Multiobjective Compiler-Based Optimizations for Hard Real-Time Systems

Vom Promotionsausschuss der
Technischen Universität Hamburg

zur Erlangung des akademischen Grades

Doktorin der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von
Kateryna Muts

aus
Charkiw

2022

Creative Commons Lizenzvertrag
Der Text steht, soweit nicht anders gekennzeichnet, unter der Creative-Commons-Lizenz Namensnennung 4.0 (CC BY 4.0). Das bedeutet, dass er vervielfältigt, verbreitet und öffentlich zugänglich gemacht werden darf, auch kommerziell, sofern dabei stets der Urheber, die Quelle des Textes und o.g. Lizenz genannt werden. Die genaue Formulierung der Lizenz kann unter https://creativecommons.org/licenses/by/4.0/legalcode.de aufgerufen werden.

Gutachter: Prof. Dr. Heiko Falk
Prof. Dr. Matthias Mnich

Tag der mündlichen Prüfung: 2. November 2022

DOI : 10.15480/882.4799
ORCID iD: 0000-0002-2255-3410

Summary

An embedded system is a computer system consisting of software and hardware that is dedicated to a specific task within a larger system. Modern embedded systems are small devices operating on batteries and having a limited memory space, so their important restrictions are the code size and energy consumption of embedded programs.

An embedded systems that must react before a given deadline is called a real-time system. If violation of timing constraints might lead to catastrophic consequences, the system is called a hard real-time system. The most important property of a hard real-time system is its worst-case execution time.

The thesis introduces a new compiler-based compression technique that compresses chunks of a program at compile time and decompresses them at run-time. The technique guarantees that the compiled program with the runtime decompression satisfies its timing requirements. The results show that the method minimizes program size without violating timing constraints.

Many embedded systems have to fulfil multiple contradicting design criteria. This thesis presents a new approach to solve multiobjective problems at compile time when considering worst-case execution time, code size, and energy consumption as objectives. The thesis analyses evolutionary algorithms in terms of solving multiobjective problems at compile time. The results show that evolutionary algorithms can find trade-offs between the objectives but only for small programs since they require extensive evaluations of worst-case execution time and energy consumption, which are time-consuming processes.

To tackle this issue, the thesis introduces

- a prediction model based on machine learning techniques to predict worst-case execution time and energy consumption instead of costly estimating them;
- a search space reduction technique to run evolutionary algorithms on smaller search spaces.

Both techniques can speed up evolutionary algorithms and improve the quality of solutions, but evaluations show that the search space reduction leads to better results in terms of evolutionary algorithms' runtime and solution quality than the prediction model.

Kurzzusammenfassung

Die Arbeit präsentiert Compiler-basierte Mehrzieloptimierungen, um Kompromisse zwischen der maximalen Ausführungszeit, dem Energieverbrauch und der Codegröße von harten Echtzeitsystemen zu finden. Sie stellt eine neuartige Kompressionstechnik vor, die garantiert, dass das kompilierte Programm seine zeitlichen Beschränkungen erfüllt. Die Arbeit befasst sich mit der lang dauernden Lösung eines multikriteriellen Problems durch einen evolutionären Algorithmus. Um den Lösungsprozess zu beschleunigen, stellt die Arbeit eine Suchraumreduktionstechnik und ein Vorhersagemodell basierend auf maschinellem Lernen vor.

Abstract

The thesis presents multiobjective compiler-based optimizations to find trade-offs between the worst-case execution time, energy consumption, and code size of hard real-time systems. It introduces a novel compiler-based compression technique guaranteeing that a compiled program satisfies its timing constraints. The thesis tackles timing issues caused by solving multiobjective problems by evolutionary algorithms at compile time. The thesis introduces a search space reduction technique and a prediction model based on machine learning to speed up the solution process.

Publications

Parts of this thesis have been published in proceedings of conferences, workshops, and a book chapter listed below in chronological order:

- Kateryna Muts, Arno Luppold, and Heiko Falk. "Multi-Criteria Compiler-Based Optimization of Hard Real-Time Systems". In: *21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, May 2018. DOI: 10.1145/3207719.3207730
- Kateryna Muts, Arno Luppold, and Heiko Falk. "Compiler-Based Code Compression for Hard Real-Time Systems". In: *22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, May 2019. DOI: 10.1145/3323439.3323976
- Kateryna Muts and Heiko Falk. "Compiler-based WCET prediction performing function specialization". In: *23rd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, May 2020. DOI: 10.1145/3378678.3391879
- Kateryna Muts and Heiko Falk. "Multi-Criteria Function Inlining for Hard Real-Time Systems". In: *28th International Conference on Real-Time Networks and Systems (RTNS)*. ACM, June 2020. DOI: 10.1145/3394810.3394819
- Heiko Falk, Shashank Jadhav, Arno Luppold, Kateryna Muts, Dominic Oehlert, Nina Piontek, and Mikko Roth. "Compilation for Real-Time Systems a Decade After Predator". In: *A Journey of Embedded and Cyber-Physical Systems*. Springer International Publishing, July 2020, pp. 151–169. DOI: 10.1007/978-3-030-47487-4_10
- Kateryna Muts and Heiko Falk. "Predicting Worst-Case Execution Times During Multi-Criterial Function Inlining". In: *7th International Conference on Machine Learning, Optimization, and Data Science (LOD)*. Springer International Publishing, Oct. 2021. DOI: 10.1007/978-3-030-95467-3_21
- Kateryna Muts and Heiko Falk. "Predicting Objectives on a Reduced Search Space of Multiobjective Function Inlining". In: *24th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, Nov. 2021. DOI: 10.1145/3493229.3493303

Contents

1 Introduction **1**
1.1 Contribution 3
1.2 Structure 4

2 Embedded Systems **5**
2.1 Timing and Energy Consumption Analyses 7
2.1.1 Worst-Case Execution Time 7
2.1.2 Energy Consumption 10
2.1.3 WCET and Energy Consumption Analyses Tools 12
2.2 Embedded Microprocessors 17
2.2.1 ARM Cortex-M0 18
2.2.2 Infineon TriCore TC1797 18

3 WCET-Aware C Compiler **21**
3.1 Internal Structure 22
3.2 Flow Facts 23
3.3 Static Analyser Framework 26

4 Fundamentals **27**
4.1 Multiobjective Optimization 27
4.1.1 Problem Formulation 28
4.1.2 Pareto-Optimality 28
4.1.3 Quality Indicators 30
4.2 Evolutionary Algorithms 32
4.3 Machine Learning 36

5 Code Compression for Hard Real-Time Systems **39**
5.1 Related Work 40
5.2 WCET-Aware Compiler-Based Code Compression 44
5.2.1 Selection Model 47
5.2.2 Compression 53
5.2.3 Decompression 54
5.2.4 WCET Estimation 55

