QoS. Our goal is to determine the minimum value of LTM as it corresponds to the proper WCET^{opt}, shown in Eq. (4.3).

$$WCET^{analy} = WCET^{curr}(t)|_{LTM\ is\ minimum} \tag{4.3}$$

WCET^{analy} represents that for any application, the cycle corresponding to the minimum LTM value serves as the optimal cycle for the WCET^{opt} since it signifies the high processor utilization and fewer mode switches, as observed in experiments.

Algorithm 2 shows the pseudo-code of the analytical approach. As inputs, the algorithm takes the application for which we aim to find $WCET^{analy}$, along with the platform and the variable N, which denotes the total number of application executions. The output is the application's $WCET^{analy}$. The procedure commences by executing the application N times to measure both the frequency for each cycle (lines 2-4). For each time from one to $WCET^{pess}$, α and LTM values are calculated (lines 7,8). The calculated LTM is compared with the variable $LTMmin$ value, and if the variable is changed, the cycle of this lower LTM value is assigned to T^{opt} (lines 9-11). This process is iteratively repeated until we find the optimal value of T^{opt} and assign it to $WCET^{analy}$ (line 13).

Algorithm 2 Analytical Approach

Input: Application, Single Processor Platform, N
Output: $WCET^{analy}$

```
1: Procedure AnalyticalMethod()
2: for i =1 to N: do
3:     {freq} = execute(Application)
4: end for
5: $T_{opt} = WCET^{pess}$; $LTMmin = \infty$;
6: for T = 1 to $WCET^{pess}$ do
7:     $\alpha(T)$ = freq($T$) / Total Frequency {Eq. 4.1}
8:     $LTM(T) = \alpha(T) * T + (1 - \alpha(T)) * WCET^{pess}$ {Eq. 4.2}
9:     if $LTMmin > LTM(T)$ then
10:        LTMmin = LTM(T);   $T_{opt} = T$;
11:    end if
12: end for
13: $WCET^{analy} = T_{opt}$
14: end Procedure
```

4.4.2 ML-based Approach

Determining $WCET^{analy}$ for each application through repeated executions can be time-consuming and challenging. Therefore, we propose an ML-based approach, which provides a more efficient alternative by enabling us to run each application only once to obtain its WCET^{opt}. This reduces the timing overhead, making it a practical, and time-saving solution for obtaining WCET^{opt} without needing multiple executions.

4.4.2.1 Training of the Models

The training phase of the ML-based approach is performed once per target architecture. It is mainly focused on learning the timing model of the processor, as shown in Fig. 4.3. Firstly, the selection of training programs has been made through GENE [140]. It is a tool that provides both WCET benchmarks as well as their flow facts. It is able to achieve this because instead of analyzing existing programs, the tool generates new benchmarks by relying on small building blocks (for which the flow facts are known) and combining them in a way that allows the tool to determine the flow facts of the resulting complex benchmark automatically. Typical design and implementation patterns that have been taken from WCET benchmark suites and current real-world applications serve as the foundation for Gene's creation of realistic benchmarks. The selected programs are given to SWEET tool [95] to extract the features. The main function of SWEET is flow analysis, whose results are flow facts, i.e., information about loop bounds and infeasible paths in the program. It analyzes programs in ALF format, which is an intermediate program language format specially developed for flow analysis. The chosen program is compiled and linked in the third step to generate the execution cycle as a label. Besides, $WCET^{analy}$

Table 4.1: Extracted Features

# addition operations	# division operations	# subtraction operations
# logic operations	# function calls	# return statements
# load operations	# store operations	# multiplication operations
# shift operations	# jump statements	# comparison operations

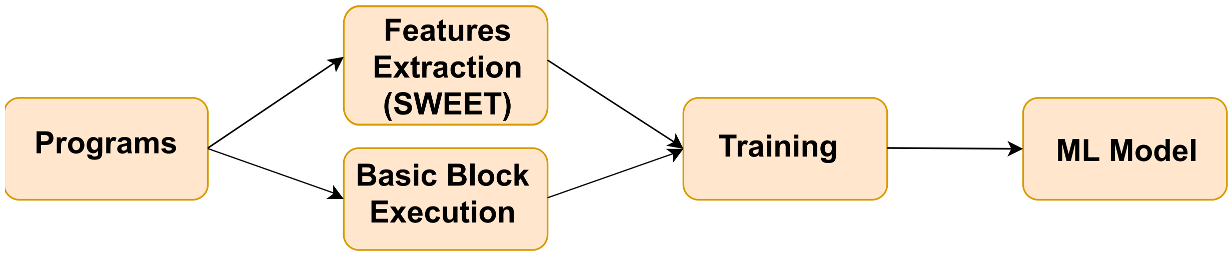


Figure 4.3: Learning of the ESOMICS

for any given application is fed up to ML models in this training phase as a label for a dataset. In the fourth step, features and labels are fed to the ML model for training. We have used five ML models trained on the large data set, which covers a large variety of code structures in real code. After training, the various ML algorithms can estimate different programs' $WCET^{opt}$. ML models are trained using the numerical quantities called features, and some are listed in TABLE 4.1. To understand how the training part works, we need to explore some statistics behind it. As mentioned, we already have features and labels for each training example. We assume each training example has n instructions $i_1, ..., i_n$ as features with the total observation of size m. Different training example has measured observed cycle $T = (T_1, ..., T_m)$. The model then predicts $IC_{mxn} * W_{nx1}$, where IC is an $m \times n$ matrix whose rows are observed array of instruction counts and, $W = (w_1, ..., w_n)$ is an array of cycles that minimizes the deviation from real. The main goal here is to reduce the $IC * W$ from T to get the best model, as explained in [7]. The standard ML models have been used for the training in this work, described in detail in the next section.

4.4.2.2 Inference of the Models

The model evaluation is shown in Fig. 4.4. Firstly, benchmark programs or unseen programs not used during the training phase are carefully selected. These selected programs are utilized to infer and evaluate the model. The features are extracted for the selected benchmark programs using the SWEET tool [95]. These features are given to the trained ML model to predict WCET^{opt}.

We have different trained ML models. WCET^{ML} is obtained from the best-chosen prediction

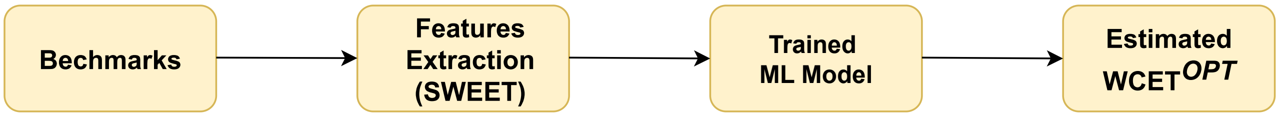


Figure 4.4: Evaluation of the ESOMICS

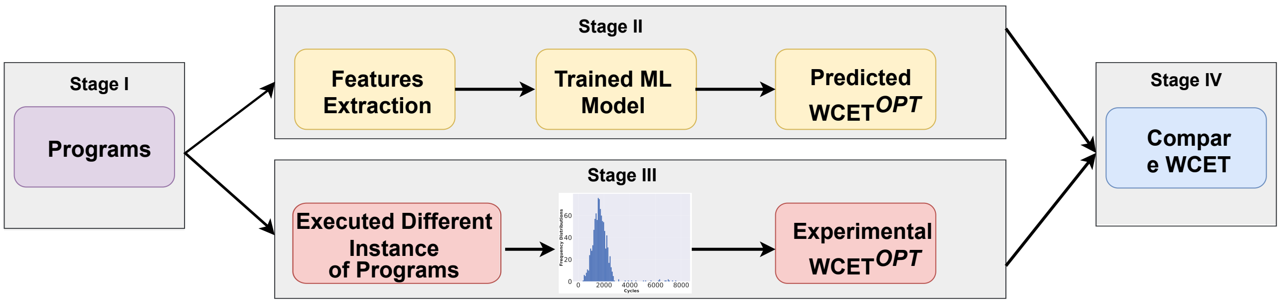


Figure 4.5: An overview of analytical and inference approaches

model. Once the different ML models have been trained, we require a parameter to determine the model that performs best and provides superior accuracy. This is achieved in our case by Mean Square Error (MSE) to find the model that gives the slightest error on training and validation data. The MSE of each model is calculated using predicted and observed data. To validate whether the chosen model is best, we have compared it with WCET^{analy}, which needs to be carefully examined through experiments.

Fig. 4.5 shows a summary to provide a comprehensive overview of analytical and ML-based approaches. The scheme comprises several distinct stages, which are outlined below:

- *Stage I*: The selection of the programs is made in this stage. This program is an unseen program or benchmark whose WCET^{opt} we are eager to determine.
- *Stage II*: The features are extracted from the selected programs using SWEET tool [95]. The features are given to the trained ML models in the next step. The trained model predicts the WCET^{ML}, which is equivalent to WCET^{opt}.
- *Stage III*: The same selected programs are executed different numbers of times using different inputs to generate the execution distribution plot. The generated plot in the next step is observed with the derived Eq. (4.2) to get WCET^{analy}.
- *Stage IV*: The WCET^{ML}s and WCET^{analy} for stages II and III are compared to find the proper WCET^{opt}. The comparison is essential due to the variability in the WCET^{ML} values obtained from different trained ML models in Stage II. The primary objective

is to identify the WCET^{opt} value from an ML model that its value closely aligns with WCET^{analy}.

Stage III illustrates how good the proposed ML-based approach is in terms of accuracy. Note that the results of our approach, which is deployed on the target hardware, do not need to perform at this stage. A comprehensive analysis of approaches has been done in the next section.

4.4.3 Generation of Training Data

Due to a lack of training benchmarks, ML methods utilized in earlier research articles [7, 66, 80, 84] for WCET prediction have experienced constraints. This restriction results from the training data being too few and maybe too undifferentiated, which could lead to worse training quality. Our strategy, which is similar to the one described in [140], [8], solves the problem by using a sizable dataset of automatically created basic blocks as training data. By utilizing a more extensive and diverse dataset, our approach aims to improve the training quality and enhance the accuracy of WCET estimation. The generated code employs all standard basic kinds, including arrays of basic types and char, short, int, and long in both signed and unsigned variants. The most popular C operations are addressed, including binary and unary operators on booleans, array indexing, shift and rotate operations, and arithmetic and logical operations. Listing 1 shows an example of generated application used for training with the help of GENE [140].

The first line in the main function is a call to a special register for measuring the number of cycles taken by the given application. It contains a set of variables and applies operations on these variables. It also contains if statements to cover branch instructions. However, the generator ensures that there is no data-dependent execution. Similarly, the other generated application contains different statements and operations. With the help of the SWEET tool, we extract the features and using a special register, and we measure the execution time of these generated applications.

```c
int main(){
var_2 = ( uzero != uone ) ? var_2 : var_1;
var_2 = array_0[array_index] - array_0[array_index];
if (zero < one) {
    array_0[array_index] = var_2 == small_int;
    var_2 = var_2 * var_2;
    }
return 0;
```

Algorithm 3 ESOMICS Approach

Input: Applications, Single Processor Platform, n
Output: $WCET^{opt}$

```
1: Procedure ESOMICS()
2: $cg = DataGenerator(n)$
3: features = $SWEET(cg)$
4: labels = $AnalyticalMethod(cg)$
5: MLFunc = $Training$(features,labels)
6: $WCET^{opt} = Testing(MLFunc(Application))$
7: end Procedure
```

```
}
```

Listing 4.1: Example of generated application for training

4.4.4 ESOMICS Algorithm

Algorithm 3 outlines the pseudo-code of ESOMICS scheme, which includes code generation, analytical and ML-based approaches. As inputs, the algorithm takes the applications, platform, and variable n, representing the total dataset generated for training ML models. $WCET^{opt}$s are the output. The procedure begins with variable cg, which is assigned with running the function *DataGenerator* (line 2). The *DataGenerator* function uses GENE to generate dataset. The created dataset is then fed to the SWEET tool [95] to extract the features (line 3). The same dataset is then fed to $AnalyticalMethod$ to generate the labels (line 4). Once the features and labels are extracted, it is fed for $Training$ to ML models (line 5). After the training phase, the function $Testing$ is ready to generate $WCET^{opt}$ for each application (line 6).

4.4.5 System Objectives Analysis

Improving the timing behavior of the system is driven by the following two primary objectives [120]:

- Resource Utilization is improved by achieving a substantial increase in the utilization that can be assigned to LC tasks in the LO mode (U_{LC}^{LO}). U_{LC}^{LO} is upper-bounded by the schedulability constraints in both LO and HI modes [14, 120] (shown in Eq. (4.4)). U_{HC}^{LO} is computed based on the $WCET^{opt}$.

$$U_{LC}^{LO} \leq min\{1 - U_{HC}^{LO}, \frac{1 - U_{HC}^{HI}}{1 - U_{HC}^{HI} + U_{HC}^{LO}}\} \tag{4.4}$$

- Mode Switching Probability Reduction has a beneficial effect on the performance or functionality of MC systems by reducing frequent drops of LC tasks in the HI mode. P_i^{MS}

is the exceeding probability of task τ_i from $WCET^{opt}$, and P_{Sys}^{MS} is the mode switching probability of the system. Since tasks are independent, P_{Sys}^{MS} is computed as follows [120]:

$$P_{Sys}^{MS} = 1 - \prod_{\zeta_i \in HC} (1 - P_i^{MS}) \tag{4.5}$$

To improve the system timing behavior, $(U_{LC}^{LO} \times (1 - P_{Sys}^{MS}))$ equation should be maximized [120]. As a result, an approach capable of obtaining appropriate values for $WCET^{opt}$, should yield a higher value for this equation.

4.5 Evaluation

We presented the experimental setup to evaluate the effectiveness of our proposed scheme for the Raspberry Pi 4 platform in this Section. We described the hardware and software environment used in the experiments. We evaluated our scheme on the real benchmarks called the Mälardalen benchmark suite. The Hardware platform and software environments are discussed in detail in Subsection 4.5.1. The ESOMICS evaluation with results are presented in the Subsection 4.5.2. The comparison with other policies is explained in Subsection 4.5.3.

4.5.1 Hardware Platform and Software Environments

The Raspberry pi model B 4 relies on a Broadcom BCM2711 SoC, which is based on a 1.5 GHz 64-bit quad-core ARM Cortex-A72 processor, a 2-wide superscalar processor. The architecture has a specific private L1 cache and a 1 MB shared L2 cache. The Raspbian Lite Operating System (OS) runs on the Raspberry 4, which is a Linux kernel version which better than other OS in terms of overhead generated on the timing. The compiled code is then executed on a specific core. When compiling the benchmarks, the compiler optimizations were disabled to facilitate the flow of information during the WCET analysis. This feature will be revisited in future work and is also open to research.

A total of 15000 programs were generated for the training of the model. We have made sure that the construction and selection of the training program cover all the instructions for the ARM Cortex-A72 processor. As the training process only needs to be completed once for each architecture and can easily be carried out simultaneously on numerous boards, we do not consider the delay in running the ML model. The training programs are executed on the ARM cortex processor to measure the CPU cycle, which is used as labels. The same training programs are translated into virtual instructions using SWEET, and features are extracted. The prediction model quality depends on the input training data, which needs to be biased-free as much

Table 4.2: Experimented ML Algorithm

Algorithm	**Description**
Decision Tree (DT)	A single decision tree
Random Forest (RF)	A multitude of decision tree
K-Nearest Neighbors (KNN)	K-Nearest Neighbors regression
Adaptive Boosting (AdaBoost)	Boosting algorithm with simpler dataset
Gradient Boosting (GB)	Boosting algorithm with complex dataset

as possible. The features need to be extracted carefully as it leads to the bias in the system. We also normalized our data values in the range of 0 to 1 to avoid the prediction model's inclination towards the features having larger values. WCET estimation is evaluated on the ML algorithm provided by the Scikit library. Preparatory experiments made us select the five algorithms that gave the best result. A selection of five ML algorithms based on Mälardalen benchmark [52], such as Decision Tree (DT), Random Forest (RF), K-Nearest Neighbor's (KNN), AdaBoost, and Gradient Boosting (GB), has been made as shown in Table 4.2. The abbreviations DT, RF, KNN, AdaBoost, and GB will be used throughout the remainder of the chapter in place of the full names. None of the above methods outperforms the others for all the benchmarks. However, the model that gives better accuracy is chosen as the best model. The 5 ML algorithms' training took about 75 minutes to complete on a macOS operating on a MacBook Air with a 1.8 GHz Dual-Core Intel Core i5 processor.

4.5.2 ESOMICS Evaluation

Table 4.3 reports the predicted WCET^{opt} for the seven benchmarks using five ML models and the analytical approach presented in Section 4. The first column represents the different benchmarks used in the experiment. The column from second to fifth represents the predicted cycles for different ML approaches obtained using our trained ML models. These values represent how many cycles each prediction model takes to give WCET^{opt}. The last column represents the observed cycles using the analytical approach. This value is obtained by equation 2, in which each application is executed to find the cycle at which *LTM* is minimum. However, the best model DT overestimates the WCET^{opt} by 25% on average on the given benchmarks, which shows promising results compared to other prediction models and we keep the analytical approach as the base for the calculation of percentage. This is because the observed cycle from the analytical approach is the suitable WCET^{opt}, and our goal here is to predict that value as close to it.

Table 4.4 shows the percentage deviation between the Analytical and ML approach on the metric *LTM* given in Eq. 4.2. The first column represents the benchmark's name. The second

Table 4.3: Predicted Cycle using ML and Observed Cycle using Analytical approach for the Mälardalen benchmark

	Predicted Cycle					Observed Cycle
Benchmarks	**DT**	**RF**	**KNN**	**AdaBoost**	**GB**	**Analy. Approach**
cnt	1,875	3,832	841	1,000	11,400	1,616
compress	5,175	4,047	1,010	1,069	1,328	4,797
duff	5,175	3,904	1,041	1,015	18,582	3,597
expint	1,109	1,310	845	1,008	6,811	781
fdct	11,012	5,732	979	1,791	76,308	6,371
insertsort	1,356	1,181	1,073	992	62,364	1,265
ns	5,175	3,844	954	1,015	15,129	2,763

Table 4.4: Percentage Deviation between Analytical and Best ML approach for the Mälardalen benchmark on the metric *LTM*

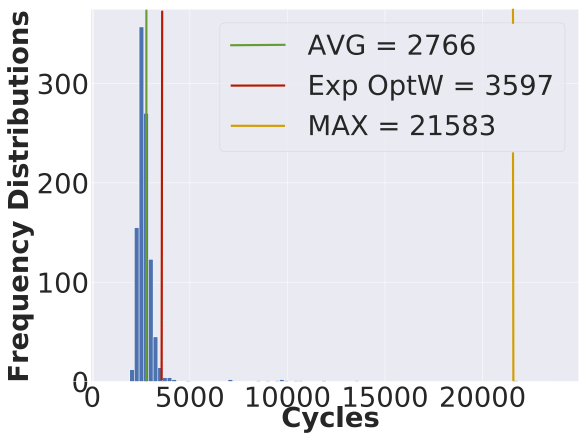
Benchmarks	**Analy. Approach**	**ML Approach**	**% Deviation**
cnt	1,78,909	1,98,884	10.04%
compress	2,13,488	2,19,159	2.58%
duff	4,47,067	5,62,772	20.55%
expint	2,50,414	2,76,326	9.37%
fdct	6,68,570	11,05,879	39.54%
insertsort	1,28,717	1,35,907	5.15%
ns	5,10,209	6,73,774	24.27%

and third columns are the *LTM* values from the analytical approach and the best prediction model from the ML approach. The percentage deviation between them is shown in the fourth column. We observed from the table that the deviation between the above two is very minimal, and Our prediction model is the best choice of all the approaches to set WCET^{opt} because, in the worst-case scenario, the maximum and minimum percentage deviation is 39% and 2%, respectively. The average percentage deviation reported for these benchmarks is 15%.

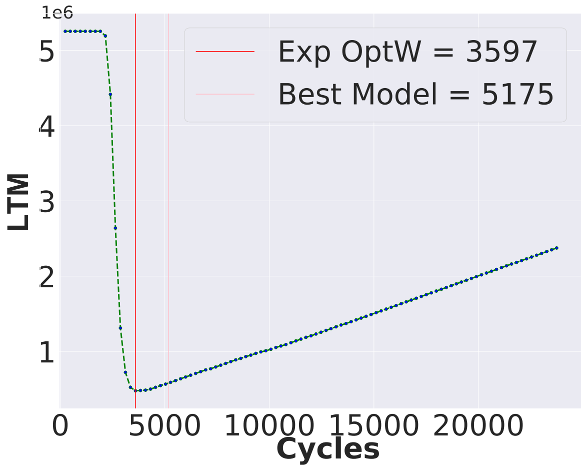
Figure 4.6 shows the analytical analysis of the duff application from Mälardalen benchmarks using the metric presented in Eq. 4.2. Figure 4.6a represents the execution distribution on the Y-axis, and the clock cycle is on the X-axis. The vertical line on the graph represents the average, analytical (Exp Optw means experimental WCET^{opt} from Analytical approach), and maximal observed cycles for the application in the color green, red and brown, respectively. The *LTM* value is represented on the Y-axis, and the clock cycle is on the X-axis of Figure 4.6b. The red and magenta vertical lines represent the analytical and best ML model intersecting the *LTM* on the Y-axis. We observed that both lines are close to each other and proving our

Table 4.5: Comparison between ACET and WCET of different applications

Benchmarks	ACET (Cycle)	Pessimistic-WCET (Cycle)	Standard-Deviation (Cycle)	Percentage (%) of Samples that Overruns if the Optimistic WCET is set to:					
				ACET	DT	RF	KNN	AdaBoost	Gradient Boosting
cnt	1,441	20,849	2.30×10^2	24.20	0.7	0.2	100.00	100.00	100.00
compress	4,268	15,640	7.67×10^2	27.10	2.7	73.5	100.00	100.00	100.00
duff	2,766	40,000	10.38×10^2	29.9	1.3	1.9	100.00	100.00	100.00
expint	398	13,737	1.73×10^2	37.0	0.3	0.2	0.3	0.3	100.00
fdct	5,845	16,861	9.02×10^2	30.3	1.1	58.1	100.00	100.00	100.00
insertsort	1,219	4,433	1.40×10^2	45.4	0.1	100.00	100.00	100.00	100.00
ns	1,714	52,531	16.94×10^2	49.3	21.0	49.3	96.1	95.4	0.1



(a) Frequency distribution



(b) LTM metric

Figure 4.6: Analysis of duff application

approach to be suitable for optimistic WCET. We have only shown one application analysis here, and the remaining application analysis is shown in the appendix Section A.

We executed 1000 instances of seven applications, and the ACET and WCET of them in terms of CPU cycle are reported in Table 4.5. The standard deviation of each application is also presented. WCET of each application is estimated by SWEET. We observed that for each application, how many instances violate the WCET^{opt} when it is set to ACET or the predicted cycle from the ML model.

We observed from the table that KNN and AdaBoost is not appropriate parameter to set WCET^{opt}. For instance, the mode-switching probability for applications countnonnegative (cnt), compress, and insertion sort is 100%, while it is less than 1% for expint. Similarly, this is the case with Gradient Boosting. On the other hand, when the WCET^{opt} is set to ACET, the mode-switching probability behavior is easier to predict. Too many system mode changes, however, result from just setting WCET^{opt} equal to ACET. However, When we set WCET^{opt} to Decision Tree, the probability of mode switching is very minimal, and processor utilization is improved because more LC tasks are allowed to execute without degrading the QoS. It is

Table 4.6: Percentage of tasks overruns for different approaches

Benchmarks	**Analy. Approach**	**ML Approach**	[14]	[51]	[98]	[120]
cnt	0.9%	0.7%	0.1%	100.0%	0.0%	0.3%
compress	4.3%	2.7%	96.1%	100.0%	1.4%	0.4%
duff	2.4%	1.3%	0.6%	68.7%	0.1%	0.2%
expint	0.5%	0.3%	0.0%	0.3%	0.0%	0.1%
fdct	3.2%	1.1%	100.0%	100.0%	2.5%	0.6%
insertsort	0.7%	0.1%	100.0%	100.0%	0.0%	0.0%
ns	2.0%	1.2%	0.0%	1.7%	0.0%	0.0%
P_{Sys}^{MS}	13.24%	7.19%	100%	100%	3.96%	1.59%

also observed that in the case of DT, the maximum percentage of the sample overrun is 21% in the worst scenario. To evaluate whether our prediction model is accurate, we have compared it with our analytical approach.

4.5.3 Comparison with other policies

We compare our method with state-of-the-art approaches in terms of estimating WCET^{opt}, processor utilization, and percentage of tasks overrun. As discussed in Section 2, most of the state-of-the-art research has defined WCET^{opt} as a ratio of WCET^{pess}. For instance, if we define $\lambda = \frac{WCET^{opt}}{WCET^{pess}}$, the ratio of these two values provides insight into the degree of variation or uncertainty in task execution times within the system. A higher ratio indicates greater variability or uncertainty, while a lower ratio suggests more predictable and deterministic behavior. The researchers in [98] have assigned $\lambda \in [\frac{1}{2.5}, \frac{1}{1.5}]$ in their experiments. Two different ranges for λ have been assigned $\lambda \in [\frac{1}{4}, 1]$ and $\lambda \in [\frac{1}{8}, 1]$ in research [14]. Different amount of ranges have been assigned for $\lambda = \{\frac{1}{16}, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1\}$ in research [51]. Since the method for calculating WCET is the same for all articles, We select [120] as the standard research methodology for the experiments. In our method, WCET^{opt} is chosen from the best prediction model trained on large data set once for the target architecture. The trained model is then used to predict WCET^{opt} for unseen applications. In Ranjbar's approach [120], the WCET^{opt} is based on the Chebyshev theorem using ACET and the standard deviation of the application with tuning parameter n. The optimal n value in our work is set to 8. Figure 4.7 shows the comparison of the estimated WCET^{opt} between our and the state-of-the-art approach. We observed that our analytical and data-driven approach shows appropriate WCET^{opt} estimation in comparison with other approaches. The major drawback of [120] is to tune the n parameter and also needs to execute each application which is not the case in our approach. Once the model is trained, it can be used multiple times for any application to estimate the WCET^{opt}.

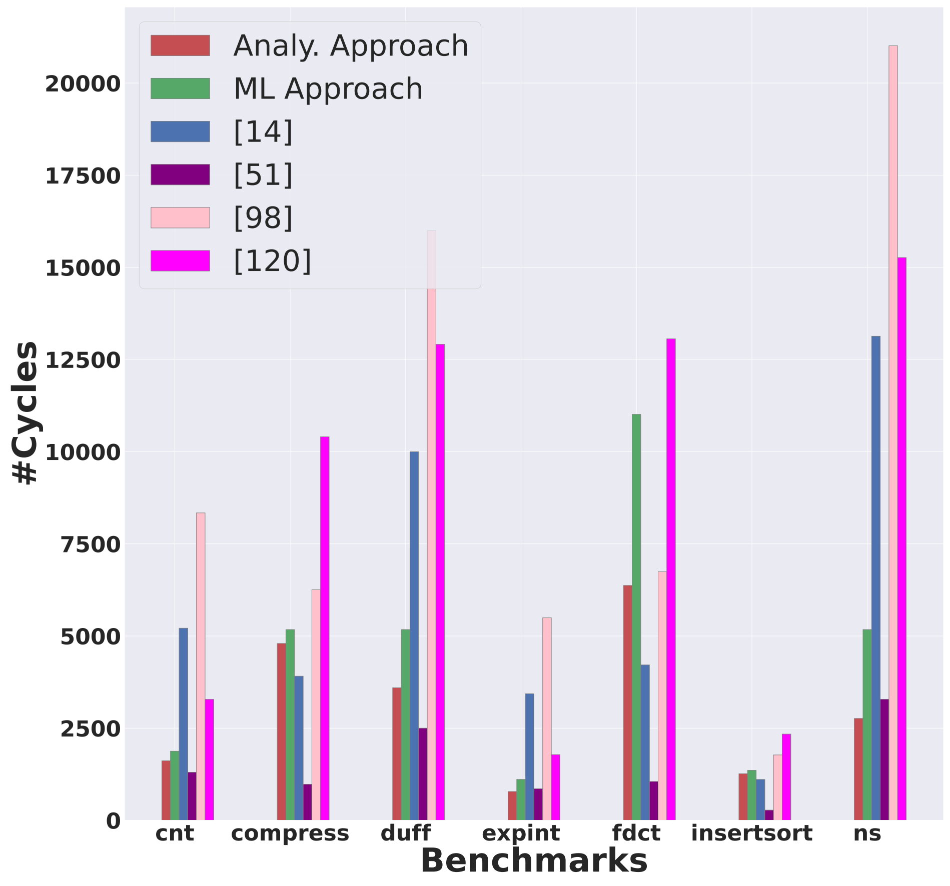


Figure 4.7: Comparison of different approaches for determining WCET^{opt}

Table 4.6 and Figure 4.8 show the percentage of tasks overrun out of 1000 instances of the applications by different approaches if we set WCET^{opt} according to their methods and the utilization of the system, respectively. We observed that in [98] and [120], the percentage of overrun is minimum, corresponding to less mode switching, but the processor utilization also decreases the authors in both approaches choose the WCET^{opt} to be too pessimistic. In [14] and [51], we observed that the percentage of overrun is maximum, corresponding to more mode switching, but processor utilization increases because, in both approaches, the chosen WCET^{opt} is too optimistic. Therefore choosing the suitable WET^{opt} is paramount for avoiding more mode switches and less processor utilization of the system. Our approaches have achieved the trade-off between less mode switching and more processor utilization. The percentage of tasks overrun

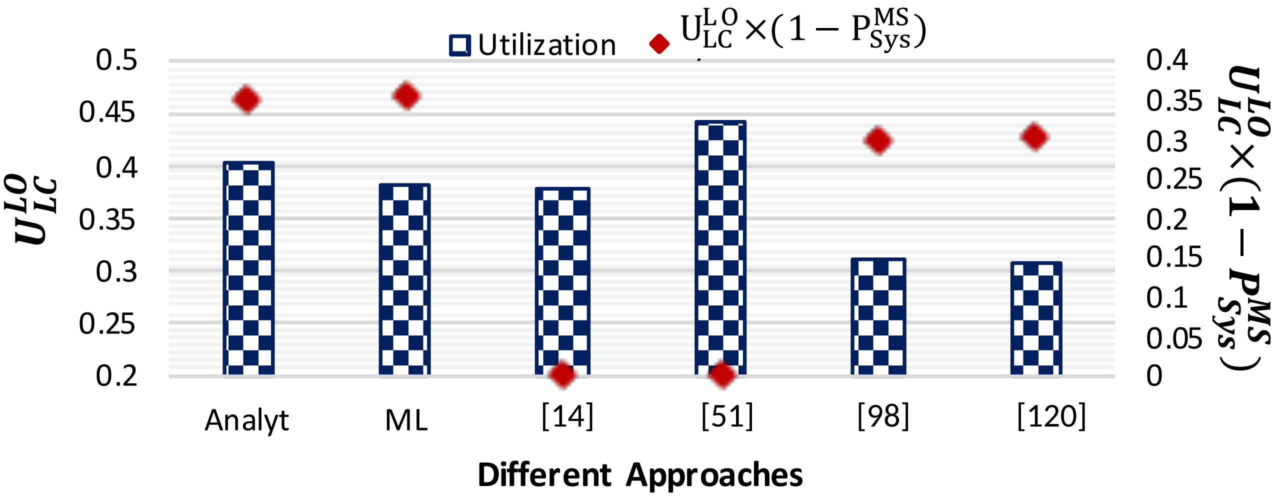


Figure 4.8: U_{LC}^{LO} and system goal based on two objectives for different approaches

in the worst-case scenario for the analytical and ML approach is 4% and 3%, respectively. The improvement of processor utilization is 14% and 7% compared with other policies using our analytical and ML approaches. The reason for having better processor utilization and fewer mode switches in our approaches is because of estimating the suitable WCET^{opt} for HC tasks and executing more tasks in the systems.

4.6 Summary

In this chapter, we presented ESOMICS, an approach that can be used to predict WCET^{opt} using machine learning models. The model estimates WCET^{opt} from the source code of the applications, and our analytical approach verify its correctness using the newly defined metric *LTM*. Features are generated using the SWEET tool, i.e., the number of statements and operations in the source code, which are fed into our ML models to predict WCET after scaling data. We demonstrate the model for ARM processors. We have used the Scikit library to implement all the ML models. The model performance is evaluated on real benchmarks, and our approach outperforms all the previous research in terms of estimating appropriate optimistic WCET, system utilization, and percentage of tasks overrun. The proposed scheme, on average, exceeds 25% and 15% on the prediction cycle and the *LTM* metric, respectively, on the benchmarks. We investigated the trade-off between the percentage of tasks overrun and processor utilization which is improved by 14% and 7% while percentage of task overrunning in a worst-case mode is 4% and 3% in both analytical and ML approaches, respectively.

We believe that machine learning models can be applied to improve these results further. Different neural networks like Graph Neural Networks (GNN) and Recurrent Neural Networks

(RNN) can be used to get better accuracy on the performance of the proposed methodology. This is a subject for future research because it has good potential for WCET analysis.

Chapter 5

Estimation of Worst-Case Data for WCET

Temporal Verification of Real-Time and Embedded Systems requires the determination of Worst-Case Execution Time (WCET). The strict timing requirements set by the regulations are met by the design of these systems. A system's delay resulting from failure to meet the deadline will cause catastrophic outcomes. The worst-case data, which provides the maximum execution time, is essential for WCET estimation. The complexity of an evolutionary algorithm requires the use of several computational resources.

In this chapter, we present a novel method to replace the simulator-based actual execution with a predictive model trained using the samples acquired on the simulator. This method reduces the overall time required to generate Worst-Case Data. Different machine learning models are trained to integrate with genetic algorithms. Our machine-learning models are created using the Pygad Framework. The feasibility of the proposed approach is validated using benchmarks from different domains. The results show the speedup in the generation of Worst-Case Data up to 24 times, and the best accuracy achieved is 98%.

In the rest of chapter, the problem statement, motivational example and contributions are presented in Section 5.1. Section 5.2 describes evolutionary techniques for testing real-time systems. Section 5.3 elaborates our methodology to generate the worst-case data. Section 5.4 describes the experimental setup and the selection of certain specific parameters during experiments. The evaluation of machine learning models are presented in Section 5.5. Finally, we conclude the chapter in Section 5.6.

5.1 Introduction

Real-Time and Embedded Systems play an essential role in our lives. We are surrounded by control systems like railways, automotive, avionics, telecommunications, and medical care. Our lives depend on them; therefore, making it a reliable system is of paramount importance. WCET is the maximum execution time an application takes to complete. Estimating WCET [145] is a difficult task as it depends not only on the architecture configuration on which we are executing the application but also on the input we are providing. The input data which produces the maximum execution time plays a vital role in estimating the WCET [76]. Once we fix the hardware architecture and the application, the next thing is finding the input data that produces WCET. Traditionally, the Measurement and the Static approaches are used to estimate the WCET. In the Measurement approach, the application is executed on the hardware or the simulator to measure the execution time, but a guarantee of WCET is not possible. Whereas, in the Static approach, an analytical method is used to measure the execution time, but it overestimates the result too [122]. Therefore, evolutionary techniques such as Genetic Algorithms (GA) have shown promising results in pruning the vast search space. In general terms, an optimization problem involves searching for the worst-case input data points from an enormous input space. The search for data points is then refined based on specific fitness evaluation criteria [78].

Multiple fitness evaluations are involved in the GA evolution process, which usually requires several executions of applications on the target hardware or its simulator. The availability of target hardware for early system design phases can be challenging [7, 22, 79]. The execution of a given software or application on the simulator can also be time-consuming. Executions are typically cumbersome and require significant resources to complete. Therefore, there is a need to create such models that can minimize the time and resource-intensive nature of the execution. This motivated the researchers to ponder this issue. So, our prediction model eliminates the need for hardware or a simulator for fitness evaluation. As a result, the temporal verification time can be reduced drastically.

Different machine learning models can be used in the GA evolution process [5]. In this chapter, we present a prediction model that can be used to identify the execution time of an application. The model should be selected according to the complexity and type of the application problem. Some typical examples of machine learning models used in this chapter are Linear Regression [47], Polynomial Regression (2nd Degree) [107], Support Vector Regression (Linear kernel and RBF kernel) [129], Tree Regression [150], Forest Regression [135], K-Nearest Neighbors Regression [77] and Ridge Regression [101]. The initial results for ANN as a prediction model in a GA-based test data generation methodology are presented in [127]. The prediction models have been used in real-time and embedded systems for many years, their use in timing analysis is still unexplored. Therefore, a detailed study of these models is needed to verify their correctness.

In this chapter, a thorough analysis of many prediction models for temporal real-time system verification is conducted employing specific performance metrics. The performance measures defined are prediction performance, isolated timing performance, evolution performance, integrated timing performance, and test data quality. It is necessary to train each of the previously described target prediction models using a cycle-accurate hardware simulator. A sufficient amount of input-output data is needed for a model's training, and they may be produced by running the simulator with a randomly chosen set of input data. The prediction model may be utilized to calculate the fitness function during the GA-evolution process once trained using a simulator. Predicting the fitness function during the GA-evolution process eliminates the need for actual execution of the application program, either on a simulator or hardware.