5.3 Evaluation 57
5.3.1 Experimental Setup 57
5.3.2 Compression Results 58
5.4 Conclusion 65

6 Evolutionary Algorithms for Multiobjective Compiler-Based Optimizations 67
6.1 Related Work 68
6.2 Differential Evolution 71
6.3 Multiobjective Function Inlining 78
6.4 Evaluation 83
6.4.1 Experimental Setup 83
6.4.2 Parameter tuning 84
6.4.3 Comparison of BGDE3 and MBPOA 89
6.5 Conclusion 93

7 Predicting Objectives at Compile Time 95
7.1 Related Work 96
7.2 Prediction Model 99
7.2.1 Classification 101
7.2.2 Selection of Best Classifiers 114
7.2.3 Building Prediction Models 116
7.2.4 WCC and Prediction Models 118
7.3 Evolutionary Algorithm and Prediction Model 120
7.4 Evaluation 121
7.4.1 Experimental Setup 121
7.4.2 Selection of Best Classifiers for Function Inlining 123
7.4.3 Evolutionary Algorithm Based on Predicted Objectives . . 126
7.5 Conclusion 137

8 Search Space Reduction 139
8.1 Related Work 139
8.2 Reduction Method 141
8.2.1 Selection Strategy 142
8.2.2 Search Space Reduction 149
8.3 Evaluation 150
8.3.1 Experimental Setup 151
8.3.2 Selection of Important Features 151
8.3.3 Search Space Reduction 154
8.3.4 Evolutionary Algorithm and Search Space Reduction . . . 159
8.4 Conclusion 167

9 Search Space Reduction and Prediction Model **169**
9.1 Method 169
9.2 Evaluation 172
9.2.1 Experimental Setup 172
9.2.2 Prediction Models on Reduced Search Spaces 172
9.3 Conclusion 183

10 Conclusion and Future Work **185**
10.1 Summary 185
10.2 Future Work 187

Bibliography **189**

Acronyms and Notational Conventions **213**
Acronyms 213
Commonly Used Symbols 215
Compression 215
Evolutionary Algorithm 215
Prediction of Objectives 215
Search Space Reduction 216
Reduced Search Space and Prediction Model 216

List of Figures **217**

List of Tables **221**

Appendices **223**

Appendix A Compression: Benchmarks **225**

Appendix B Function Inlining: Benchmarks **231**

Appendix C The Third Version of the Generalized Differential Evolution **233**

Appendix D Unique WCET and Energy Consumption Values for Function Inlining **237**

Appendix E Scores of Logistic Regression, Decision Tree, and AdaBoost **245**

1 Introduction

Computers, laptops, and smartphones have become an essential part of our everyday life but we often forget about many tiny computers called *embedded systems* that surround us every day. They are present in almost all aspects of our life: microwave ovens in our kitchens, antilock brakes in our cars, printers in our offices, etc.

An embedded system is a combination of software and hardware dedicated to performing a certain function. Many embedded systems must react before a deadline: airbag systems, medical monitoring devices, flight control systems, etc. They are called *real-time systems* since they must make calculations or decisions within a given time interval. Real-time systems differ in the severity of the consequences caused by a missed deadline. A failure of a *hard real-time system* may have catastrophic consequences (a defective flight control system endangers passengers' and crew's lives), whereas a failure in a *soft real-time system* may have harmless consequences (a defective radio receiver influences only the sound quality). A key property of hard real-time systems is *Worst-Case Execution Time (WCET)*, which is the worst possible execution time of a program independent from its input data [Mar11].

The code size of an embedded application is a key factor that influences the manufacturing cost of the system. Many embedded systems are implemented as System on a Chip (SOC) and have a very limited amount of memory. So besides timing requirements, real-time systems often have to satisfy *code size* constraints.

The portable devices market develops extensively and requires energy-efficient embedded systems. *Energy consumption* is a critical factor because it affects battery life and the intervals between battery charges. So minimizing energy consumption has become another common requirement of modern real-time systems.

If a real-time system violates its deadline, code size limitation, or energy consumption requirement, system designers can fix it at hardware and software levels. Upgrading hardware usually results in higher costs, whereas optimizing software requires fewer resources and prevents excessive costs. Nowadays, to design an embedded system (e.g. in avionics or automotive domains), a system designer

1. sets up requirements for the embedded system concerning performance, energy consumption, code size, etc.;

2. models the embedded system in graphical high-level tools like, e.g. MATLAB [MAT22] or LabVIEW [Bre13];

3. generates C code out of the graphical specifications from Step 2;

4. optimizes the generated C code by manually profiling the code, eliminating redundant parts — e.g. redundant variables — and optimizing loops;

5. compiles the optimized C code into an executable file for an architecture;

6. validates the requirements from Step 1 by simulating or executing the executable using a subset of all possible inputs;

7. adds safety margins to the values measured in Step 6 (e.g. WCET) to guarantee that the requirements are satisfied, since it is infeasible to validate all possible inputs in Step 6.

If any constraint from Step 1 is violated, the system designer has two options: to change the high-level specification of the system in Step 2 and repeat Steps 3–7, or to optimize further the C code in Step 4 and repeat Steps 5–7. The problem is that improving one design objective most likely degrades other objectives: e.g. WCET decrease often leads to code size increase. The contradicting objectives and many manual steps in the design flow complicate the system designer's task, but modern compilers offer many optimizations that automatically improve code quality and can simplify Steps 4–7. If the system designer has a set of trade-offs between the objectives, it can help to decide which specifications suit most of the system's requirements.

To optimize a program concerning several requirements at compile time, a common approach is to formulate a single-objective problem with one requirement as an objective and the others as constraints. Such an approach is appropriate if the goal is to improve one objective as much as possible and keep the others within predefined limits. The major advantages of the approach are:

1. the optimum objective value is unique;

2. such compiler-based optimizations can be formulated as an Integer Linear Programming (ILP) problem, which can often be efficiently solved by ILP solvers, e.g. Gurobi [Gur21] or CPLEX [Cpl17].

If no solution exists or the goal is to optimize the objectives simultaneously, then one formulates a multiobjective optimization problem. In this case, the objectives usually contradict each other and no unique solution exists that simultaneously optimizes all objectives. A solution to any multiobjective problem is a solution set comprising trade-offs between the objectives.

To solve a multiobjective optimization problem, widely used methods such as evolutionary algorithms extensively explore the search space of the problem to find a solution set. They evaluate objectives at each newly found point of the search space which makes it problematic to solve multiobjective problems at compile time because estimations of the objectives like WCET and energy consumption require time-consuming analyses.

This thesis extends compiler-based optimizations to the field of multiobjective optimizations considering WCET, code size, and energy consumption as objectives. We demonstrate application of the ILP and pure multiobjective approaches at compile time. We show that the solution process of the pure multiobjective approach is very time-consuming because of timing and energy analyses. We propose novel methods to speed up evolutionary algorithms at compile time and to provide solutions to multiobjective problems within a feasible time frame.