The proposed methodology is evaluated by using different case studies. The results of these experiments show that the proposed method can generate and predict worst-case data nearer to the WCET. The use of a prediction model also provided a comprehensive speedup. In this chapter, we have used the Pygad framework [43], an open-source Python library, for building the genetic algorithm and optimizing machine learning algorithms. It allows different types of problems to be optimized using the genetic algorithm by customizing the fitness function. To our best knowledge, nobody has proposed the idea of determining the Worst-case data using different machine learning models in the place of the simulator or the hardware to reduce the resource consumption during fitness evaluation.

Contributions: The contributions of the proposed approach are:

- We propose a novel approach to determine Worst-case data using ML based timing model.
- Our approach eliminates the time-complexity reduction of state-of-the-art GA-based temporal testing techniques using ML model, hence saving the execution time during GA-evolution.
- A selection of best prediction model based on R^2 learning score obtained after experiments.
- We evaluate our approach with several different benchmarks using various metrics such as prediction performance, isolated timing performance, evolution performance, integrated timing performance, and test data quality.

5.2 Related Work

This section overviews the state-of-the-art methods and techniques used in temporal verification of the Real-Time Systems. In this, we describe the work done on the analysis of WCET and summarizes the use of evolutionary techniques.

Buzdalov et al. [29] presented an approach for the generation of test data for the Knapsack Problem. The approach is based on the genetic algorithm. It is evaluated on five different algorithms, including one simple branch-and-bound algorithm and two algorithms by David Pisinger and their implementations. The experimental results showed that the genetic algorithm produces better test data than random test generations.

Many researchers have taken avenues to use GA to find the worst-case data for Real-Time and Embedded systems. Wegener et al. [142] proposed a method to generate worst-case data using the GA. The goal is to introduce an improved classification-tree method for functional testing embedded systems. This method should be used in combination with other tests to ensure that the embedded system is thoroughly tested. A new approach was developed to complement the classification-tree method, which focuses on the temporal system behavior. It involves using genetic algorithms to find cases where the execution times are long, and the timing constraints are tight.

Pohlheim et al. [115] proposed a new approach for testing temporal behavior. It aims to study the effectiveness of evolutionary algorithms for testing the temporal behavior of embedded systems. It is based on the idea that the algorithms can be used to establish the minimum and maximum execution time. In order to determine if input scenarios result in a temporal mistake, the tester's job is to identify which ones have the longest or shortest execution timings. Finding such inputs by human temporal behavior analysis and testing complicated systems is nearly difficult. Nonetheless, the inputs with the longest execution durations can be found via evolutionary computation if searching for such inputs is considered an optimization issue.

A Measurement-Based Timing Analysis (MBTA) to establish the WCET is only useful if we can reasonably be assured that we have input the worst-case execution trace into the study. Because of this, the quality of these traces is crucial for certification. The goal of this study [89] is to explore the use of search techniques to produce test cases automatically and consistently, hence providing the necessary execution traces to support MBTA. The work done in this study aims to generate "good data" by creating many fitness functions using a typical search technique. Next, a commercial measurement-based WCET analysis tool is used to enter the data. This paper ultimately presents a new fitness function to support a standard search algorithm; the algorithm focuses on achieving full branch coverage and maximizing loop counts.

One of the most essential steps in the software development process is software testing. For the past ten years, search-based methodologies have been widely applied in the software testing domain. Researchers have focused mostly on metaheuristic search approaches, such as genetic algorithms, out of all search-based strategies. Given the substantial amount of research that has been done and is being conducted in this area, we believe it is past due for an investigation of the performance of genetic algorithm-based testing methods. Based on a literature analysis [133], we provide a road map for the future of software testing using genetic algorithms. Our evaluation primarily focused on efforts that generate software test data using genetic algorithms. The purpose of this impartial assessment is to draw future researchers' attention to the shortcomings of testing based on genetic algorithms, as well as potential fixes and the degree to which they may be made. The insights from the chosen main studies draw attention to the problems that arise when software testing uses evolutionary algorithms. The review's observations show that the type of genetic algorithm employed, fitness function design, population initiation, and parameter settings all have an effect on the caliber of solutions found when employing genetic algorithms for software testing.

The fitness function is determined using an experiment or a computational simulation in the abovementioned studies. Nevertheless, fitness evaluations could become non-trivial when the computer simulation required for every fitness assessment is too time-consuming, or the experiments required to determine fitness functions are excessively expensive. In order to prune the state space of measurement-based analysis and anticipate a set of test data where the execution time is close to the worst case, it offers the impetus for prediction-assisted (machine learning models) evolutionary computation.

The aforementioned limitations of current GA-based temporal testing methods create the framework for prediction model integration. In a GA-based solution, predicting the worst-case input data is the goal of model integration. By using a prediction model, fewer simulation runs are needed, which cuts down on the total amount of time needed for testing. By approximating the fitness function or functions for at least two decades, prediction models are employed in the literature to make the optimization of costly computer simulations easier [54]. The use of prediction models to assist GAs in various industries to solve diverse nature of problems is getting popular. For instance, Koopialipoor et al. [74] used a hybrid neuro-genetic (GA-ANN) prediction model to simulate over breaking caused by drilling and blasting operations in tunnel construction. Moayedi et al. [102] anticipate the eventual bearing capacity of shallow footing on soil using a variety of evolutionary and neural network models, including a genetic algorithm optimized with ANN (GA-ANN). Air blast prediction is carried out by Armaghani et al. [68] using a hybrid genetic algorithm and ANN model. Similar to this, Rodriguez-Roman's [123] work improves travel times and highway safety by utilizing genetic algorithms aided by surrogate models in project designs. Promising outcomes have been observed when prediction models are used to support GAs in a variety of contexts.

In this chapter, we present an integrated approach of genetic algorithms and machine learning models to determine the worst-case data. The fitness function can be computed using a computational simulation or a simple experiment. Generally, fitness evaluations are non-trivial when the simulation is time-consuming, and the experiments are expensive. It motivates the researchers to use machine learning models with GA to determine the worst-case data from a large search space.

5.3 Methodology

The proposed method is shown in Figure 5.1. It intertwines a Genetic Algorithm (GA) and a machine learning model to optimize an application's performance by targeting the identification of the optimal solution, particularly in scenarios involving worst-case data. The GA, a heuristic algorithm inspired by natural selection, commences with the initialization of random data and fitness values, establishing an initial population of potential solutions. Crucially, this process incorporates a fitness function that evaluates the end-to-end execution time of the application, aiming to minimize this metric. Through iterative generations, the GA employs selection, crossover, and mutation mechanisms to evolve and refine these solutions, constantly aiming for improvements in execution time.

Parallelly, a machine learning model is trained using initial training data, enabling the model to discern intricate patterns and relationships within the data. Once trained, this model operates in conjunction with the GA. As the GA generates potential solutions, the machine learning model comes into play by predicting fitness values (execution times) for these solutions without the need to execute the application. These predicted fitness values serve as a guide for the GA, aiding in selecting and prioritizing solutions likely to yield better performance.

The synergy between the GA's evolutionary exploration and the predictive capabilities of the machine learning model results in an optimized approach for handling worst-case scenarios. While the GA navigates the solution space, continuously evolving and selecting solutions based on their evaluated fitness, the machine learning model contributes by estimating fitness values, streamlining the GA's search for the most efficient solution regarding application execution time. This combined approach harnesses the strengths of evolutionary search and predictive modeling, aiming to enhance the application's performance by identifying and refining solutions tailored to worst-case data scenarios.

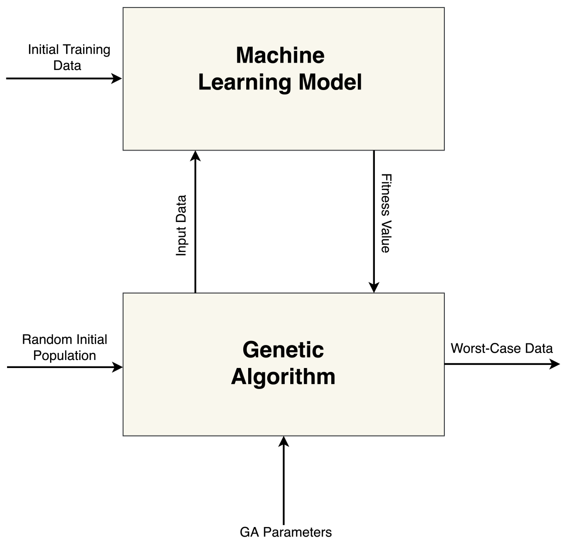


Figure 5.1: Methodology

5.3.1 Genetic Algorithm

Various optimization techniques designed for continuous or differentiable functions often struggle when faced with complex scenarios involving multi-modal and noisy functions. To overcome these challenges, research focuses on devising more robust methodologies capable of handling such intricate problems. In this pursuit, biological and physical principles have emerged as valuable paradigms for optimization. Notably, Genetic Algorithms (GAs) draw inspiration from Darwin's theory of natural selection, emphasizing the survival of the fittest in search procedures.

The fundamental structure of a GA, illustrated in Figure 5.2, embodies a sequential process aimed at evolving a population towards an optimal solution. It creates a random set of individuals, forming the initial population. Subsequently, each individual's fitness—its adequacy or suitability concerning the problem—is assessed. The algorithm proceeds to the next phase if the termination criteria are unmet.

Central to the GA's functionality is selecting parents from the initial population. Parents' choice heavily relies on their fitness values, favoring those deemed more fit for reproduction. Once the parents are identified, genetic operators such as crossover and mutation come into play, facilitating the creation of new offspring that inherit characteristics from their parents. These newly formed offspring then undergo an evaluation of their fitness. The evolutionary cycle continues across generations, perpetuating the process of selection, reproduction, and evaluation. Each successive iteration refines the population, fostering the emergence of po-

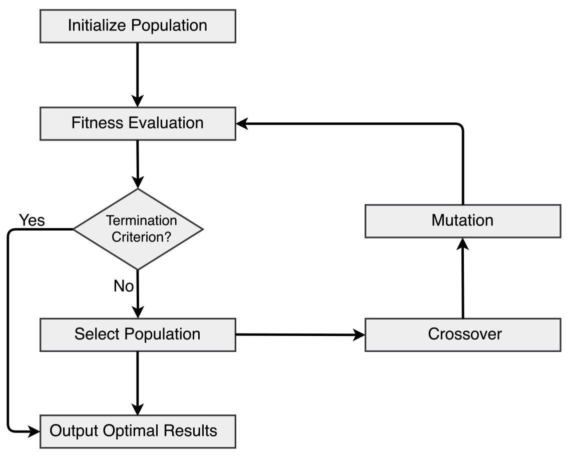


Figure 5.2: Structure of Genetic Algorithm

tentially superior solutions. This cyclical evolution persists until the predetermined stopping criteria are met, signaling the end of the algorithm.

By mimicking evolutionary principles, GAs traverse the solution space, iteratively improving the population's fitness and diversity. This iterative optimization, driven by selection pressure and genetic variation, enables GAs to explore and exploit the solution landscape efficiently. Ultimately, GAs offer a versatile approach to tackling complex optimization problems, particularly those characterized by multi-modalities and noisy functions, by iteratively evolving towards better solutions in a manner inspired by natural selection.

5.3.2 Machine Learning Model

The machine learning model is used to determine the fitness value of an individual solution in any generation of GA. It is first trained and then used with the GA, as shown in Figure 5.3. Consequently, the use of a machine learning model eliminates the need for a simulator or actual hardware for the fitness evaluation. In this work, we have targeted different machine learning models such as Linear Regression, Polynomial Regression (2nd Degree), Support Vector Regression (Linear kernel and RBF kernel), Tree Regression, Forest Regression, K-Nearest Neighbors Regression and Ridge Regression. The process begins with the random input provided to both the gem5 Simulator and the machine learning model to train our model. This process is repeated until the error is bounded according to certain stopping criteria. The machine learning model is considered to be trained once the stopping criteria are satisfied. This model is then used to predict the execution time for the given inputs and is integrated into the proposed test data generation methodology.

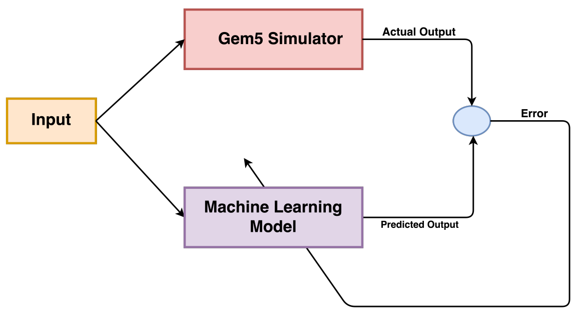


Figure 5.3: Training of Model

5.4 Experimental Setup

This section elaborates the experimental setup for the evaluation of prediction models. It first describes the particular settings for target architecture and framework in subsection 5.4.1. The benchmark used for evaluation is described in subsection 5.4.2. The parameters for GA-evolution and model training are presented in subsection 5.4.3 and 5.4.4, respectively.

5.4.1 Target Architecture and Framework

The ARM v7-A instruction set architecture-based micro-architectural model is used as the target platform, available in the Gem5 simulator. The selected architecture model is a single-core processor clocked at a frequency of 2 GHz with 512 MB of physical memory. Furthermore, the memory model used is simple classic memory, and the simulator mode used is called emulation mode. All this experimentation, including the running of the simulator, was performed on a Macbook Air with a 1.8 GHz Dual-Core Intel Core i5 processor and 8 GB of physical memory. The prediction models employed in this work are realized in the Pygad framework. It is a versatile and accessible Python library designed to efficiently implement genetic algorithms (GAs) for optimization and machine learning tasks. This framework simplifies the integration of GAs into Python-based projects, offering a range of functionalities to facilitate optimization processes. It provides a straightforward and flexible API, allowing users to easily create and customize GA-based solutions. With its modular structure, PyGAD supports customizing genetic operators such as crossover and mutation, enabling tailored solutions for various problem domains.

5.4.2 Benchmarks

We chose two benchmarks from the Mälardarlen WCET suite to examine various prediction models. Insertion sort and Bubble sort are these two benchmarks. These benchmarks were chosen because, as a result of the input data, they can most accurately depict the variation in their execution times. Fixed-sized unsorted arrays of random numbers between 0 and 1000 are the inputs used in this study's benchmarks (sorting techniques). These benchmarks' algorithms sort the arrays. The method determines how many swaps are needed to sort the list. On the other hand, the beginning order of the list to be sorted also determines the number of swaps needed. A benchmark's execution time, a function of the number of swaps, highly depends on the input data. $O(n^2)$ swaps, where n is the length of the supplied list of numbers, represent the worst-case performance for these sorting methods. A reverse-sorted array of integers serves as the benchmarks' worst-case input.

5.4.3 GA-evolution parameters

The Genetic Algorithm (GA) performance heavily relies on consistent and well-defined parameters governing its operations. In this context, the parameters utilized in the GA framework are meticulously chosen and maintained constant across experiments. The GA begins with unsorted, fixed-sized arrays of randomly generated integers within a specified range, forming the initial population of potential solutions. Throughout the GA iterations, the population remains steady at 50 individuals per generation, ensuring a stable solution for space exploration. Table 5.1 outlines the fixed GA parameters employed consistently in the experiments. Notably, the convergence of a GA toward the desired outcome is profoundly influenced by the probabilities governing crossover and mutation operations. Crossover, responsible for generating new offspring from parent solutions, is regulated by a probability typically between 0.6 and 1.0. A higher crossover probability encourages exploration and diversifying solutions, while a lower value may lead to premature convergence, limiting the algorithm's effectiveness. Similar to adding variation or exploration, mutation operates with a relatively smaller probability than crossover. Its role is to introduce novelty and explore uncharted areas in the solution space. Mutation probability generally ranges between 0.005 and 0.05 to prevent excessive randomness that could derail the GA from a structured search. Determining optimal GA parameters follows a methodology inspired by Pongcharoen et al. [116]. This approach evaluates parameters based on balancing minimum total cost and minimum spread. It systematically explores various combinations of parameters like population size, number of generations, crossover, and mutation probabilities. The aim is to pinpoint parameter values that balance exploration and exploitation, ensuring an effective search while preventing premature convergence or excessive randomness. This systematic evaluation enables the selection of GA parameters that foster efficient exploration of the solution space, aligning with the objectives of the optimization problem at hand.

Table 5.1: GA parameters used in different Generation Evolution

GA Parameters	**Methods/Values**
Population size	200
Parent selection	Roulette wheel selection
Crossover	Arithmetic crossover
Crossover Probability	0.6
Mutation	Single point Random Mutation
Mutation Probability	0.05
Population selection	Elitism, 2 best individuals
Maximum number of generations	1000

5.4.4 Parameters for model training

The R^2 score, also known as the coefficient of determination, is a statistical measure used to assess the goodness of fit of a regression model to the observed data. It quantifies the proportion of the variance in the dependent variable (the variable being predicted) that is explained by the independent variables (the predictors) in the regression model. R^2 score ranges between 0 and 1, where:

- An R^2 score of 1 indicates that the regression model perfectly predicts the dependent variable based on the independent variables.
- An R^2 score of 0 suggests that the model fails to explain any variability in the dependent variable beyond the mean of the dependent variable.
- Values between 0 and 1 denote the proportion of the variance in the dependent variable that the model explains.

Mathematically, the R^2 score is calculated as the ratio of the explained variance to the total variance of the dependent variable. It is computed as one minus the ratio of the sum of squared errors of the model to the total sum of squares.

R^2 score is a useful metric in evaluating the performance of regression models; however, it does have limitations, especially when dealing with complex data or overfitting. Therefore, it is often recommended to complement the R^2 score with other evaluation metrics to gain a comprehensive understanding of a model's predictive performance

5.5 Evaluation

This section presents the experimental results with the target prediction models and the selected benchmarks, such as Bubble and Insertion Sort on various performance measures, described in subsection 5.5.1 and 5.5.2, respectively.

5.5.1 Bubble Sort Experiments

The provided scatter plot in Figure 5.4 showcases the correlation between the predicted end-to-end execution times by various machine learning models and the actual execution times obtained from the gem5 simulator. Each point on the plot represents a data point from the 3000 randomly generated lists, where the x-axis denotes the true execution time obtained from gem5, and the y-axis depicts the predicted execution time by the trained machine learning models. The closeness of points to the diagonal line signifies the accuracy of the predictions. Ideally, points lying close to this line indicate that the predicted values align closely with the actual execution times, illustrating a strong predictive capability of the machine learning models. The R^2 values associated with different models, displayed within the figure, quantify the goodness of fit of these models to the data. A higher R^2 score (closer to 1) indicates a better fit, signifying that the models explain a larger proportion of the variance in the execution times. Overall, the scatter plot and R^2 values indicate the accuracy and performance of the machine learning models before their integration into the Genetic Algorithm (GA) evolution process, validating their predictive capabilities for assessing the fitness of individuals within the GA population.

In Table 5.2, the percentage deviation of predicted worst data generated by each machine learning model is highlighted. This deviation comparison reflects the difference between the predicted execution times generated by these models and the actual execution times obtained from the gem5 simulator for the test data—referred to as the final population in the GA evolution. Notably, this test data holds a distinct characteristic: its execution times closely align with the maximum possible execution time. The computation of percentage deviation provides insight into how accurately these machine learning models predict the extreme or worst-case scenarios represented by this test data. A smaller percentage deviation signifies a closer alignment between the predicted execution times by the models and the actual execution times measured by the gem5 simulator for this critical test dataset. This analysis offers an evaluation of the model's performance under extreme conditions, shedding light on their ability to predict execution times accurately in scenarios where the application faces maximum workload, thereby assessing the robustness and reliability of these models in handling worst-case scenarios.

Table 5.3 shows the comparison between the execution time measured during the simulation and through the prediction. The first column represents the benchmark name. The second column represents the simulation time for the execution of single input of this benchmark. The third column presents the time taken by the best machine learning model to predict execution

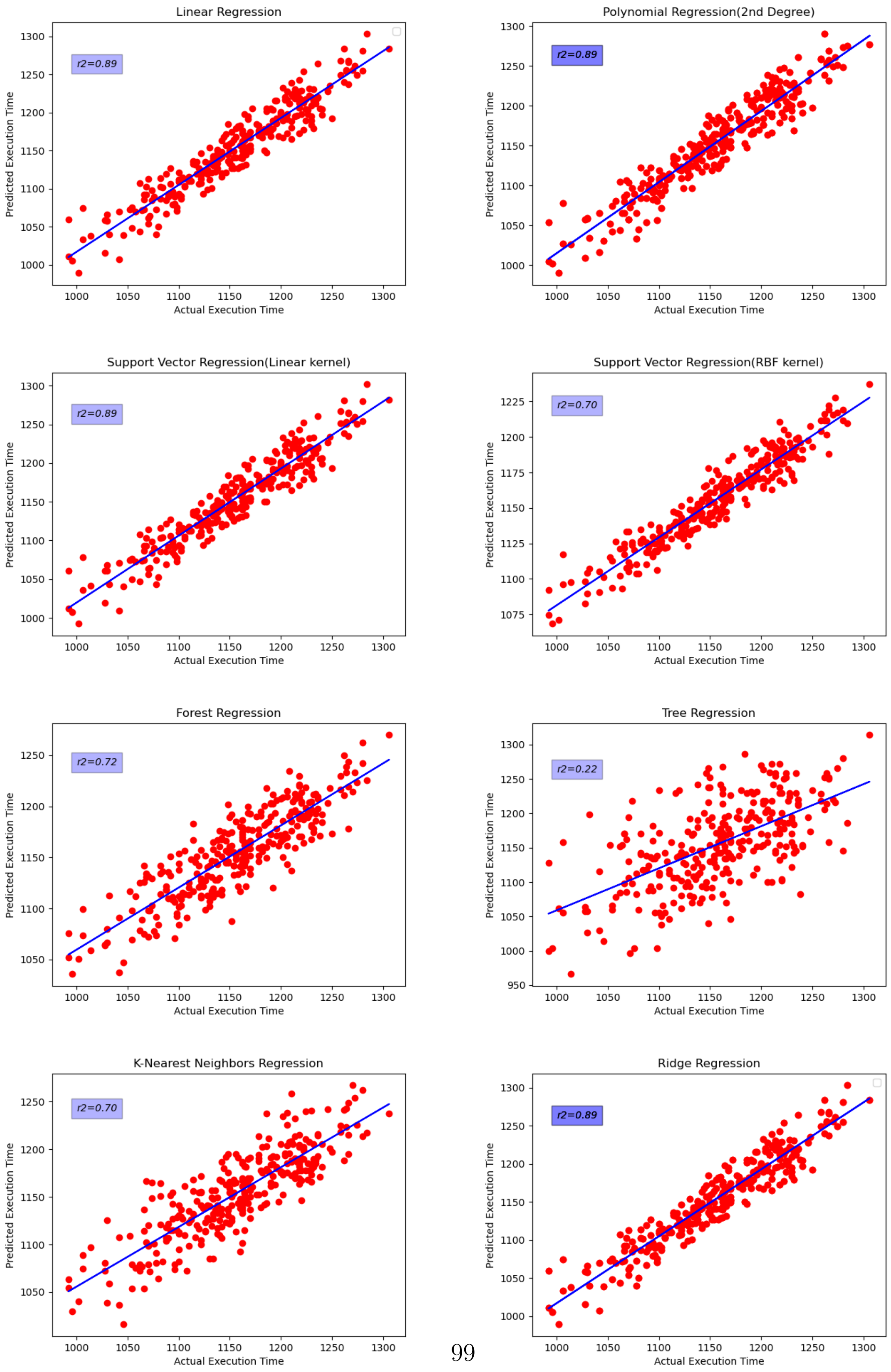


Figure 5.4: A scatter plot of measured vs predicted execution times for bubble sort using different prediction models

Table 5.2: Percentage deviation between the execution time of each data point and the maximum possible execution time for bubble sort

Models	Percentage Deviation from maximum execution time
Linear Regression	7.26
Polynomial Regression (2nd Degree)	8.19
Support Vector Regression (Linear kernel)	7.39
Support Vector Regression (RBF kernel)	11.22
Tree Regression	13.73
Forest Regression	12.28
K-Nearest Neighbors Regression	13.07
Ridge Regression	7.26

Table 5.3: Speedup ratio for bubble sort

Benchmark	Simulation Time(millisec)	Prediction Time(millisec)	Speedup
Bubble Sort	138	5.60	24.64

time values for this benchmark. Finally, the last column represents the speedup gain.

5.5.2 Insertion Sort Experiments

The input data is generated randomly. It consists of lists of 16 elements. Each data list is considered as a separate input. The type of elements in each input data list is an integer and varies between 0 to 1000. The benchmark is executed using the gem5 simulator, and the execution time is obtained against each of the 3000 input lists.

The scatter plot in Figure 5.5 shows that the execution time is accurate. Almost all the data points are near the straight line. In each figure, the x-axis represents the actual execution time obtained from the simulator against a given input, whereas the y-axis shows the execution time predicted by the model against the same input. The R^2 values represent how well the model fits the input data. The higher the R^2 values, the better the model learns about the data. After training the machine learning model, it is ready to be used to predict the execution times in the genetic algorithm evolution.

The test data collected during the evolution of GA is now considered to be the final population. The salient feature of this test data is that its execution time is near to the maximum possible execution time. The execution time of the test data is computed by using the gem5

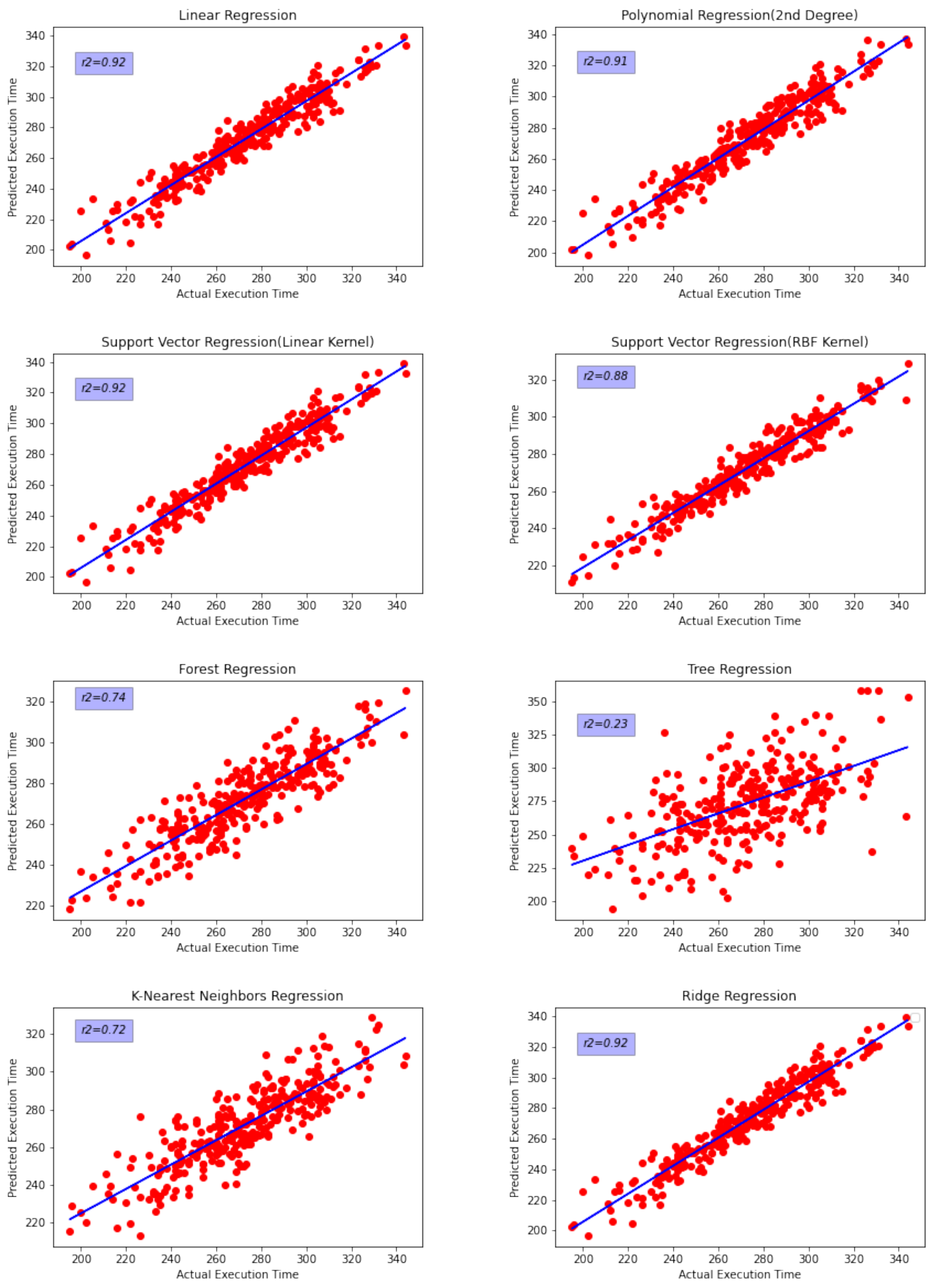


Figure 5.5: A scatter plot of measured vs predicted execution times for insertion sort using different prediction models

Table 5.4: Percentage deviation between the execution time of each data point and the maximum possible execution time for insertion sort

Models	Percentage Deviation from maximum execution time
Linear Regression	14.12
Polynomial Regression (2nd Degree)	13.24
Support Vector Regression (Linear kernel)	14.34
Support Vector Regression (RBF kernel)	19.42
Tree Regression	25.38
Forest Regression	23.39
K-Nearest Neighbors Regression	26.04
Ridge Regression	14.12

Table 5.5: Speedup ratio for insertion sort

Benchmark	Simulation Time(millisec)	Prediction Time(millisec)	Speedup
Insertion Sort	140	5.12	27.34

simulator. This measures the percentage deviation of predicted worst data generated by each model, as shown in Table 5.4.

Table 5.5 shows the comparison between the execution time measured during the simulation and through the prediction. The first column represents the benchmark name. The second column represents the simulation time for the execution of single input of this benchmark. The third column presents the time taken by the best machine learning model to predict execution time values for this benchmark. Finally, the last column represents the speedup gain.

5.6 Summary

In this chapter, we introduce a methodology employing Genetic Algorithms (GAs) in conjunction with diverse machine learning models to identify worst-case data scenarios in applications. By utilizing various machine learning models facilitated by the Pygad Framework, substantial time reductions were achieved. The models underwent training utilizing the gem5 simulator, after which they were integrated into the GA evolution process to predict application execution times. The outcomes exhibited promising enhancements in speed, demonstrating the efficacy of this integrated approach. The ability to generate test data closely resembling worst-case scenarios significantly expedites temporal verification processes for real-time and embedded systems. The proposed methodology underwent comprehensive scrutiny, particularly focusing on frequently employed sorting algorithms, showcasing its versatility and potential applicability across various computational scenarios. This amalgamation of GA, machine learning, and simulation techniques offers a promising pathway towards expedited temporal analysis and optimization in critical system verification and benchmarking processes. The suggested method is universal even if the testing in this work is restricted to the sorting tasks. Examining the effects of various hardware platforms and prediction models on the effectiveness of the suggested technique may be a further avenue for future study.

Chapter 6

Early WCET Estimation using ML and DNN

The Worst-Case Execution Time (WCET) is crucial and an essential factor in analyzing and developing Real-Time Embedded Systems. The idea of the WCET allows the designer to create safe and reliable real-time systems. The WCET is used by the scheduler to determine an appropriate scheduling scheme for the application and to guarantee timing constraints. These systems need to satisfy a strict deadline, and failing leads to catastrophic events such as loss of life. Generally, WCET analysis is applied only in the late stages of Safety-Critical Systems development when the hardware is available, and code is compiled and linked. Different methods exist to determine WCET, but none of these provide early insight into WCET. Suppose early WCET is unavailable during system development. In that case, many systems design decisions are made using experience, which might be impractical if systems don't satisfy timing constraints, and it may result in costly system re-design. However, we need early WCET in the initial stages of systems development as an essential prerequisite to configure the system properly. We propose a method to determine early WCET using Machine Learning and Deep Neural Networks models without the need for the hardware and binaries, i.e., source-level timing analysis—this new approach estimate WCET using source code. The models can predict WCET without running the code on the platform once it is trained, which is created using the Pytorch framework. The viability of the proposed method is demonstrated with the TACLeBench benchmark suite. The results show reasonable estimates on the predicted execution time of the final and compiled code, proving the prospect of the methodology.

In the rest of this chapter, the introduction and motivational examples are presented in Section 6.1. Section 6.2 describes the related work in estimating early WCET with and without ML approaches. The early WCET estimation using ML is presented in section

6.3. Experimental setup and ML results are described in Sections 6.4 and 6.5, respectively. The early WCET estimation using DNNs is presented in Section 6.6. Section 6.7 describes the DNN approach in detail, while our experimental setup and results are described in Section 6.8. and 6.9, respectively. Finally, we conclude the chapter in Section 6.10.

6.1 Introduction

In safety-critical systems, the timing domain is as important as the value domain. These systems need to satisfy the timing constraint; otherwise, resource damage or even life loss could occur. For instance, it is essential to know that airbags in cars open fast enough to save lives. Besides, these systems not only satisfy the correctness of the system but also be responsive. If the system does not satisfy the timing constraints after manufacture, then changing the hardware that cannot schedule tasks would be more expensive to redesign. Therefore, estimating the worst-case execution time is very crucial. Estimating WCET [145] for the given architecture is difficult, if not impossible, to cover all the system states, and it requires the user's input. Modern processors are equipped with complex architectural features such as superscalar pipelines and caches that make WCET estimation complex. For instance, caches introduce the variance in operations execution time based on the hit or miss in the caches. In the previous decade, many optimizations have been done to improve the average-case execution time, but less work has been done to estimate WCET precisely and accurately. The process of estimating the WCET is called timing analysis. The timing analysis of the given system is possible in the last stage of the system development process. Hardware platform and compiled binary code are required to estimate the WCET. By getting the early estimate of WCET, we will prune the system's unwanted design points based on the parameter of interest, i.e., design system exploration.

In the last decade, Real-Time Embedded Systems have covered all aspects of our lives. We are surrounded by it, and the possible applications with these systems are endless such as avionics, automotive, nuclear power plant, robotics, e-health, etc. Unlike general-purpose computer systems, critical systems and cyber-physical systems need to satisfy timing constraints and the functionalities correctness [145]. For instance, the airbag in the car must open on time if any unwanted condition happens; otherwise, it would lead to the loss of life or significant damages. The other different application is a nuclear power plant in which gas emission should be reported to the station as soon as possible; otherwise, a considerable loss may happen. Needless to say that all these systems are ubiquitous. It is crucial to choose the correct hardware configuration such as caches, pipelines, branch prediction, etc., in the high-volume market of Real-Time Embedded Systems to reduce the development costs. It will reduce the risk of the final systems not meeting the timing constraints.If the timing constraints of the systems don't meet, then costly re-design of the system is required.

Traditionally timing estimation is done in the late stage of the systems development when the hardware is available, and the compiled code is linked. All the WCET available tools such as aiT [39], Heptane [57], and OTAWA [13] can estimate WCET with the binary codes for the selected hardware configuration. However, none of these tools give an early estimation of WCET. Therefore, a method to find early WCET would be very useful in developing the systems as it will reduce the design space exploration.

In this chapter, we propose a method for source-level timing analysis for the given hardware configuration and compiler. We show how the machine learning model is used to estimate WCET. Our approach doesn't need binary to be analyzed. The method has the following properties:

- It is beneficial for both soft and hard real-time systems for the early development phase.
- It is based on the set of virtual instructions which define the abstract of the selected hardware and compiler.
- The timing model is linear and consists of fixed costs for the virtual instructions.

Various prediction models can be used to estimate WCET. In this chapter, we present a framework in which an appropriate prediction model can be used to predict the execution time of a problem at hand. The selection of a proper model depends on the complexity and type of the application problem. The initial results for ANN as a prediction model in an early WCET estimation methodology are presented in [80]. However, a detailed study of prediction models to estimate WCET for temporal verification of real-time systems has not been performed yet. Moreover, the use of prediction models in timing analysis is a novel idea and thus requires a detailed evaluation of prediction models.

This article performs a detailed study of various prediction models for early WCET estimation of real-time systems using machine learning approaches. All the above-stated target prediction models are required to be trained through a cycle-accurate simulator of the hardware. The training of a model requires some adequate input-output data, which are obtained by running the simulator with the randomly selected set of input data. Once the prediction model is trained through a simulator, it can be used for the estimation of the unseen application.

The empirical nature of timing models underestimates execution times, so our timing estimation is not always safe. A safe timing estimate is always required in the final verification phase. However, this value in the early design of the system is very useful in dimensioning the correct Real-Time Systems.

6.2 Related Work

Bonenfant et al. [22] presented an approach for early WCET prediction using machine learning based on C source code. Their method used a Static approach which generated worst-case event counts such as the number of arithmetic operations like addition, subtraction, multiplication, and division, the number of function calls, the number of global variables, and the number of reads and writes access. To train the model, they used these features with labeled WCET. The worst-case events count of source code was formulated to obtain a satisfiable prediction of the future WCET. As far as considering estimating early WCET, this approach works well. However, it has some shortcomings in that event-counting of code using CFG results in the loss of valuable code flow information.