1.1 Contribution

The thesis formulates a standard compiler-based optimization as a multiobjective problem and proposes methods to find trade-offs between the WCET, code size, and energy consumption of a hard real-time system. We propose novel methods to speed up the solution process of multiobjective problems at compile time. The major contributions of the thesis are:

- We present a compiler-based code compression for hard real-time systems. It is a novel compiler-based optimization that allows to compress code chunks at compile time and decompress them at the execution time of the program without violating timing constraints. We explain the advantages and disadvantages of reformulating a multiobjective problem as a single-objective problem with additional constraints.

- We consider a pure multiobjective problem with WCET, energy consumption, and code size as objectives and evaluate evolutionary algorithms in terms of solving a multiobjective compiler-based optimization problem. We show that it is a time-consuming process if WCET and energy consumption are the objectives of the problem.

- We propose a novel method based on machine learning techniques to predict worst-case execution time and energy consumption at compile time to speed up evolutionary algorithms.

- We propose a novel method for reducing the search space of a multiobjective compiler-based optimization problem and speeding up evolutionary algorithms at compile time. The method removes irrelevant dimensions of the search space by using the values of the objective space.

- We combine the predictions of the objectives and the search space reduction technique to speed up evolutionary algorithms at compile time.
- We implemented the proposed approaches in the WCET-Aware C Compiler (WCC) to evaluate and show their practical advantages.

1.2 Structure

The thesis is structured as follows:

- Chapter 2 defines the fundamental properties of embedded systems (including hard real-time systems), explains the components of timing and energy consumption analyses, and shortly describes embedded architectures used in the thesis.
- Chapter 3 introduces the WCET-Aware C Compiler (WCC) used to show and evaluate the approaches proposed in the thesis.
- Chapter 4 introduces key concepts regarding multiobjective optimization and machine learning required for the approaches presented in the thesis.
- Chapter 5 describes the novel compiler-based code compression technique for hard real-time systems.
- Chapter 6 shows the method to solve multiobjective compiler-based optimization problems by using evolutionary algorithms and identifies the principal obstacles of the approach.
- Chapter 7 presents the novel method for predicting WCET and energy consumption at compile time in order to speed up the solution process of evolutionary algorithms.
- Chapter 8 introduces the novel technique to reduce the search space of compiler-based optimizations and shows the advantages of the method by solving multiobjective optimization problems.
- Chapter 9 describes an approach that combines the search space reduction technique, the prediction model, and an evolutionary algorithm to solve a multiobjective optimization problem at compile time.
- Chapter 10 presents a conclusion and possible future work.

2 Embedded Systems

Embedded systems play a crucial role in our daily lives today and appear in many products and devices, e.g. medical devices, GPS systems, central heating systems, etc. Marwedel defines an embedded system as follows:

Definition 2.1 ([Mar11])
Embedded systems *are information processing systems embedded into enclosing products.*

Embedded systems are combinations of hardware and software: e.g. an automated teller machine (ATM) is an embedded system used in banking that processes input from a keyboard, communicates with a bank computer over a network, and displays transaction data.

Embedded systems must be *efficient* concerning the following metrics:

- *Energy consumption*:

  Definition 2.2
  The energy *consumed by a system for computation is the integral of the electrical power dissipated over the time interval required to complete the computation.*

  Energy consumption is measured in the unit Joule.

  Many embedded systems operate on batteries, so a limited energy consumption keeps devices at comfortable temperatures and increases the time between battery charges.

- *Execution time*: according to Hennessy and Patterson [HP03], different ways exist to define execution time but the most straightforward definition is the following:

  Definition 2.3 ([HP03])
  The execution time *of a task is the latency to complete the task.*

  Execution time depends on disk and memory accesses, input/output and operating system activities, etc., so embedded systems should efficiently exploit the available resources to finish their tasks in time. E.g. if a compiler

introduces unnecessary overhead, it leads to an increasing execution time, higher clock rates, and wasted energy.

- *Code size*: an executable file contains a code section and data section. The code section stores the code of an application, e.g. a function, whereas the data section stores the data, e.g. global variables.

  Definition 2.4
  Code size *is the size of a code section,* data size *is the size of a data section.* Program size *is the total size of a program, i.e. the sum of code size and data size.*

  The code size of an embedded application must be limited because the code is usually stored with the system with no hard disks, whereas the capacity of volatile main memories and permanent flash memories is very limited.

- *Weight*: portable systems must be lightweight.

- *Cost*: an embedded system can compete in the market if its hardware and software development budget is kept low: a minimum amount of hardware and software resources should be used to implement the required functionality.

Kopetz gives the following definition of real-time systems:

Definition 2.5 ([Kop11])
A real-time system *is an embedded system where the correctness of the system behaviour depends not only on the logical results of the computations but also on the physical time when these results are produced.*

Many real-time systems are *cyber-physical systems* that are connected to the physical environment through *sensors* to collect information about the environment and *actuators* that convert numerical values into physical effects to control the environment.

An example of a real-time system is a wearable device that checks patients' health. It monitors the heart rate, blood pressure, glucose, weight, etc. The device collects patient data and helps to prescribe treatments and medications. Such technologies augment the quality of healthcare and reduce the cost of medical management. Many real-time systems must meet *real-time constraints*. Failing to complete computations within a given deadline can degrade quality (e.g. the audio quality of a radio receiver) or cause harm (e.g. a defective car airbag system).

Definition 2.6 ([Kop11])
A real-time system that must meet at least one hard deadline is called a hard real-time

system *or a* safety-critical real-time system. *If no hard deadline exists, then the system is called a* soft real-time system.

E.g. a car airbag system includes an airbag control unit that detects accidents through sensors and deploys the airbag depending on the crash severity. If the control unit misses its deadline, it might have dramatic consequences for the passengers.

We organized this chapter as follows: Section 2.1 describes timing and energy analyses necessary to guarantee the correctness and efficiency of a real-time system, Section 2.2 presents embedded microprocessors considered in the thesis.

2.1 Timing and Energy Consumption Analyses

Any real-time system must perform calculations functionally correct and within a predefined time frame. Another significant criterion of modern real-time systems is energy consumption: low energy consumption prevents overheating of devices and increases the time between battery charges. Timing and energy analyses help to guarantee that a hard real-time system always meets its deadline and keeps energy consumption at a low level. Since many parts of timing and energy analyses used in the thesis coincide, Sections 2.1.1 and 2.1.2 present a general overview of existing timing and energy analyses and Section 2.1.3 describes the main parts of the analyses and a tool used to perform the analyses during evaluations of the approaches proposed in the thesis.