Thomas Huybrechts et al. [66] proposed a new extension to the hybrid approach to predict early WCET using machine learning. This new approach estimates the WCET on smaller entities of the code, so-called hybrid blocks, based on software and hardware features. As a result, the ML-based hybrid analysis provides insight into the WCET early on in the development process and refines its estimate when more detailed features are available. A new tool named COBRA was proposed to extract the features. The extracted features were used to train and validate the model. Machine learning approaches, such as Linear regression, Tree regression, and Support vector machines, were used to compare the results of TACLeBench [38] applications. The mean relative error for support vector regression with Linear kernel was 40.2%, which was too high to use as an upper bound on WCET.

Oyamada et al. [108] presented a neural network-based approach for accurate software performance estimation, which also deals with the non-linear behavior of execution times due to complex modern architecture such as deep pipeline, branch prediction, and cache sizes. Assembly instructions were used as features categorized as floating-point, integers, branches, and load/store operations. Based on the Control Flow Graph (CFG), the trained data is classified into two parts as control dominated applications and data dominated applications. Feed-Forward neural networks have been used with one input layer, one hidden layer, and one output layer with different neurons at each layer. CFG weights were used to make the distinction of application domains. The generic estimator had a maximum overestimation of 41.01% and a maximum underestimation of 20.69%. However, for the specialized estimator, they improved the overestimation and underestimation slightly. The error was too high for the estimate to be used as upper bound but obtaining such results in the development process is useful for system design.

The approach proposed in [67] extended the work done in [66] with a deep neural network to estimate WCET. This work used two different models: a feed-forward neural network and a tree recursive neural network. The data they used in their experiments were taken from TACLeBench benchmark suits. The architecture they used for dataset A was one input layer of ten neurons, three hidden layers, 32 neurons, and one output layer. The results were given in terms of Root Mean Square Error (RMSE), and for dataset A, it was around 40% on validation. The results are too large to obtain any useful upper bound.

Lisper and Santos [96] developed a new Measurement-based WCET analysis method, which uses regression to identify parameters in the common linear Implicit Path Enumeration Technique (IPET) model to calculate WCET. The method can use different granularity timing measurements, including end-to-end measurements, which reduces the need for fine-grained timing measurement instrumentation. Abel and Reineke [3] developed an algorithm to model the cache's replacement policy by measuring actual hardware automatically. This work helps identify the cache-sensitive timing model.

In this chapter, we present a ML and DL based approach to predict early WCET with a network architecture different from the aforementioned approaches. Our models are evaluated on TACLeBench [38] benchmark suites. We use the SWEET [95] tool to extract the features. The primary function of SWEET is to perform *flow analysis* to identify *flow facts* i.e., information about loop bounds and infeasible paths in the program and explained in detail in chapter 2. Flow facts are necessary for finding safe and tight WCET. Any WCET analysis must satisfy safeness and tightness conditions, which reflects the estimate of WCET precisely.

6.3 Early WCET Estimation Using ML

Our method operates in two phases. In the first phase, described in section 6.3.1, different prediction models are trained using the selected training programs. In the second phase, presented in section 6.3.2, the WCET of different benchmarks is estimated using our methods. These two phases are presented in a target-independent manner.

6.3.1 Training Phase

The training phase within the depicted process (as shown in Figure 6.1) involves crucial steps to understand and learn the timing behavior of the processor, targeting a specific hardware/compiler configuration. Initially, training programs are meticulously designed and chosen, ensuring compatibility with the target hardware or a simulator environment. These programs are then executed on the designated platform, recording the required cycles. This recorded cycle count serves as valuable timing information that reflects the actual execution behavior on the hardware or simulator. Simultaneously, the same training programs undergo translation

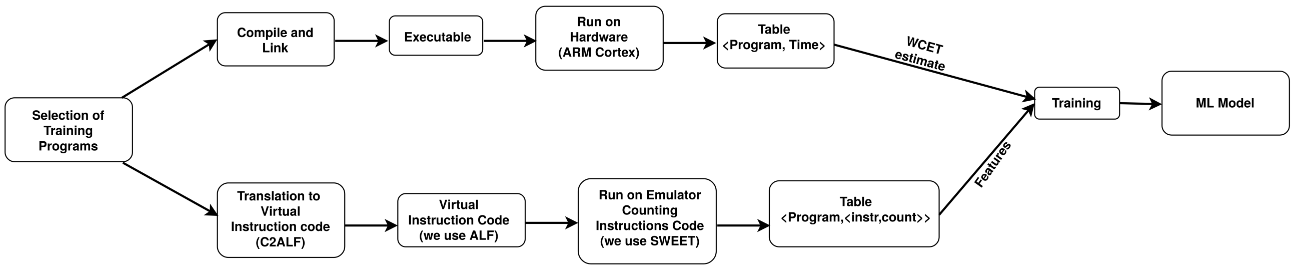

Selection of Training Programs
Compile and Link
Executable
Run on Hardware (ARM Cortex)
Table <Program, Time>
WCET estimate
Training
ML Model
Translation to Virtual Instruction code (C2ALF)
Virtual Instruction Code (we use ALF)
Run on Emulator Counting Instructions Code (we use SWEET)
Table <Program,<instr,count>>
Features

Figure 6.1: ML Model Training Phase

into an intermediate format known as ALF, generating virtual instructions that represent the program's functionality in an abstract manner. These virtual instructions are then analyzed using the SWEET tool, which specializes in conducting flow analysis. SWEET's [95] primary function involves extracting critical flow facts from the programs, such as information about infeasible paths and loop bounds. This flow analysis helps comprehend the intricate program structures, identifying critical program features impacting execution time. The collected data, comprising cycle counts from hardware execution and flow facts derived from virtual instructions via SWEET, serve as input for training the machine learning model. Feeding this data to the model, it learns to discern patterns and relationships between the recorded cycle counts and the extracted flow facts. The model iterates through various algorithms or configurations, optimizing its parameters to identify the best-fit model that accurately predicts execution times based on the learned timing behavior. This holistic approach integrates real-time hardware execution data, abstracted virtual instructions, and critical flow facts to train the machine learning model. The aim is to discern and capture the nuanced timing behaviors of the processor, enabling the selection of the most effective model for predicting WCET based on the interplay between program structures and hardware execution characteristics.

6.3.2 Testing Phase

The WCET estimation for the benchmark programs is shown in Figure 6.2. Initially, specific benchmark programs are meticulously chosen based on predetermined criteria. Subsequently, these selected programs transform virtual instructions utilizing ALF. ALF enables the conversion of the programs into a format compatible with the subsequent prediction model. The prediction model, which has been previously trained or developed using machine learning techniques, plays a pivotal role in this estimation process. Leveraging the features and characteristics extracted from the transformed virtual instructions, the prediction model computes an estimation of the WCET for each program. This estimation leverages patterns, relationships, and learned behaviors from the training data to predict the potential maximum execution time

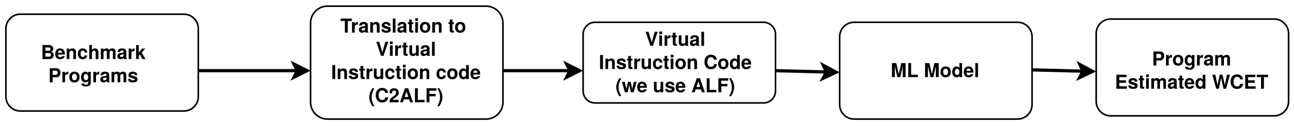

Benchmark Programs
Translation to Virtual Instruction code (C2ALF)
Virtual Instruction Code (we use ALF)
ML Model
Program Estimated WCET

Figure 6.2: WCET Estimation Phase

of these benchmark programs.

6.4 Experimental Setup

In this section, we detailed the experimental setup used to evaluate our approach for the Raspberry Pi 4 [121] platform. The hardware and software environments are first introduced. The TACLebench benchmark suite is presented. We then detail the learning and prediction phase of our approach.

The Raspberry pi B 4 relies on a Broadcom BCM2711 SoC, which is based on a 1.5 GHz 64-bit quad-core ARM Cortex-A72 processor, a 2-wide super scalar processor. The architecture features a private L1 cache and a 1 MB shared L2 cache. The Raspberry Pi runs the Raspbian Lite operating system (Linux kernel version 4.19, a light operating system without a user interface to minimize the impact of the operating-system activity on timing). The timing noise coming from the operating system can be avoided by running the codes as Linux kernel modules. The compiled codes are executed on a specific core (core 3) on which no user task is allowed to run (isolated core, using the Linux isolcpus facility). The quality of our approach was evaluated on benchmarks from the TACLeBench [38] benchmark suite. The benchmark having floating point values are not supported by our approach because execution in kernel mode does not support this. We also discarded benchmarks having recursion and complex call due to the same reason.

A total of 242 programs were used for the training of the prediction models. We have made sure that the construction and selection of the training program cover all the instructions for the ARM Cortex-A72 processors. The training programs are constructed using the extended approach presented in [2], which covers all the context-dependent timing effects due to hardware features such as caches, pipelines, and branch prediction units, and code optimizations due to the compiler. The training programs are executed on the ARM cortex processor to measure the execution time, which is used as labels. The same training programs are translated into virtual instructions using SWEET, and features are extracted. The prediction model quality depends on the input training data, which needs to be biased-free as much as possible. The features extracted using SWEET tools are shown in Table 6.1, such as a number of additions, subtractions and modulo operations, and so on. The features need to be extracted carefully

Table 6.1: Features Extracted

Number of Addition Operations	Number of Subtraction operations	Number of Multiplication Operations
Number of Division Operations	Number of Logic Operations	Number of Shift Operations
Number of Function calls	Number of Return Statements	Number of Jump statements
Number of Load Operations	Number of Store Operations	Number of Comparisons

Table 6.2: Experimented Machine Learning Algorithm

Algorithm	**Description**
Random Forest	A multitude of decision trees
Tree	Tree regression
SVM (Linear Kernel)	Support Vector regression (Linear Kernel
SVM (RBF Kernel)	Support Vector regression (RBF Kernel)
K-Nearest Neighbors	K-Nearest Neighbors regression
Bayesian Ridge	Bayesian ridge regression

as it leads to the bias in the system. We also normalized our data values in the range of 0 to 1 to avoid the prediction model's inclination towards the features having larger values. There are many frameworks available nowadays to use for Machine learning, such as Pytorch and Tensorflow. In our experiences, We have used a Pytorch framework. The actual training and validation are performed through a 4-fold cross-validation process.

WCET estimation is evaluated on the machine learning algorithm provided by the Scikit library [110]. Preparatory experiments made us select the six algorithms that gave the best result among those provided by the Scikit library, as shown in Table 6.2.

6.5 Experimental Results

Table 6.3 reports the Mean Squared Error (MSE) on the benchmark using the six prediction models. The lower the value of MSE, the better the prediction model. The results show that out of all the prediction models, none of them outperforms the other in all benchmarks. However, the best prediction model in this is Support Vector Regression with Linear Kernel followed by RBF Kernel. In Figure 6.3, a scatter plot illustrates the predicted WCET values against the actual execution times of the validation set. Each scattered point corresponds to a specific benchmark utilized in the experiment, and the points are differentiated by color. The proximity of a point to the vertical line indicates the magnitude of prediction error, with closer

Table 6.3: 4-Fold cross-validated Mean-Squared Error of each Regression Model

Regression Model	MSE
Forest Regression	0.03873
Tree Regression	0.03883
Support Vector Regression (Linear Kernel)	0.03767
Support Vector Regression (RBF Kernel)	0.03768
K-Nearest Neighbors Regression	0.03969
Bayesian Regression	0.03969

points indicating lower error rates. Upon inspection, it becomes evident that SVR with both Linear Kernel and RBF Kernel exhibit a noteworthy accuracy in their predictions. The tight clustering of points around the vertical line, particularly in the vicinity of the SVR predictions, underscores the efficacy of these models in accurately estimating WCET values. This graphical representation not only offers a visual depiction of the predictive performance of various regression models but also emphasizes the superior accuracy achieved by SVR with Linear and RBF Kernels, thereby highlighting their potential suitability for WCET prediction tasks. This is because of the simple architecture in our experiments with fewer features such as a single core and no cache. Compiler optimizations were disabled when compiling the program to ease the flow of information during WCET analysis.

6.6 Early WCET Estimation Using DNN

Deep Neural Networks [90] are gaining popularity in every field of science due to their ability to solve complicated applications with increasing accuracy over time. They are a subset of Artificial intelligence that attempt to learn patterns based on input data. It is the machine learning techniques that provide computers with the ability to learn from observed data. Supervised learning and Unsupervised learning are different types of machine learning. In supervised learning, the system is given labelled data, whereas in unsupervised learning, the system is given unlabeled data. Our approach uses supervised learning in which labels are formed out of the number of cycles consumed for each training program.

The analogy of a neural network has been taken from the neuron present in the human brain. The whole concept of deep learning is to try and mimic the human brain and get similar functions as the human brain has and leverage the things that evolution has already developed for us. Millions of neurons are present in the human brain. Neurons send and process signals in the form of electrical and chemical signals. Biological neural networks consist of interconnected neurons with dendrites that receive inputs. Based on these inputs, they produce an output through an axon to another neuron. The neuron (node) is the building block of any deep neural

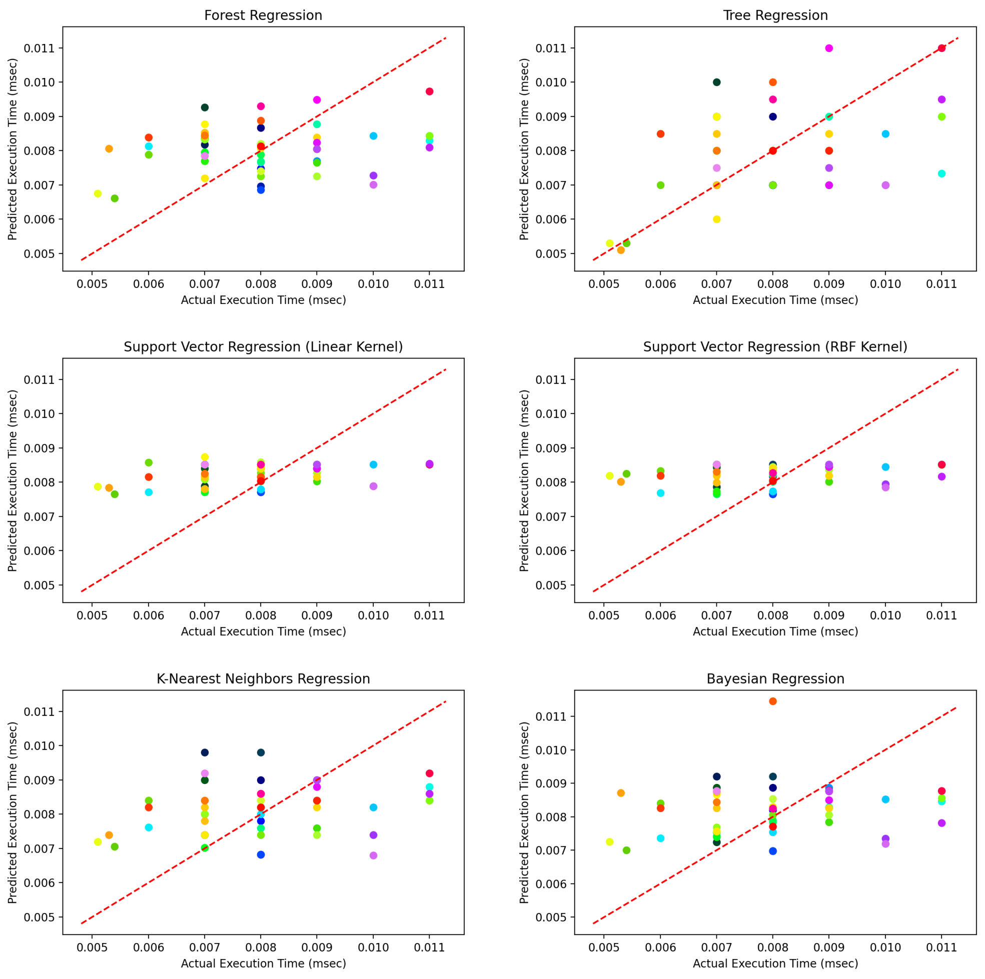


Figure 6.3: A scatter plot of measured vs predicted execution times for different trained prediction models on the validation set.

network. An example of a neuron in fig. 6.4. shows the input (X_1 - X_n), their corresponding weights (W_1 - W_n), a bias (b) and the activation function f applied to the weighted sum of the inputs. The parts/components of a typical deep learning system are described below, and later we apply the following steps to create the model, train the model, and for the accuracy of the model.

- Data - The data is what we apply deep learning techniques on. The data gives insights into how our features and labels are related.
- Task - On the given data, what tasks have to be performed – such as classification and regression.

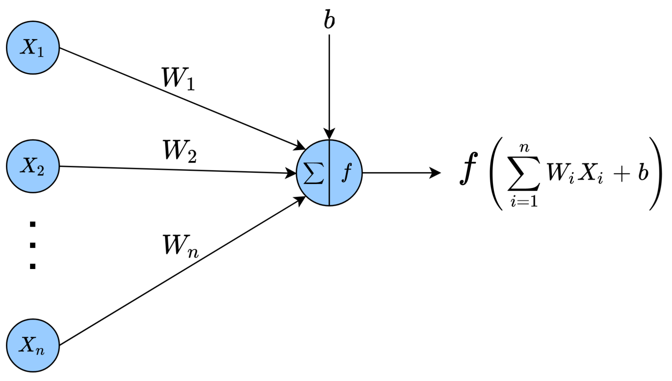


Figure 6.4: A Neuron.

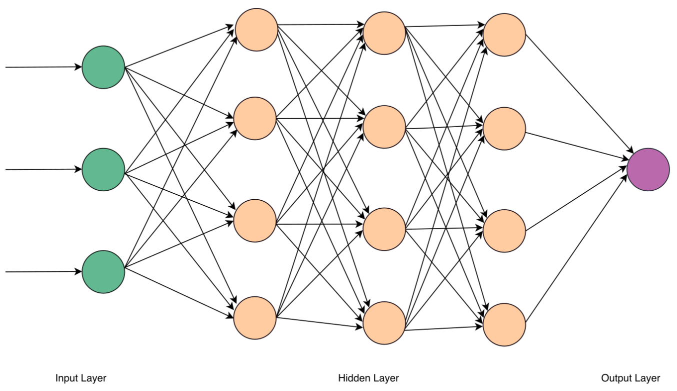


Figure 6.5: A Feed Forward Neural Network.

- Model - Model represents the details of the architecture. Some of the popular models are Feed-Forward Neural Network, Convolution Neural Network, and Recurrent Neural Network.

- Loss Function - The loss function is used to evaluate how well the learning algorithm predicts the outcome. The learning algorithm tries to improve itself by loss functions. There are different types of loss functions such as mean square error and cross-entropy loss.

- Learning Algorithm - The learning algorithm is used to update each parameter in our neural network. Using a learning algorithm, our model learns to identify trends in the data. Some of the different learning algorithms are gradient descent and Adam [73].

- Accuracy - The predicted value is compared with the actual value to find the accuracy which tells us how well our network performs.

A feed-forward neural network or Multi-layer Perceptron (MLP) [136] is an essential deep learning model. A feed-forward network, is as shown in Figure 6.5. consists of one input layer, one and more hidden layers, and one output layer. In the feed-forward network, neurons are not connected to themselves or neurons in the same layer. A fully connected network is a particular case of a feed-forward network where all neurons of one layer are connected to all the following layer's neurons. In the general case, not all the neurons need to be active, and in some networks, most of them are inactive. Neurons at the hidden layer have two portions – linear, and non-linear activation functions. For the given inputs and weights of each layer, the feed-forward network can predict the desired output. This process is known as a forward pass in deep learning terms. Later, using back propagation, we update our models' parameters.

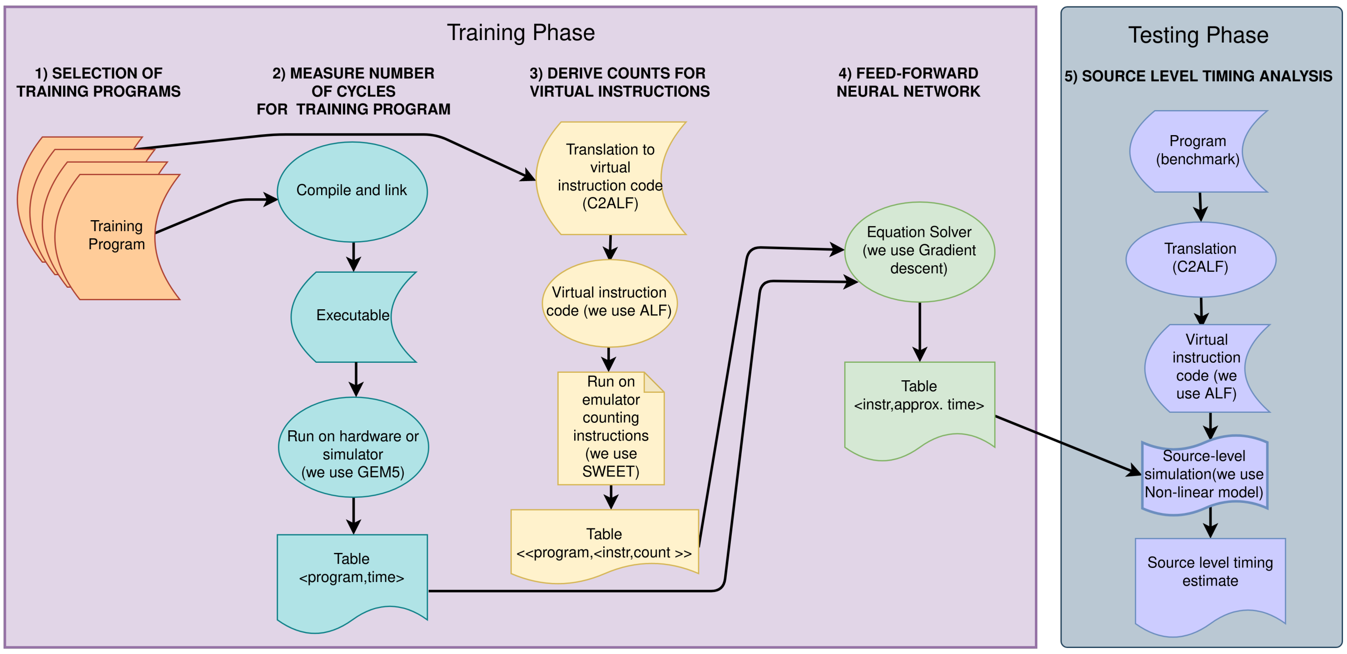
6.7 DNN Approach

Traditionally, WCET estimation has relied on static analysis techniques, such as abstract interpretation or model checking, to derive upper bounds on execution times. These methods often involve manual annotation or instrumentation of code, making them labor-intensive and prone to inaccuracies, especially in complex software systems. However, recent advancements in machine learning, particularly in the field of deep learning, have shown promise in automating WCET estimation with improved accuracy and efficiency. One such approach involves the use of feed-forward neural networks (FNN) to model the relationship between program characteristics and execution times. The detailed explanation of the proposed approach is explained below. Figure 6.6 depicts both the training and testing phases. The detailed explanation of the proposed approach is explained below. Figure 6.6 depicts both the training and testing phases.

The process begins with the selection of training programs, which represent a diverse range of tasks and execution patterns that the system is expected to encounter. These programs are compiled and executed on the target hardware or a simulator, and their execution times are measured. Next, the training programs are converted into a virtual instruction set, which abstracts away hardware-specific details and represents program behavior at a higher level. Using tools like SWEET (Statistical Wcet Estimation Enhancement Tool), the instruction count for each program is recorded. This step is crucial as it provides input features for training the neural network model.

The collected data, comprising both instruction counts and corresponding execution times, serve as the training dataset for the feed-forward neural network. The FNN is trained to learn the complex mapping between program characteristics (represented by instruction counts) and their corresponding WCETs.

Unlike previous approaches [7], that often relied on simplistic models such as linear regression, the FNN offers the advantage of capturing nonlinear relationships and interactions between



Training Phase
1) SELECTION OF TRAINING PROGRAMS
2) MEASURE NUMBER OF CYCLES FOR TRAINING PROGRAM
3) DERIVE COUNTS FOR VIRTUAL INSTRUCTIONS
4) FEED-FORWARD NEURAL NETWORK
Training Program
Compile and link
Executable
Run on hardware or simulator (we use GEM5)
Table <program,time>
Translation to virtual instruction code (C2ALF)
Virtual instruction code (we use ALF)
Run on emulator counting instructions (we use SWEET)
Table <<program,<instr,count >>
Equation Solver (we use Gradient descent)
Table <instr,approx. time>
Testing Phase
5) SOURCE LEVEL TIMING ANALYSIS
Program (benchmark)
Translation (C2ALF)
Virtual instruction code (we use ALF)
Source-level simulation(we use Non-linear model)
Source level timing estimate

Figure 6.6: Early-timing analysis approach using DNN.

Table 6.4: Extracted Features

# addition operations	# division operations	# subtraction operations
# logic operations	# function calls	# return statements
# load operations	# store operations	# multiplication operations
# shift operations	# jump statements	# comparison operations

input features. This enables more accurate WCET estimation, particularly for complex software systems with intricate control flow and data dependencies. Once the FNN is trained, it can be deployed for timing analysis of new programs or system configurations. Given the instruction count of a program, the FNN predicts its WCET with high accuracy, allowing developers to assess timing requirements and identify potential bottlenecks or performance optimizations.

6.8 Experimental Setup

Selection and construction of training programs are of utmost importance. Each training program is constructed using the extended approach presented in [7], which covers all the context-dependent timing effects due to hardware features such as caches, pipelines and branch prediction units, and code optimizations due to the compiler. The training programs are executed on the gem5 [20] simulator to measure the number of cycles which are used as labels. The same training programs are translated into a virtual instruction set using SWEET [95] tool.

Table 6.5: Layers and Properties of our Neural Networks

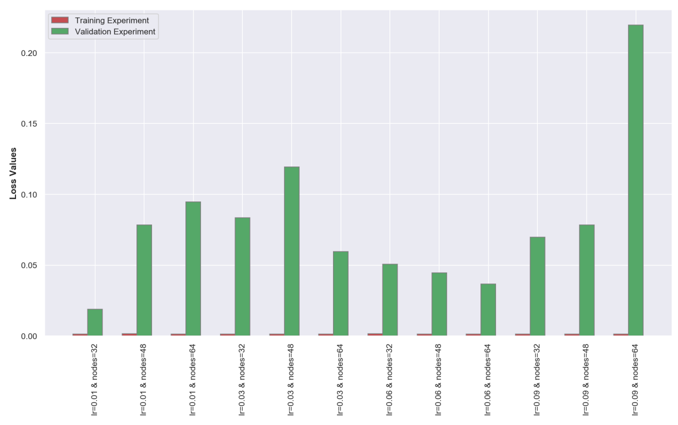
Dataset A	**Input**	**Layer 1**	**Layer 2**	**Layer 3**	**Layer 4**	**Properties**	
No. Neurons	12	32	32	32	1	Learning rate / Optimizer	0.01 / Adam
Activation f.	-	Leaky Relu	Leaky Relu	Leaky Relu	Leaky Relu	No. epochs / Batch size	100 / 10
Regularisation	-	L2 (β=0.01)	L2 (β=0.01)	L2 (β=0.01)	L2 (β=0.01)	No. Samples (train / test)	57 / 23
Dataset B	**Input**	**Layer 1**	**Layer 2**	**Layer 3**	**Layer 4**	**Properties**	
No. Neurons	12	32	32	32	1	Learning rate / Optimizer	0.03 / Adam
Activation f.	-	Leaky Relu	Leaky Relu	Leaky Relu	Leaky Relu	No. epochs / Batch size	100 / 40
Regularisation	-	L2 (β=0.01)	L2 (β=0.01)	L2 (β=0.01)	L2 (β=0.01)	No. Samples (train / test)	224 / 23

This virtual ISA acts as a feature set of the proposed predictor network. Combining these two essentially creates the data which can be appropriately used by neural networks.

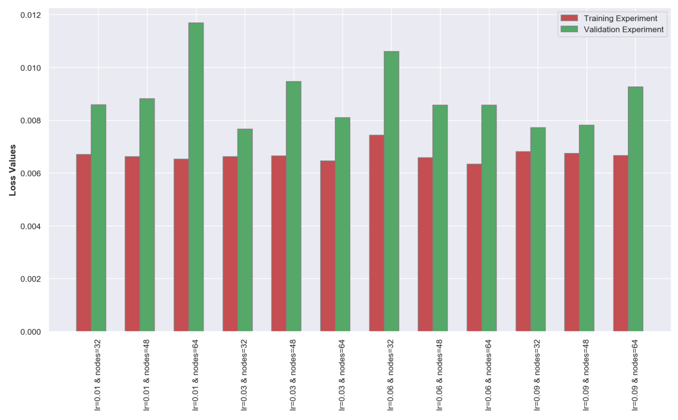
In deep learning, we need to pre-process and clean the data before feeding it to a neural network. We have used feature selection on our training data, and we found that the features shown in Table 6.4 are the most frequently occurring out of all the features. So we need to choose the features carefully [53]. We found that some features have values in the range of 200 - 800, and some features have values in the range of 2 - 10. These differences in the range of values bias the model's prediction to be inclined towards values of features that are larger, and the features having lower values contribute much less to the prediction, i.e., low value features have no significance in the Neural Network [55].

To tackle this issue, we need to normalize our data value in the range of 0 to 1. Several frameworks such as Pytorch [109] and Tensorflow [2] are available and provide a competitive arsenal of tools to perform this operation. We have used the Pytorch framework in this experiment. ALF format is suitable for our approach because it contains both high-level and low-level constructs. Statement such as CALL and RETURN represent high-level constructs, and a statements such as LOAD, STORE, and JUMP represent low-level constructs. Two datasets, A and B, are created. The total number of elements in the training datasets A and B are 57 and 224 respectively. 23 TacleBench programs have been used as testing data, and these programs are the same as those used in [7]. The training data is further divided into two parts: training, and validation through 5-fold cross-validation to check how well our model performs on the training data. The testing data is taken from the TacleBench benchmark which contains a test case for WCET analysis for different platforms. RMSE scores are used to compare our predicted value to the actual value. RMSE is calculated as the root of the mean of the squared differences between the predictions and the real values.

Gem5 [20] simulator is used to carry out all experiments to configure one processor with different attributes. The ARM810 processor is used with 5 stage pipeline, 8KB unified cache, MMU, and static branch prediction. Operations like floating points are implemented in the



(a) Dataset A

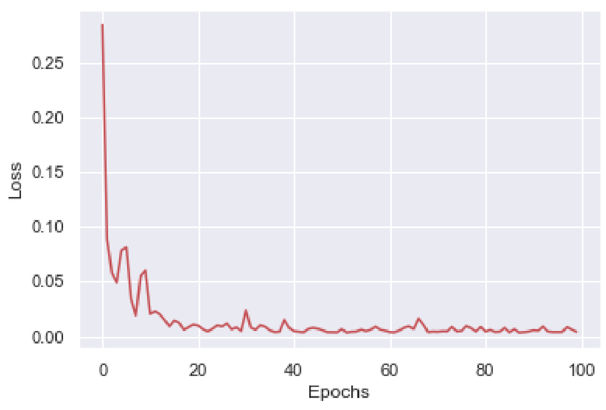


(b) Dataset B

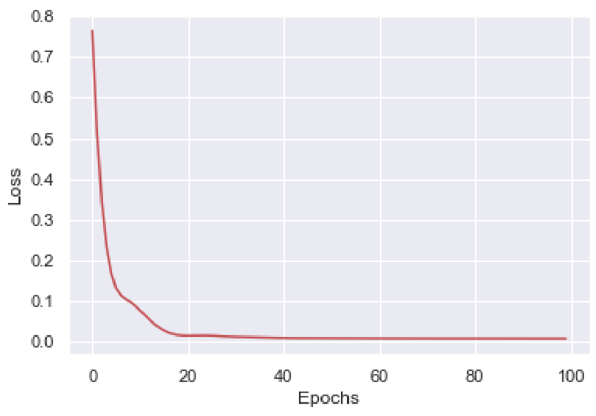
Figure 6.7: Comparison of Loss values using different configuration.

software. LLVM [88] compiler is used for compiling the test programs. Clang is used as a front-end to convert C source code into LLVM intermediate format. The LLVM IR file is given to ALF backend (C2ALF) to convert it into an ALF format, and the ARM backend (LLC) to convert it into an ARM object file. The SWEET tool is used to read the ALF code; it counts the number of different ALF constructs that appear as statements and operations.

We use hyperparameter tuning [40], such as grid search and randomized search, to find the best possible neural network configuration by modifying the hyperparameters like learning rate (lr), number of epochs, and different optimization and activation functions in other experiments. The best structure of the deep learning model is shown in table 6.5. We executed 12 different experiments by varying all the hyperparameters to find which model gives the least error on training and validation data. The comparison of different training and testing loss is shown in Figure 6.7. The X-axis represents a different experiment with hyperparameter tuning and the Y-axis loss values. In Figure 6.7a, the experiment with lr $= 0.01$ and neurons $= 32$ is the best one for dataset A as it gives the least error on both training and validation data. The training loss error is significantly less than the validation loss error because dataset A has a considerably smaller sample program. Similarly, the experiment with lr $= 0.03$ and neurons $= 32$ is the best for dataset B is shown in Fig 6.7b. Learning curve of the best model for both datasets is shown in Figure 6.8. The loss scale is different in both figures because of the different numbers of the samples in each dataset. The figure indicates some oscillation in loss values at the beginning for dataset A and is smooth for dataset B, but as the number of epochs increases with time, loss values start converging and saturate to zero. Hence, we limit the number of training epochs to 100.



(a) Dataset A



(b) Dataset B

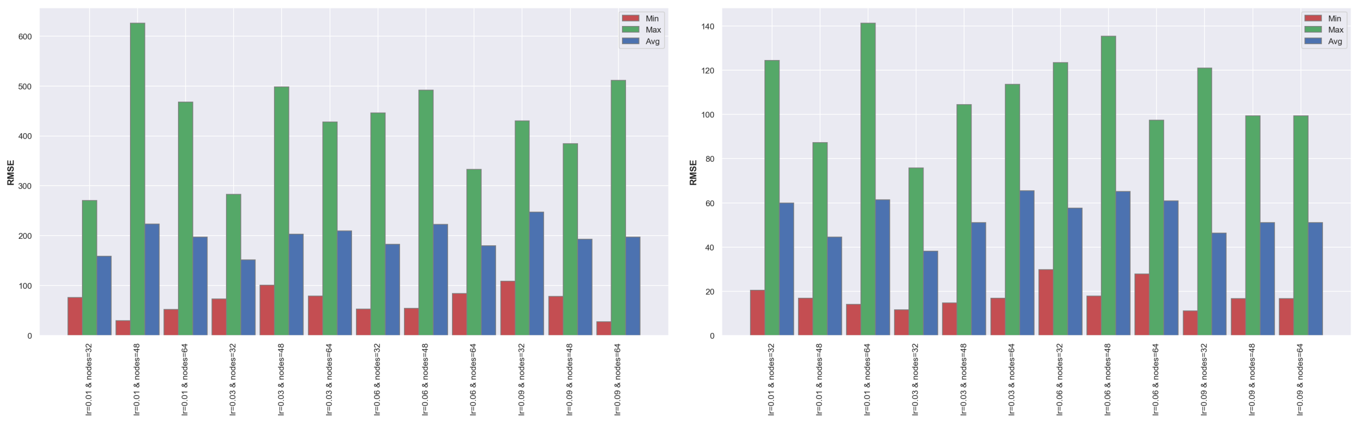
Figure 6.8: Comparison of Learning Curve.

6.9 Results

The results are shown in the graph in Figure 6; the percentage in the figure, and the data in Table 6.6 to get better insights into the model. We have executed our Deep learning model 12 times with each combination of different hyperparameters values. This allows us to calculate the minimum, maximum, and average error values for each configuration set up. The minimum, maximum, and average values are shown in Figure 6.9. For dataset A, we notice that the results with $lr = 0.01$ and $nodes = 32$ have shown better results where the minimum and average RMSE values are very close. Also, the maximum RMSE is comparatively close to these values. The trend in dataset B is different as the results with $lr = 0.03$ and $nodes = 32$ have shown the lowest error rate. Although there is variation in results, there is not a big difference between the minimum and maximum error values, which shows that, in most cases, our model has lower error rates as compared to few cases where the error rate is high. The large error is due to the considerable difference between the properties of training and testing datasets.

We compare our method with other methods in the literature by also pointing out similarities and differences between the two:

- We use statements and operations of source programs as a feature, similar to the approaches presented in [22, 66, 108, 67].
- Unlike [22] and [66], we do not use a static approach. Measurement-based approach is used in a target-hardware agnostic manner.
- Unlike [108] we do not use assembly instructions for feature categorization. We have used



(a) Dataset A (b) Dataset B

Figure 6.9: Comparison of Min, Max, and Avg RMSE values by executing the model using 12 different configuration settings.