2.1.1 Worst-Case Execution Time

Worst-case execution time helps to verify the timing constraints of real-time systems. Marwedel defines worst-case execution time as follows:

Definition 2.7 ([Mar11])
The Worst-Case Execution Time (WCET) *of a program is the largest execution time of the program for any input and any initial execution state.*

WCET is measured in clock cycles. If the WCET of a real-time system is less than a given limit, it guarantees that the system always satisfies its timing constraint.

Programs for real-time systems must always terminate (it is achieved by explicitly bounding recursions and loops in the programs), so the WCETs of such systems are always finite, but WCET computation is a nontrivial task. One way to get WCET of a program is to solve the halting problem that determines whether the program will terminate for a given input. The idea is to solve the problem

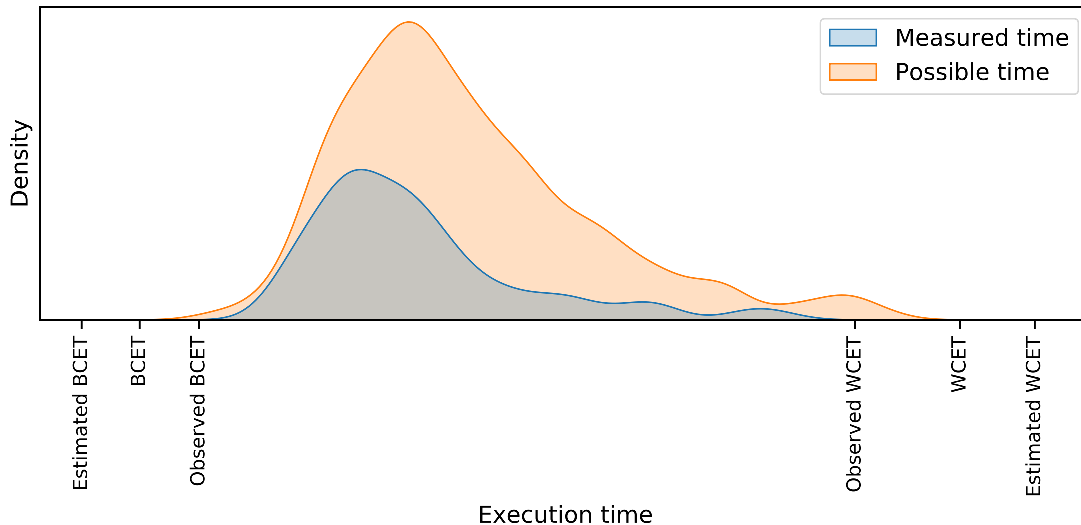


Figure 2.1: An exemplary distribution of execution time adapted from Wilhelm et al. [Wil+08]. The blue (bottom) curve represents the density of a subset of measured execution times. Their minimum and maximum are the observed Best-Case Execution Time (BCET) and the observed Worst-Case Execution Time (WCET). The orange (top) curve represents the density of all possible execution times. Their minimum and maximum are the BCET and the WCET.

for all possible inputs, but in 1937, Turing [Tur37] proved that there does not exist a general algorithm that solves the halting program for all possible programs and inputs. Another way is to compute the WCET by knowing a worst-case input, but the state space is usually large and cannot be fully explored to find the worst-case input.

Measurement-based [Pet00; BCP02; Wen+08], static [SP10; Mar+14; Abs22], and probabilistic [HHM09; Caz+12; DCG19] methods are three main approaches of timing analysis presented in the literature.

Measurement-based methods: A program is executed on hardware or simulator with some given inputs. Figure 2.1 shows an example of a possible WCET distribution for all possible inputs (orange curve) and a measured WCET distribution observed for the given inputs (blue curve). The figure also presents the Best-Case Execution Time (BCET) which is the smallest execution time of the program. A safe estimation of BCET must be smaller than or equal to the actual BCET, whereas a safe estimation of WCET must be greater than or equal to the actual WCET. Since execution time can be measured only for a subset of all possible inputs, the observed BCET is larger than the BCET, whereas the observed

WCET is less than the WCET, so measurement-based methods cannot prove that real-time systems satisfy their timing constraints.

Static methods: A lower bound of BCET and an upper bound of WCET are obtained by combining the control flow of a program with an abstract hardware architecture model without executing the program on a real hardware or a simulator. The methods underestimate BCET and overestimate WCET as shown in Figure 2.1. Static methods are used for hard real-time systems, since they guarantee that estimated BCETs and WCETs are *safe*: the actual BCET of a program is greater or equal to the estimated BCET and the WCET is less or equal to the estimated WCET. Often, to prove that a real-time system satisfies its timing constraint, its estimated WCET must also be *tight*, i.e. the estimated WCET is close to the actual WCET, because otherwise, the timing analysis might show that the system violates its timing constraint, whereas it does not.

Since static methods are suitable for hard real-time systems and are often used to get WCETs at compile time, in the thesis, we rely on one of such methods and describe the workflow of its static WCET analysis in Section 2.1.3.

Probabilistic methods: Whereas measurement-based and static methods aim to find a single upper bound of the actual WCET of a program, probabilistic methods seek a tight upper bound of probabilistic WCET distribution, also known as pWCET distribution. Two major categories of probabilistic timing analysis are Measurement-Based Probabilistic Timing Analysis (MBPTA) and Static Probabilistic Timing Analysis (SPTA).

MBPTA estimates the pWCET distribution by executing a program on hardware or a cycle-accurate simulator and sampling the observed execution times. E.g. it was used by Cucu-Grosjean et al. [CG+12] and Hernandez et al. [Her+15]. Most MBPTA methods rely on Extreme Value Theory (EVT) described by Ferreira and Haan [FH10]; it assesses the probability of an event to be more extreme than any observed one, i.e. the probability of execution time to be larger than any previously observed execution time. The major challenge of MBPTA is to identify the sample of inputs representing all possible inputs. (The representative sample might not exist.)

SPTA estimates the pWCET distribution by utilizing an abstract hardware model and analysing the program's code. It was used by Cazorla et al. [Caz+12] and Altmeyer and Davis [AD14]. According to Edgar and Burns [EB01], in the context of SPTA, pWCET distribution represents the confidence that WCET is not greater than some arbitrary value x. SPTA methods rely on the probabilistic execution time distribution of each instruction and derive the overall probability distribution for the execution time by combining the instructions' distributions with possible paths of the program. In contrast to traditional static methods,

using probability distributions, SPTA accounts for some random behaviour in hardware, software, or inputs. The major challenge of SPTA is an accurate modelling of hardware components and their interactions.

Both probabilistic timing analyses require hardware that fulfils certain requirements to guarantee a safe upper bound of pWCET distribution. E.g. Hernandez et al. [Her+15] assumed that caches are turned off or follow a random replacement policy to guarantee that the difference in the execution time of an instruction is random and independent of the previous measurements.

2.1.2 Energy Consumption

Modern embedded systems must be efficient in terms of energy since many of them operate on batteries. Two main methods exist to determine the energy consumption: physical measurements [TN65; GWW10; Bak+14] and estimation methods [LGT08; ZY15; Man+19].

Physical measurements: The energy consumption of an operation is the integral of the *power consumption* over the time required to perform the operation. The main methods to measure power consumption are shunt resistor approach and oscilloscope approach.

The *shunt resistor approach* calculates power consumption P by using the voltage drop over a shunt resistor that intercepts the power supply circuit:

$$P = V_{bus} \frac{V_{bus} - V_{shunt}}{R_{shunt}}, \tag{2.1}$$

where V_{bus} and V_{shunt} are the voltages measured on the two sides of the resistor and R_{shunt} is the resistance of the shunt resistor.

During a software development process, applying such methods is infeasible in most cases, since

- it is time-consuming if the measurements must be replicated across multiple devices;
- most software developers do not know how to change hardware to access the power supply – it requires advanced hardware knowledge.

The *oscilloscope approach* invokes oscilloscopes – devices to measure voltage U and current I. Power consumption P is then computed by $P = U \cdot I$. Such approaches can produce more accurate results than the shunt resistor approach, but oscilloscopes are expensive and generate a lot of data requiring post-processing.

Estimation methods: According to Georgiou, Souza, and Eder [GSE18], estimation methods comprise two major components: a technique to estimate energy consumption and an energy model (for a given architecture) to convey energy information to the estimation technique. Energy models rely on energy costs and static energy consumption: the energy costs represent the energy of events such as memory access or instruction execution, whereas the static energy consumption corresponds to the energy consumption of the idle hardware.

The state-of-the-art approaches to estimate energy consumption are based on profiling or Static Resource Analysis (SRA) according to Georgiou et al. [GSE18].

Profiling-based energy consumption estimation: These methods collect execution statistics and pass them to an energy model. Simulation, code instrumentation, and Performance Monitoring Counters (PMC) are the fundamental techniques for collecting execution statistics.

The *simulation* is usually performed at the Instruction Set Architecture (ISA) level following Tiwari et al. [Tiw+96]. ISA simulation is faster than hardware simulation, but energy consumption estimation at the ISA level is usually insufficient for complex architectures because of hardware components like, e.g. caches — it is difficult to capture their behaviour.

Code Instrumentation instruments code with additional instructions to extract execution statistics at runtime, e.g. as shown by Santos et al. [San+17]. This method is faster than simulation-based estimations, but its energy consumption estimation is less accurate because of the instrumentation overhead.

PMC methods extract execution statistics by counting certain performance events like, e.g. data dependency or cache miss as presented by Contreras and Martonosi [CM05]. Schubert et al. [Sch+12] stated that these counts help to estimate the energy consumption of complex architectures but it can be challenging because one can monitor only some types of events and simultaneously sample only a few counters.

SRA-based energy consumption estimation: An alternative to profiling is Static Resource Analysis (SRA). The most popular static technique to estimate energy consumption uses Implicit Path Enumeration Technique (IPET) introduced by Li and Malik [LM97]; it retrieves the most energy-consuming control flow path of a program based on energy costs. SRA-based estimation methods model the behaviour and energy consumption characteristics of a processor and perform IPET to find an energy consumption bound. Georgiou et al. [GSE18] stated that SRA successfully analyses predictable architectures and software, but it becomes a challenging task if a program is multi-threaded.

An estimated energy consumption is less accurate than a measured one but the estimations are faster and cheaper than the measurements, so the energy

analysis tool used in the thesis relies on an estimation technique. We describe the WCET and energy consumption analyses tool in the next section.

2.1.3 WCET and Energy Consumption Analyses Tools

In the thesis, we use AbsInt's Static WCET Analyser (aiT) [Abs22] and Static Energy Consumption Analyser (EnergyAnalyser) [Tea]. aiT supports different target architectures like ARM, TriCore, LEON, etc., and determines tight and safe upper bounds of WCETs as shown by Heckmann et al. [Hec+03], Baufreton and Heckmann [BH07], and Kästner et al. [Kä+08]. EnergyAnalyser supports the ARM Cortex-M0 and LEON3 processors. We shortly describe the main parts of aiT and EnergyAnalyser; for more details, we refer to AbsInt's website [Abs22] and the deliverable of the TeamPlay Horizon2020 project [Tea].

Figure 2.2 shows AbsInt's WCET and energy analyses framework. At the shared part of the analyses, the tool takes a fully linked binary executable and user annotations of flow facts described in Section 3.2, and reconstructs the control flow graph of the program.

Definition 2.8
The Control Flow Graph (CFG) *of a program is a directed graph that represents all possible execution paths of the program.*

A node of a CFG usually represents a basic block, and an edge represents the transfer of the execution from one basic block to another. Muchnick gives the following definition of basic blocks:

Definition 2.9 ([Muc98])
A basic block *is a maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them.*

When reconstructing the CFG from an executable, a decoder identifies machine instructions, e.g. call and return instructions, branch instructions, etc., groups sequential instructions into basic blocks, and connects the related basic blocks.

Figure 2.3 presents an example of an input code and its CFG. The resulting CFG contains four nodes, i.e. the input code has four basic blocks. The if-statement in the program causes branches in the CFG. Depending on the value of the variable a, the execution is transferred from the node B1 to the node B2 (if $a < 10$) or to the node B3 (if $a \geqslant 10$).

Value analysis mentioned in Figure 2.2 safely approximates the values of registers and memory cells, whereas *control flow analysis* uses its results to identify contradictory conditions, eliminate infeasible paths, and determine the execution frequencies of paths.