Table 6.6: RMSE Errors of Neural Networks

Dataset	**Average Error (RMSE)**	**Min. Error (RMSE)**	**Max. Error (RMSE)**
	Training / Test	**Training / Test**	**Training / Test**
A	21% / 41.3%	17.8% / 23%	22.9% / 66.7%
B	12.7% / 20.6%	11.8% / 17.4%	14.8% / 23.5%

a source program to extract the features.

- Like in [67] we have used Root Mean Square Error (RMSE). However, our results are different because we have used different datasets and WCET strategies.

6.10 Summary

In this chapter, we present an approach that can be used to predict early WCET using an ML and DL. The model estimates WCET from the source code of the applications. Features are generated using the SWEET tool, i.e., the number of statements and operations in the source code, which are fed into our models to predict WCET after scaling data. Two datasets, A and B, are created with 57 and 224 samples, respectively. We demonstrate the model for ARM processors. We have used the Pytorch framework to implement classical ML models and feed-forward neural networks that converge quickly. The model performance is evaluated using metrics called the MSE and RMSE. The MSE value is calculated for ml models, and we calculated the minimum, maximum, and average RMSE for each distinct neural network configuration. The RMSE value for the bigger dataset is much better than the smaller dataset. The results shown in this chapter are not promising as they are too large to use as an upper bound. However, getting these numbers in the early stage of developing a system is useful in many ways, such as preventing systems designers from assuming a pessimistic upper bound on the WCET.

Chapter 7

Conclusion and Future Work

Accurately estimating the Worst-Case Execution Time (WCET) holds immense significance in the development of Real-Time and Embedded systems. The scheduler within an operating system heavily relies on this estimated WCET to efficiently manage and schedule tasks within these systems, ensuring their execution before designated deadlines. Failure to meet these timing constraints could result in catastrophic consequences, including resource damage or, in extreme cases, potential loss of life. These systems necessitate strict adherence to timing requirements, for example, ensuring that car airbags deploy swiftly enough to save lives in critical situations. The fundamental components essential for estimating WCET encompass the system's architecture or platform, the specific application under consideration, and worst-case data scenarios. In this context, our proposed methodology introduces novel approaches leveraging machine learning techniques to address these components. By integrating machine learning into the estimation process, we aim to derive precise and safe WCET estimates, thus enhancing the predictability and reliability of these systems beyond conventional methodologies. This innovative approach seeks to optimize safety and reliability by leveraging machine learning's capabilities to model intricate system behaviors, facilitating more accurate WCET predictions crucial for ensuring the robustness and effectiveness of Real-Time and Embedded systems.

In this thesis, we studied how to enable ML and DL approaches to estimate WCET. Especially we studied (i) the estimation of WCET on GPU architecture using ML approaches in Chapter 3, (ii) the estimation of optimistic WCET on mixed-criticality systems using ML approaches in Chapter 4, (iii) the determination of Worst-case data for the estimation of WCET by integration of Genetic Algorithm and machine learning in Chapter 5, (iv) the estimation of early WCET using ML and DL approaches in Chapter 6. Note that different scenarios have different issues that hinder us from estimating WCET precisely and tightly.

The purpose of this chapter is to summarize the discussion and findings in Chapters 3 through 6, address the limitations of the implementation and analysis, and present opportunities for future work.

7.1 Conclusions

This section summarizes the main contributions of our work in each scenario.

- **Estimation of WCET on GPU architecture (Chapter 3)**

  With the advances in machine learning and artificial intelligence in every field of life, due to its tendency to solve many problems with accuracy, it requires Graphics Processing Units (GPUs) to provide massive parallelism for computation. GPUs are designed to provide high-performance throughput, but their integration into real-time systems focuses on predictability because most safety-critical applications have strict deadlines that need to be followed to avoid unwanted situations. In this chapter, we propose a Machine Learning approach to estimate the WCET of the GPU kernel from the binary of the applications. The approach helps reduce the significant design space exploration in a short time. We use a measurement-based approach to train the machine-learning model using different kernel instructions, which can predict the WCET of the GPU kernel to detect timing misconfiguration in the later development phase of the systems.

- **Estimation of WCET on Mixed-Criticality Systems (Chapter 4)**

  In Mixed-Criticality (MC) Systems, there is a trend of having multiple functionalities upon a single shared computing platform for better cost and power efficiency. In this regard, estimating the suitable optimistic WCET based on the different system modes is essential to provide these functionalities, is presented in this Chapter. A single application has assigned multiple WCETs based on the criticality of the system, such as safety-critical, mission-critical, and non-critical. We propose ESOMICS, a novel method to estimate suitable optimistic WCET using a Machine Learning model. Our approach is based on the application's source, and the model is trained based on the large data set. To prove the effectiveness of our approach, we evaluated it with a newly defined metric EDT using an analytical solution that allows us to compute the optimum value in a mixed-criticality system based on experimentation. Our experimental results outperform all the previous state-of-the-art approaches.

- **Estimation of Worst-Case Data for WCET (Chapter 5)**

  Worst-Case Data which gives maximum execution time, plays a vital role in the estimation of WCET. An evolutionary algorithm, such as the Genetic Algorithm, has been employed to generate the Worst-Case Data. The complexity of an evolutionary algorithm requires the use of several computational resources. In this Chapter, we propose a method to replace the hardware and simulator used in the evolution process with Machine Learning models. This method reduces the overall time required to generate Worst-Case Data. Different machine learning models are trained to integrate with genetic algorithms. The feasibility of the proposed approach is validated using benchmarks from different domains. The results show the speedup in the generation of Worst-Case Data.

- **Estimation of Early WCET (Chapter 6)**

  WCET is available to us in the last stage of systems development when the hardware is available, and the application code is compiled. Different methodologies measure the WCET, but none give early insights into WCET, which is crucial for system development. If the system designers overestimate WCET in the early stage, then it would lead to an overqualified system, which will increase the cost of the final product, and if they underestimate WCET in the early stage, then it would lead to financial loss as the system would not perform as expected. In this Chapter, we propose to estimate early WCET using Machine Learning and Deep Neural Networks as an approximate predictor model for hardware architecture and compiler. This model predicts the WCET based on the source code without compiling and running on the hardware architecture. The resulting WCET needs to be revised to be used as an upper bound on the WCET. However, getting these results in the early stages of system development is an essential prerequisite for the system's dimension's and configuration of the hardware setup.

7.2 Future Work

In this section, we discuss some potential directions for the future work. These potential future directions are either some extensions or complementaries of the works presented in the main chapters.There are, however, several open issues that need to be addressed when designing such systems, including:

- **Safety and Reliability Management:** To avoid failure and disastrous outcomes, RTS systems must guarantee the proper execution of functions, particularly HC tasks, throughout run-time under a variety of scenarios (e.g., hardware faults, software errors, etc.). Thus, in order to guarantee long-term and application-specific dependability, RTS systems need to be carefully built. Fault-tolerance techniques are used in the design of such systems to ensure system safety. Different approaches, including replication or re-execution, are required in the event of fault incidence in order to strengthen their defenses against future failures. Furthermore, the requirements for dependability might vary among tasks due to the differing safety standards. The impact of a failure on a system varies depending on the task's criticality level, ranging from minimal to catastrophic. Therefore, considering cross-layer reliability is also crucial in efficiently designing these RTS systems that need to be focused on and deeply studied in the future.

- **Feature Engineering/Selection and Hyper-Parameter Tuning:** Feature engineering is a pivotal process in machine learning, where raw data is transformed or enhanced to create more informative and predictive features for training models. It involves selecting, creating, or modifying features that can improve a model's performance. Techniques in feature engineering may include scaling, normalization, one-hot encoding, binning, and generating new features through mathematical transformations or domain-specific knowledge. Effective feature engineering can significantly enhance a model's ability to discern patterns and relationships within the data, leading to improved predictive accuracy and generalization. In order to acquire a set of qualitative features, we need to gain insight into the actual data while looking for correlations between them.

  Parameter tuning, often referred to as hyperparameter optimization, involves fine-tuning the settings or configurations of a machine learning algorithm to achieve optimal performance. Hyperparameters control the learning process of the model and are distinct from the parameters learned during training. Techniques like grid search, random search, or more advanced methods like Bayesian optimization or genetic algorithms are used to explore various hyperparameter combinations efficiently. Optimizing these hyperparameters is crucial as it can substantially impact a model's predictive power, generalization ability, and overall effectiveness in tackling unseen data.

  Both feature engineering and parameter tuning are critical stages in building robust and accurate machine-learning models. While feature engineering focuses on crafting informative representations of data, parameter tuning aims to optimize the settings governing the learning process, collectively contributing to the model's predictive performance and adaptability across diverse datasets.

- **Bigger Dataset:** To train any prediction model, we require a large dataset in addition to features that are well-designed. The automated testbench is a first step towards rapidly gathering additional data. However, there are still ways to improve the upper bound measurement precision and expedite the profiling procedure. The flow graph's inclusion of static data analysis would be one of these enhancements. This would reduce the need for human code annotation by enabling us to locate and provide data input that seeks to trigger worst-case scenarios. An alternative strategy is to use augmentation techniques, as suggested by [82] and [207], to enhance the dataset's size. 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 Best Model:** Our focus primarily revolved around employing regression models for WCET prediction in our conducted tests. However, in contrast with regression models, alternatives such as better DNN models like CNN, RNN, and Transformer possess the capability to grasp intricate nonlinear system behaviors. Nevertheless, DNNs often require substantial amounts of training data to yield meaningful results. Another approach involves mitigating outliers within individual models by combining multiple models into an ensemble, culminating in a singular, highly predictive model. By combining various models spanning diverse domains, an ensemble model amalgamates superior performances, frequently outperforming the best individual model within the group.

A Appendix

The experimental analysis of different applications from the Mälardalen benchmark is shown below for the newly proposed metric *LTM/EDT*. Each Figures part (a) represents the execution distribution on the Y-axis, and the clock cycle is on the X-axis. In Figure part (b) *LTM/EDT* value is represented on the Y-axis, and the clock cycle is on the X-axis. These figures show us that through extensive and rich experiments, our proposed approach is a better choice of suitable WCET^{opt} because none of the state-of-the-art approaches estimate suitable WCET^{opt} with better accuracy as our approach.

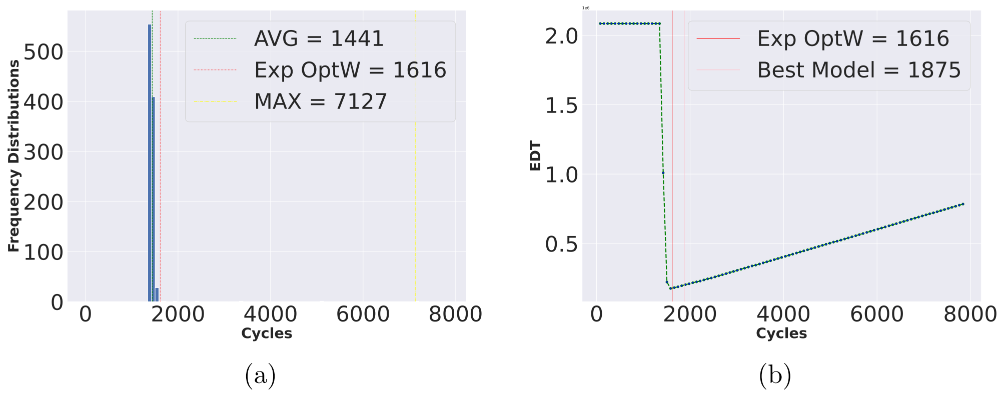


Figure 7.1: Analysis of cnt application

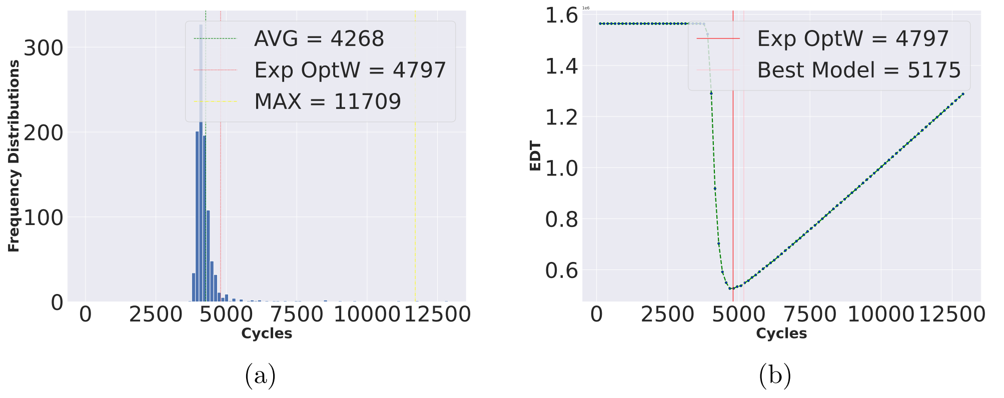
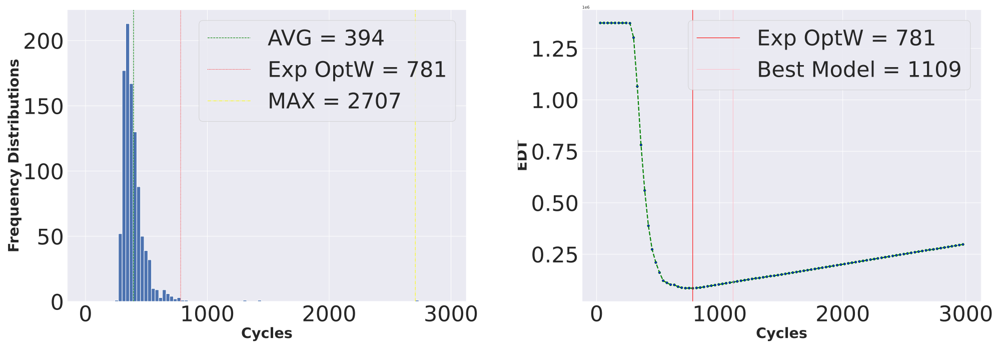
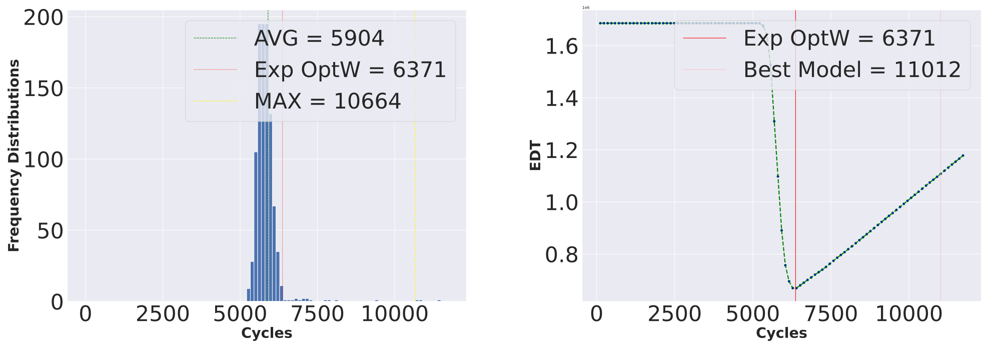


Figure 7.2: Analysis of compress application



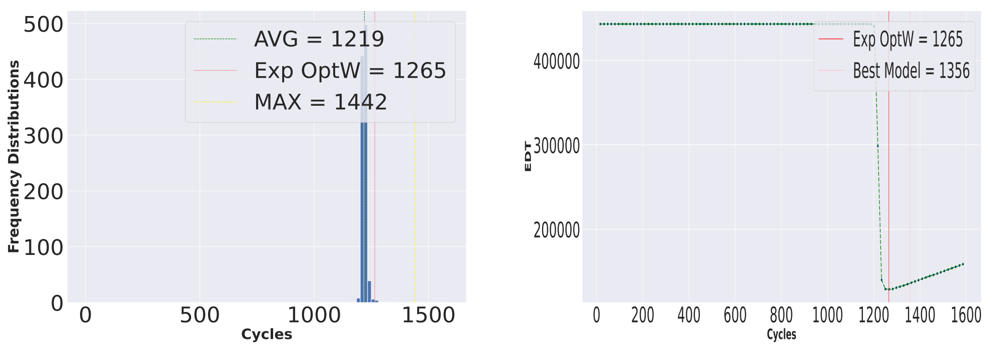
(a) (b)

Figure 7.3: Analysis of expint application



(a) (b)

Figure 7.4: Analysis of fdct application



(a) (b)

Figure 7.5: Analysis of insertsort application

Bibliography

[1] Geforce 256. Nvidia corporation, 1999. 39

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *Osdi*, volume 16, pages 265–283. Savannah, GA, USA, 2016. 10, 54, 118

[3] Andreas Abel and Jan Reineke. Measurement-based modeling of the cache replacement policy. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 65–74. IEEE, 2013. 109

[4] Jaume Abella, Carles Hernández, Eduardo Quiñones, Francisco J Cazorla, Philippa Ryan Conmy, Mikel Azkarate-Askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10. IEEE, 2015. 7

[5] Alan Agresti. *Categorical data analysis*, volume 482. John Wiley & Sons, 2003. 88

[6] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *Static Analysis: Third International Symposium, SAS'96 Aachen, Germany, September 24–26, 1996 Proceedings 3*, pages 52–66. Springer, 1996. 69

[7] Peter Altenbernd, Jan Gustafsson, Björn Lisper, and Friedhelm Stappert. Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 52: 731–760, 2016. 69, 74, 76, 88, 116, 117, 118

[8] Abderaouf N Amalou, Isabelle Puaut, and Gilles Muller. We-hml: hybrid wcet estimation using machine learning for architectures with caches. In *2021 IEEE 27th International*

Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pages 31–40. IEEE, 2021. 69, 76

[9] Rimmi Anand, Divya Aggarwal, and Vijay Kumar. A comparative analysis of optimization solvers. *Journal of Statistics and Management Systems*, 20(4):623–635, 2017. 19

[10] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE international symposium on performance analysis of systems and software*, pages 163–174. IEEE, 2009. 54

[11] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, et al. Enabling gpgpu low-level hardware explorations with miaow: An open-source rtl implementation of a gpgpu. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(2):21–1, 2015. 41

[12] Clément Ballabriga and Hugues Cassé. Improving the wcet computation time by ipet using control flow graph partitioning. In *8th International Workshop on Worst-Case Execution Time WCET Analysis (WCET'08)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008. 69

[13] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems: 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings 8*, pages 35–46. Springer, 2010. 23, 65, 107

[14] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo DAngelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 145–154. IEEE, 2012. 65, 66, 69, 77, 82, 83

[15] Iain Bate and Ralf Reutemann. Worst-case execution time analysis for dynamic branch predictors. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pages 215–222. IEEE, 2004. 69

[16] Kostiantyn Berezovskyi, Konstantinos Bletsas, and Björn Andersson. Makespan computation for gpu threads running on a single streaming multiprocessor. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 277–286. IEEE, 2012. 47