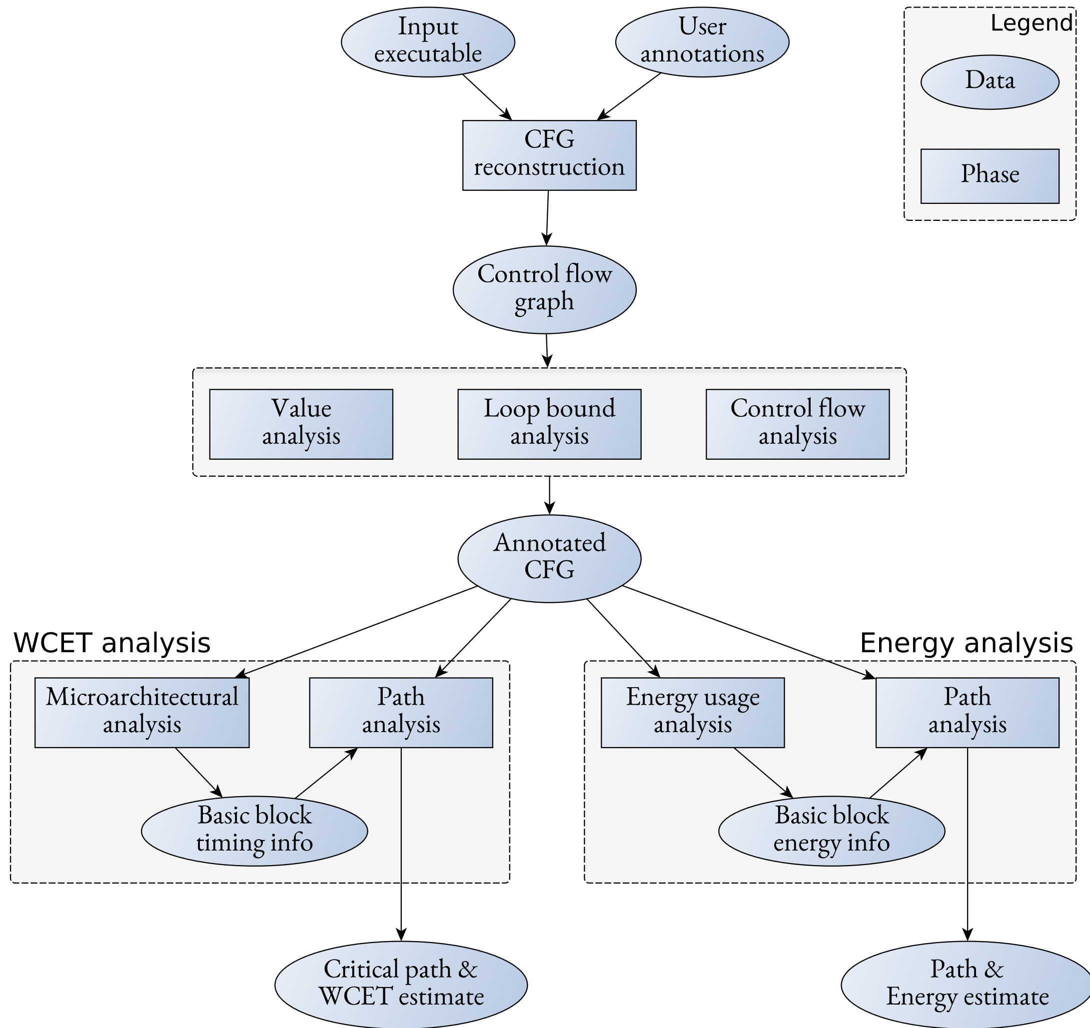
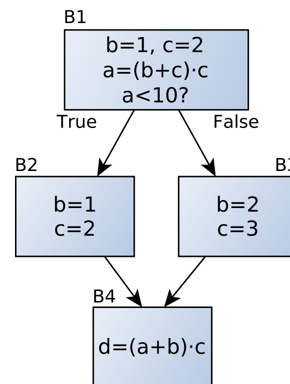


Figure 2.2: WCET and energy analyses framework adapted from Cullmann et al. [Cul+10] and the deliverable of the TeamPlay Horizon2020 project [Tea].

A program spends most of its execution time in loops and/or (recursive) functions, so WCET and energy consumption depend on the number of loop iterations and recursion depth. *Loop bound analysis* estimates the upper bounds of loop iterations and recursion depths to determine the execution frequency of a path. If a loop or recursion is complex and loop bound analysis fails to determine the upper bounds, then a programmer must provide loop bound and recursion depth annotations. We describe the annotations in Section 3.2.

```
1:  b = 1
2:  c = 2
3:  a = (b + c) · c
4:
5:  if a < 10 then
6:      b = 1
7:      c = 2
8:  else
9:      b = 2
10:     c = 3
11:
12: d = (a + b) · c
```

(a) Original program.



(b) Control flow graph.

Figure 2.3: Example of a control flow graph.

WCET Analysis

The WCET analysis starts with *microarchitectural analysis*. It determines the execution times of basic blocks using an abstract model of a target architecture. The microarchitectural analysis comprises cache and pipeline analyses.

Definition 2.10
Cache memory *stores frequently or recently used instructions or data.*

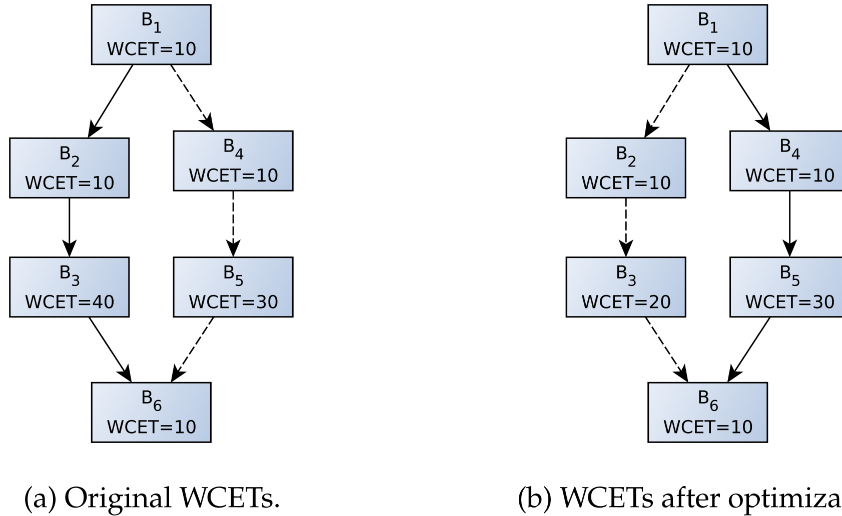
Cache memory is characterized by a short access time in the case of a cache hit (the searched instruction or data is in the cache memory) and a potentially long time in the case of a cache miss (the searched instruction or data is missed in the cache memory). We present more details on caches in Section 2.2. The variability of access times makes it challenging to compute a safe and tight WCET estimate. *Cache analysis* classifies memory references as cache hits and potential cache misses.

Definition 2.11
Pipeling *parallelizes the processing of machine instructions by dividing them into sequential parts performed by different processor units.*

Examples of processor units are an arithmetic logic unit, address generation unit, memory management unit, etc.

Pipeline analysis predicts the behaviour of the processor pipeline and peripheral devices. It outputs the WCETs of basic blocks.

The last step of the WCET analysis is *path analysis*. It uses the WCETs of basic blocks to determine the Worst-Case Execution Path (WCEP) of the program,



(a) Original WCETs. (b) WCETs after optimization.

Figure 2.4: Example of a WCEP switch adapted from Lokuciejewski and Marwedel [LM11]. The CFG consists of six basic blocks and the corresponding WCETs. The solid edges represent the WCEP.

which is its most time-consuming execution path. Only parts of the code lying on the WCEP contribute to the WCET, so by shortening (optimizing) the WCEP, we reduce the WCET of the program.

Optimizing the code of a program, its WCEP should be recomputed due to its instability. Figure 2.4 shows an example of a WCEP switch because of optimization. Figure 2.4(a) presents the original CFG consisting of six basic blocks and their WCETs. The solid edges show the WCEP of the program. Its original WCET is equal to 70 cycles. Figure 2.4(b) presents the program, where the basic block B_3 is optimized such that its WCET decreases by 20 cycles. In this case, the WCEP switches to the right branch and the WCET of the program becomes 60 cycles.

aiT's path analysis is formulated as a flow maximization problem by using Implicit Path Enumeration Technique (IPET) and an ILP model constructed from CFG. We refer to the original paper of Li, Malik, and Wolfe [LMW95] for more details on the ILP formulation of the path analysis.

Energy Consumption Analysis

In the thesis, we used AbsInt's energy consumption model for the ARM Cortex-M0 processor, so we describe it here.

Energy usage analysis introduced in the deliverable of the TeamPlay Horizon2020 project [Tea] is based on an energy model that estimates energy con-

Table 2.1: Counters of the energy model defined in Equation (2.2). Adapted from the deliverable of the TeamPlay Horizon2020 project [Tea].

i	Coefficient β_i	Counter C_i	Description
1	0.97256503	Ins	Total number of instructions without multiplication instructions
2	0.65287177	RAM_{read}	Total number of data-read accesses from RAM
3	1.031341343	RAM_{write}	Total number of data-write accesses from RAM
4	1.037625441	$Flash_{data}$	Total number of data-read accesses from FLASH
5	1.354953706	$Brch$	Total number of taken branches
6	2.274650563	Mul	Total number of multiplication instructions

sumption in nJ for a basic block. The energy model is a linear regression model that predicts energy consumption E for each basic block b:

$$E^b = \sum_{i=1}^{6} \beta_i \cdot C_i^b + \alpha, \tag{2.2}$$

where

- β_i are constants independent from basic blocks;
- C_i^b are basic block's event counters presented in Table 2.1;
- α is an error term independent from basic blocks.

Table 2.1 describes the counters C_i used in the model. In the deliverable [Tea], the authors have shown that these six event counters apply to energy consumption. The authors also determined the independent constants β_i and α by training the energy model (2.2) with BEEBS benchmarks introduced by Pallister, Hollis, and Bennett [PHB13]. Table 2.1 presents the determined coefficients β_i; the determined error term α equals 0.

Similar to the WCET analysis, knowing energy consumption for each basic block, EnergyAnalyser's *path analysis* determines the program path with the highest energy consumption.

2.2 Embedded Microprocessors

In the thesis, we perform evaluations using the WCET-Aware C Compiler (WCC) described in Chapter 3. WCC supports TC1796 and TC1797 microprocessors from the Infineon TriCore as well as ARM7TDMI and Cortex-M0 microprocessors from the ARM family. We consider the following microprocessors: ARM Cortex-M0 and Infineon TriCore TC1797. We shortly describe their fundamental structures.

Both microprocessors are based on the Reduced Instruction Set Computer (RISC) architecture described by Aletan [Ale92]. RISC architectures support a small set of instructions, which are highly optimized. RISC processors usually feature the following basic stages in the pipeline:

1. *instruction fetch*: the Central Processing Unit (CPU) reads an instruction from the memory address stored in the program counter;
2. *instruction decode*: the instruction is decoded and, for the registers used in the instruction, values are extracted;
3. *instruction execute*: the Arithmetic Logic Unit (ALU) performs instruction's operations;
4. *memory access*: the instruction's memory operands are read from the memory and written to the memory;
5. *write back*: the result of the instruction is written into a register.

RISC processors in general have a different number of pipeline stages, but they are a combination of the basic stages.

RISC architectures usually require many fast instructions to execute a task, so instructions should be read fast from the memory to reduce idle times. Since fast memories are expensive, small fast memories called *caches* buffer data from the main memory. A cache controller autonomously manages caches at the hardware level and decides which data to be stored in caches. Caches store frequently or recently used data and if data is present in the cache, then the CPU communicates with the fast cache instead of the slow main memory.

Since caches are hardware-controllable memories, their timing behaviour is hardly predictable, so in real-time systems *Scratchpad Memories (SPMs)* are usually used. SPMs are regular fast and energy-efficient small memories that can be used identically to the main memory. In contrast to caches, SPMs are software-controllable and can be efficiently analysed in terms of execution time and energy consumption.

Sections 2.2.1 and 2.2.2 give more details about the considered microprocessors, ARM Cortex-M0 and Infineon TriCore TC1797, respectively, and explain the difference between them.

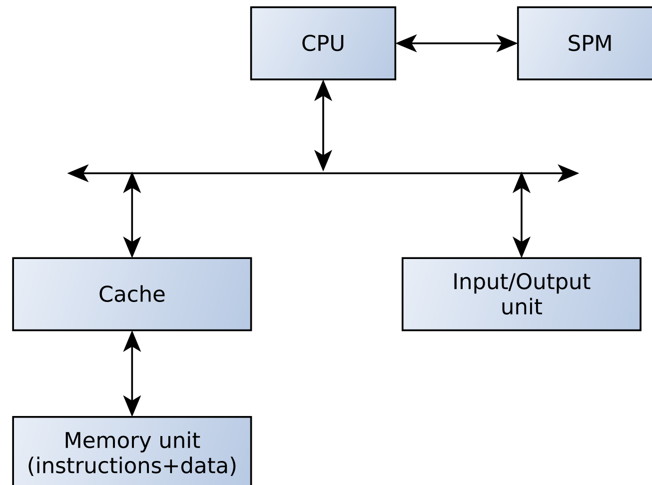


Figure 2.5: von Neumann architecture with a Scratchpad Memory (SPM).

2.2.1 ARM Cortex-M0

The ARM Cortex-M0 [ARM09a; ARM09b] is an energy-efficient ARM processor. It implements the ARMv6-M instruction set comprising memory access instructions, general data processing instructions, branch and control instructions, etc. The Cortex-M0 is a 32-bit microcontroller with a 3-stage pipeline: instruction fetch, instruction decode, and instruction execute. The 3-stage pipeline makes the processor energy-efficient by decreasing its power consumption. The Cortex-M0 is built on a von Neumann architecture. Figure 2.5 presents the scheme of a von Neumann architecture with an SPM: instructions and data share memory and bus, so an instruction fetch and a data operation (load or store) cannot be processed simultaneously. A cache is located between the CPU and main memory and connected to the CPU via a system bus, whereas the SPM is connected to the CPU via its own bus.

The Cortex-M0 lacks a floating-point unit, so floating-point operations are performed in software, which influences negatively performance.