[17] Kostiantyn Berezovskyi, Konstantinos Bletsas, and Stefan M Petters. Faster makespan estimation for gpu threads on a single streaming multiprocessor. In *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–8. IEEE, 2013. 47

[18] Kostiantyn Berezovskyi, Luca Santinelli, Konstantinos Bletsas, and Eduardo Tovar. Wcet measurement-based and extreme value theory characterisation of cuda kernels. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, pages 279–288, 2014. 46

[19] Adam Betts and Alastair Donaldson. Estimating the wcet of gpu-accelerated applications using hybrid analysis. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 193–202. IEEE, 2013. xii, 59, 60

[20] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011. 117, 118

[21] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006. 25

[22] Armelle Bonenfant, Denis Claraz, Marianne De Michiel, and Pascal Sotin. Early wcet prediction using machine learning. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, pages 5–1. OASICs, Dagstuhl Publishing, 2017. 69, 88, 108, 120

[23] Axel Brando, Isabel Serra, Enrico Mezzetti, Jaume Abella, and Francisco J Cazorla. Using quantile regression in neural networks for contention prediction in multicore processors. In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022. 47, 69

[24] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. 30

[25] Claire Burguiere and Christine Rochange. History-based schemes and implicit path enumeration. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006. 69

[26] Alan Burns and Robert I Davis. A survey of research into mixed criticality systems. *ACM Computing Surveys (CSUR)*, 50(6):1–37, 2017. 65

[27] Giorgio Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. *Soft Real-Time Systems*, volume 283. Springer, 2005. 6

[28] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011. 6

[29] Maxim Buzdalov and Anatoly Shalyto. Worst-case execution time test generation for solutions of the knapsack problem using a genetic algorithm. In Linqiang Pan, Gheorghe Păun, Mario J. Pérez-Jiménez, and Tao Song, editors, *Bio-Inspired Computing - Theories and Applications*, pages 1–10, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-45049-9. 90

[30] CAST-32A. Certification authorities software team. 2016. URL `https://www.cast32a.com/files/cast-32a.pdf`. 8

[31] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009. 54

[32] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11. IEEE, 2010. 42

[33] Zhiyuan Chen, Jonathan Taylor, and Martin J Wainwright. Efficient ridge regularization via approximate leave-one-out. *Journal of Machine Learning Research*, 22(8):1–45, 2021. 29

[34] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, pages 63–74, 2010. 54

[35] Iryna De Albuquerque Silva, Thomas Carle, Adrien Gauffriau, and Claire Pagetti. ACETONE: Predictable Programming Framework for ML Applications in Safety-Critical

Systems. In Martina Maggio, editor, *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, volume 231 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-239-6. doi: 10.4230/LIPIcs.ECRTS.2022.3. URL https://drops.dagstuhl.de/opus/volltexte/2022/16320. 69

[36] Lachit Dutta and Swapna Bharali. Tinyml meets iot: A comprehensive survey. *Internet of Things*, 16:100461, 2021. 10

[37] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009. 2

[38] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016. 108, 109, 111

[39] Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France*, pages 377–383. Springer, 2004. 18, 24, 65, 107

[40] Matthias Feurer and Frank Hutter. Hyperparameter optimization. *Automated machine learning: Methods, systems, challenges*, pages 3–33, 2019. 119

[41] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1): 119–139, 1997. 31

[42] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001. 31

[43] Ahmed Fawzy Gad. Pygad: An intuitive genetic algorithm python library, 2021. 89

[44] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Nvidia. 2015. 42

[45] Nilanjan Goswami, Ramkumar Shankar, Madhura Joshi, and Tao Li. Exploring gpgpu workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–10. IEEE, 2010. 42

[46] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 innovative parallel computing (InPar)*, pages 1–10. Ieee, 2012. 54

[47] Jürgen Groß. *Linear regression*, volume 175. Springer Science & Business Media, 2012. 88

[48] Xiaozhe Gu and Arvind Easwaran. Dynamic budget management and budget reclamation for mixed-criticality systems. *Real-Time Systems*, 55:552–597, 2019. 70

[49] CUDA C Programming Guide. Nvidia. URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`. 41, 42

[50] CUDA Programming Guide. Cuda nvidia. 2007. 42

[51] Zhishan Guo, Kecheng Yang, Sudharsan Vaidhun, Samsil Arefin, Sajal K Das, and Haoyi Xiong. Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 373–383. IEEE, 2018. 65, 66, 69, 82, 83

[52] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010. 60, 66, 79

[53] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003. 118

[54] Raphael T Haftka, Diane Villanueva, and Anirban Chaudhuri. Parallel surrogate-assisted global optimization with expensive functions–a survey. *Structural and Multidisciplinary Optimization*, 54:3–13, 2016. 91

[55] Mark A Hall. Correlation-based feature selection of discrete and numeric class machine learning. 2000. 118

[56] Hwansoo Han and Chau-Wen Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, 2006. 54

[57] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The heptane static worst-case execution time estimation tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. 23, 65, 107

[58] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. The elements of statistical learning: Linear regression. *The Annals of Statistics*, 43(1):1–19, 2001. 27

[59] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009. 26

[60] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, 2008. 54

[61] Niklas Holsti and Sami Saarinen. Status of the bound-t wcet tool. *Space Systems Finland Ltd*, page 18, 2002. 24

[62] Adrian Horga, Sudipta Chattopadhyay, Petru Eles, and Zebo Peng. Measurement based execution time analysis of gpgpu programs via se+ ga. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 30–37. IEEE, 2018. xii, 46, 60, 61

[63] Biao Hu, Lothar Thiele, Pengcheng Huang, Kai Huang, Christoph Griesbeck, and Alois Knoll. Ffob: Efficient online mode-switch procrastination in mixed-criticality systems. *Real-Time Systems*, 55:471–513, 2019. 70

[64] Jensen Huang. https://video.ibm.com/recorded/132655512. *NVIDIA GPU Technology Conference*, 2023. 36

[65] Yijie Huangfu and Wei Zhang. Wcet analysis of gpu l1 data caches. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018. 47

[66] Thomas Huybrechts, Siegfried Mercelis, and Peter 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)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. 69, 76, 108, 120

[67] Thomas Huybrechts, Thomas Cassimon, Siegfried Mercelis, and Peter Hellinckx. Introduction of deep neural network in hybrid wcet analysis. In *Advances on P2P, Parallel, Grid, Cloud and Internet Computing: Proceedings of the 13th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2018)*, pages 415–425. Springer, 2019. 108, 120, 121

[68] Danial Jahed Armaghani, Mahdi Hasanipanah, Amir Mahdiyar, Muhd Zaimi Abd Majid, Hassan Bakhshandeh Amnieh, and Mahmood MD Tahir. Airblast prediction through a hybrid genetic algorithm-ann model. *Neural Computing and Applications*, 29:619–629, 2018. 91

[69] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41. IEEE, 2019. 37, 47

[70] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Stargazer: Automated regression-based gpu design space exploration. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 2–13. IEEE, 2012. 46

[71] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. Tango: A deep neural network benchmark suite for various accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 137–138. IEEE, 2019. 54

[72] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 3–12. IEEE, 2009. 42

[73] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 115

[74] Mohammadreza Koopialipoor, Danial Jahed Armaghani, Mojtaba Haghighi, and Ebrahim Noroozi Ghaleini. A neuro-genetic predictive model to approximate overbreak induced by drilling and blasting operation in tunnels. *Bulletin of Engineering Geology and the Environment*, 78:981–990, 2019. 91

[75] Hermann Kopetz and Wilfried Steiner. *Real-time systems: design principles for distributed embedded applications*. Springer Nature, 2022. 5

[76] VP Kozyrev. Estimation of the execution time in real-time systems. *Programming and Computer Software*, 42(1):41–48, 2016. 88

[77] Oliver Kramer. K-nearest neighbors. In *Dimensionality reduction with unsupervised nearest neighbors*, pages 13–23. Springer, 2013. 88

[78] Patrick Kwaku Kudjo, E Ocquaye, and Wolali Ametepe. Review of genetic algorithm and application in software testing. *International Journal of Computer Applications*, 160(2): 1–6, 2017. 88

[79] Vikash Kumar. Deep neural network approach to estimate early worst-case execution time. In *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, pages 1–8, 2021. doi: 10.1109/DASC52595.2021.9594326. 4, 14, 88

[80] Vikash Kumar. Deep neural network approach to estimate early worst-case execution time. In *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, pages 1–8. IEEE, 2021. 69, 76, 107

[81] Vikash Kumar. An integrated approach of genetic algorithm and machine learning for generation of worst-case data for real-time systems. In *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 87–95, 2022. doi: 10.1109/DS-RT55542.2022.9932054. 4, 16, 18

[82] Vikash Kumar. An integrated approach of genetic algorithm and machine learning for generation of worst-case data for real-time systems. In *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 87–95. IEEE, 2022. 13, 69

[83] Vikash Kumar. Estimation of an early wcet using different machine learning approaches. In Leonard Barolli, editor, *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 297–307, Cham, 2023. Springer International Publishing. ISBN 978-3-031-19945-5. 4

[84] Vikash Kumar. Estimation of an early wcet using different machine learning approaches. In Leonard Barolli, editor, *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 297–307, Cham, 2023. Springer International Publishing. 14, 76

[85] Vikash Kumar, Behnaz Ranjbar, and Akash Kumar. Utilizing machine learning techniques for worst-case execution time estimation on gpu architectures. *IEEE Access*, 12:41464–41478, 2024. doi: 10.1109/ACCESS.2024.3379018. 13

[86] Vikash Kumar, Behnaz Ranjbar, and Akash Kumar. Esomics: Ml-based timing behavior analysis for efficient mixed-criticality system design. *IEEE Access*, 12:67013–67024, 2024. doi: 10.1109/ACCESS.2024.3396225. 13

[87] Vikash Kumar, Behnaz Ranjbar, and Akash Kumar. Motivating the use of machine-learning for improving timing behaviour of embedded mixed-criticality systems. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024. 13

[88] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004. 119

[89] Stephen Law and Iain Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 189–199. IEEE, 2016. 90

[90] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553): 436–444, 2015. 113

[91] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. Fermi architecture white paper. 42

[92] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Systems*, 34:195–227, 2006. 69

[93] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007. 23, 65

[94] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, 1995. 69

[95] Björn Lisper. Sweet–a tool for wcet flow analysis. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II 6*, pages 482–485. Springer, 2014. 22, 66, 71, 73, 74, 75, 77, 109, 110, 117

[96] Björn Lisper and Marcelo Santos. Model identification for wcet analysis. In *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 55–64. IEEE, 2009. 69, 109

[97] Di Liu, Jelena Spasic, Nan Guan, Gang Chen, Songran Liu, Todor Stefanov, and Wang Yi. Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 35–46. IEEE, 2016. 65, 69

[98] Di Liu, Nan Guan, Jelena Spasic, Gang Chen, Songran Liu, Todor Stefanov, and Wang Yi. Scheduling analysis of imprecise mixed-criticality real-time tasks. *IEEE Transactions on Computers*, 67(7):975–991, 2018. 65, 82, 83

[99] John McCarthy, Marvin L Minsky, Nathaniel Rochester, and Claude E Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI magazine*, 27(4):12–12, 2006. 4, 9

[100] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. 4

[101] Gary C McDonald. Ridge regression. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(1):93–100, 2009. 88

[102] Hossein Moayedi, Arash Moatamediyan, Hoang Nguyen, Xuan-Nam Bui, Dieu Tien Bui, and Ahmad Safuan A Rashid. Prediction of ultimate bearing capacity through various novel evolutionary and neural network models. *Engineering with Computers*, 36:671–687, 2020. 91

[103] Nicolas Navet and Françoise Simonot-Lion. Fault tolerant services for safe in-car embedded systems, 2005. 3

[104] NHTSA. https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety. *Automated Vehicles for Safety*, 2023. 36

[105] NVIDIA-MPS. https://docs.nvidia.com/deploy/pdf/cuda multi process service overview.pdf. 2017. 37

[106] Molly A O'Neil and Martin Burtscher. Microarchitectural performance characterization of irregular gpu kernels. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 130–139. IEEE, 2014. 54

[107] Eva Ostertagová. Modelling using polynomial regression. *Procedia Engineering*, 48:500–506, 2012. 88

[108] Márcio Seiji Oyamada, Felipe Zschornack, and Flávio Rech Wagner. Accurate software performance estimation using domain classification and neural networks. In *Proceedings of the 17th symposium on Integrated circuits and system design*, pages 175–180, 2004. 108, 120

[109] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019. 10, 54, 118

[110] Fabian Pedregosa et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011. 112

[111] Tiago Pereira and Luís Silva. Deep polynomial regression. *Pattern Recognition*, 126: 108441, 2022. 28

[112] Jon Perez-Cerrolaza, Jaume Abella, Leonidas Kosmidis, Alejandro J Calderon, Francisco Cazorla, and Jose Luis Flores. Gpu devices for safety-critical systems: A survey. *ACM Computing Surveys*, 55(7):1–37, 2022. 4, 36

[113] Stefan M Petters. Bounding the execution time of real-time tasks on modern processors. In *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*, pages 498–502. IEEE, 2000. 20

[114] Michel Pignol. Dmt and dt2: two fault-tolerant architectures developed by cnes for cots-based spacecraft supercomputers. In *12th IEEE International On-Line Testing Symposium (IOLTS'06)*, pages 10–pp. IEEE, 2006. 3

[115] Hartmut Pohlheim and Joachim Wegener. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1795, 1999. 90

[116] Pupong Pongcharoen, C Hicks, PM Braiden, and DJ Stewardson. Determining optimum genetic algorithm parameters for scheduling the manufacturing and assembly of complex products. *International Journal of Production Economics*, 78(3):311–322, 2002. 96

[117] Victor Prisacariu, Ian Reid, et al. fasthog-a real-time gpu implementation of hog. *Department of Engineering Science*, 2310(9), 2009. 54

[118] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. 29

[119] Behnaz Ranjbar, Bardia Safaei, Alireza Ejlali, and Akash Kumar. Fantom: Fault tolerant task-drop aware scheduling for mixed-criticality systems. *IEEE access*, 8:187232–187248, 2020. 65

[120] Behnaz Ranjbar, Ali Hoseinghorban, Siva Satyendra Sahoo, Alireza Ejlali, and Akash Kumar. Improving the timing behaviour of mixed-criticality systems using chebyshev's theorem. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 264–269. IEEE, 2021. 65, 66, 69, 77, 78, 82, 83

[121] Raspberry. Raspberry pi platforms. *[Online]. Available: https://www. raspberrypi.org/*, 2020. 111

[122] Jan Reineke and Reinhard Wilhelm. Static timing analysis–what is special? In *Semantics, Logics, and Calculi*, pages 74–87. Springer, 2016. 88

[123] Daniel Rodriguez-Roman. A surrogate-assisted genetic algorithm for the selection and design of highway safety and travel time improvement projects. *Safety science*, 103: 305–315, 2018. 91

[124] EUROCAE / RTCA. Do-178c, software considerations in airborne systems and equipment certification. 2011. URL https://cdn.vector.com/cms/content/know-how/aerospace/Documents/Complete_Verification_and_Validation_for_DO-178C.pdf. 8, 66

[125] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. 4, 33

[126] Sujan Kumar Saha. *Spatio-temporal GPU management for real-time cyber-physical systems*. University of California, Riverside, 2018. 37

[127] Syed Abdul Baqi Shah, Muhammad Rashid, and Muhammad Arif. A prediction model for measurement-based timing analysis. In *Proceedings of the 6th International Conference on Software and Computer Applications*, ICSCA '17, page 9–14, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348577. doi: 10.1145/3056662.3056666. URL https://doi.org/10.1145/3056662.3056666. 89

[128] Jayati Singh. *Toward predictable execution of real-time workloads on modern GPUs*. PhD thesis, 2021. 37, 50

[129] Alex J Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222, 2004. 88

[130] Alex J Smola and Bernhard Schölkopf. Support vector regression. *Machine learning*, 54 (1):267–293, 2004. 30

[131] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 34

[132] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127:27, 2012. 54

[133] Anupama Surendran and Philip Samuel. Evolution or revolution: the critical need in genetic algorithm based testing. *Artificial Intelligence Review*, 48:349–395, 2017. 91

[134] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018. 26

[135] Vladimir Svetnik, Andy Liaw, Christopher Tong, J Christopher Culberson, Robert P Sheridan, and Bradley P Feuston. Random forest: a classification and regression tool for compound classification and qsar modeling. *Journal of chemical information and computer sciences*, 43(6):1947–1958, 2003. 88

[136] Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, 39(1):43–62, 1997. 115

[137] TESLA. https://static.nhtsa.gov/odi/rcl/2023/rclrpt-23v838-8276.pdf. *Safety Recall Report*, 2023. 36

[138] Mithun Haridas TP, R Naveen, and A Rajeswari. Reliable and affordable embedded system solution for continuous blood glucose maintaining system with wireless connectivity to blood glucose measuring system. 2013. 3

[139] L. S. Vailshery. Number of internet of things (iot) connected devices worldwide in 2018, 2025 and 2030. 2022. Available: https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/. x, 2, 3

[140] Peter Wägemann, Tobias Distler, Timo Hönig, Volkmar Sieh, and Wolfgang Schröder-Preikschat. Gene: A benchmark generator for wcet analysis. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. 73, 76

[141] Yu Wang and Jien Kato. Integrated pedestrian detection and localization using stereo cameras. *Digital Signal Processing for In-Vehicle Systems and Safety*, pages 229–238, 2012. 36

[142] Joachim Wegener, Klaus Grimm, Matthias Grochtmann, Harmen Sthamer, and Bryan Jones. Systematic testing of real-time systems. In *4th International Conference on Software Testing Analysis and Review (EuroSTAR 96)*, 1996. 90

[143] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based worst-case execution time analysis. In *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10. IEEE, 2005. 16

[144] NVIDIA Tesla P100 Whitepaper. Nvidia. 2016. 42

[145] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008. x, 4, 8, 9, 37, 65, 69, 88, 106

[146] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE, 2010. 41

[147] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130, 2015. 50, 51

[148] Haicheng Wu, Gregory Diamos, Si Li, and Sudhakar Yalamanchili. Characterization and transformation of unstructured control flow in gpu applications. In *1st international workshop on characterizing applications for heterogeneous exascale systems*, 2011. 43

[149] Tyler Yandrofski, Jingyuan Chen, Nathan Otterness, James H Anderson, and F Donelson Smith. Making powerful enemies on nvidia gpus. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 383–395. IEEE, 2022. 47, 48, 50, 57, 59

[150] Yurong Zhong. The analysis of cases based on decision tree. In *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 142–147, 2016. doi: 10.1109/ICSESS.2016.7883035. 88

[151] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005. 32