2.2.2 Infineon TriCore TC1797

The Infineon TriCore TC1797 [Inf12; Inf14; Inf08] is a high-performance microcontroller. It supports Digital Signal Processing (DSP) operations to analyse efficiently complex real-world signals. The TriCore instruction set provides 32-bit and 16-bit instruction formats; the latter one reduces memory usage and power consumption. In contrast to the ARM Cortex-M0, the TriCore architecture supports a floating point unit and its instruction set contains some instructions for

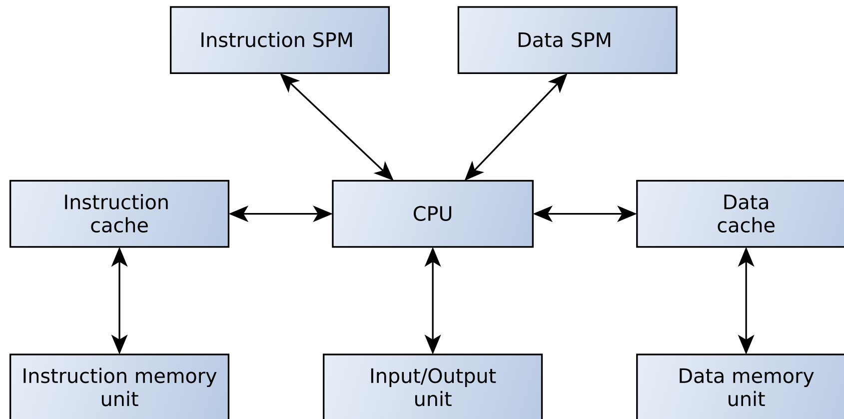


Figure 2.6: Harvard architecture with Scratchpad Memories (SPMs).

floating point arithmetic to improve performance. The TriCore architecture is a Harvard architecture shown in Figure 2.6. In contrast to von Neumann architectures, it has separate memories and buses for instructions and data, so the CPU can fetch an instruction and process data in parallel. Since Harvard architectures separate instruction and data memories, there are two caches located between the main memories and the CPU, and two separate SPMs for instructions and data.

The TriCore architecture features three pipelines, which improves the performance:

- the main pipeline offers four basic stages: instruction fetch, instruction decode, instruction execute, and write back;
- the load/store pipeline processes instructions that load or store data and perform address arithmetic;
- the loop pipeline processes loop instructions; it enables efficient implementation of loops.

3 WCET-Aware C Compiler

Chapter 1 describes a typical workflow to design an embedded system: a system designer sets up requirements (such as WCET, energy consumption, code size, etc.), models the system in a graphical tool, generates C code, optimizes and compiles it, and validates the requirements. If the system violates any requirement, the system designer either changes the model or tries to optimize the auto-generated C code and repeats the subsequent steps to validate the requirements again. Since the requirements of modern real-time systems often contradict each other, it becomes challenging to find optimal specifications for the systems.

If a compiler is aware of requirements, it can automatically optimize code concerning predefined objectives and return the possible values of the objectives to a system designer. Standard C compilers like `GCC` and `LLVM` cannot qualify the effect of their optimizations in terms of WCET and energy consumption. In contrast, WCET-Aware C Compiler (WCC) described by Falk and Lokuciejewski [FL10] is a C compiler framework written in C++ that gets WCET and energy consumption estimations from the static analysers aiT and EnergyAnalyser described in Section 2.1 and utilizes them to optimize C code with respect to the objectives.

WCC currently supports only single objective optimizations, i.e. it aggressively optimizes either the WCET or energy consumption of a program and ignores possible negative side-effects on other objectives. This thesis extends WCC with the notion of multiobjective optimizations and allows to find trade-offs between objectives at compile time.

WCC supports the following target microprocessors: TC1796 and TC1797 from the Infineon TriCore as well as ARM7TDMI and Coretex-M0 from the ARM family. Support for LEON3 processors is currently a work in progress. For scientific purposes, WCC supports an artificial ARM7TDMI multi-core system implemented by Kelter [Kel15].

In contrast to widely-used C compilers like `GCC` [Sta08] and `LLVM` [LA04], WCC provides specifications of the target architectures – e.g. memory layout – that are necessary for static WCET and energy consumption analyses. It allows WCC to optimize a program to be compiled concerning WCET and energy consumption at compile time.

We use WCC to demonstrate the approaches described in the thesis, so Section 3.1 describes the internal structure of WCC, Section 3.2 discusses the flow facts required for the WCET and energy consumption analyses, and Section 3.3

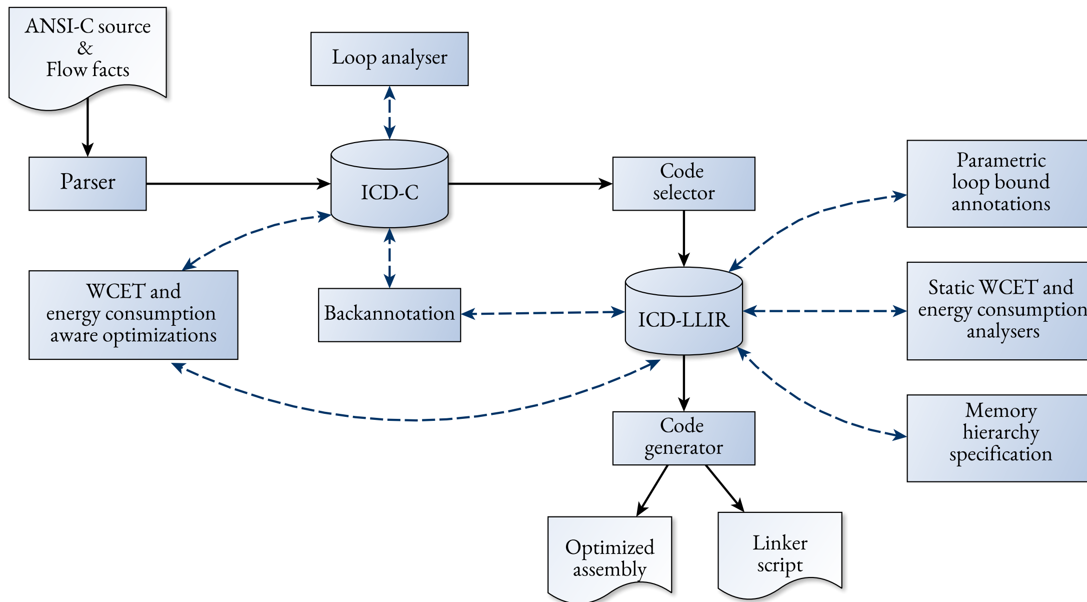


Figure 3.1: Simplified structure of WCC adapted from Lokuciejewski and Marwedel [LM11]. The solid lines indicate the typical flow of any modern compiler, whereas the dashed lines indicate additional WCC parts.

presents the integration of the WCET and energy consumption analyses within WCC.

3.1 Internal Structure

WCC consists of components that are typical for modern compilers, which are described by Muchnick [Muc98], and components that are specific for WCC as shown in Figure 3.1. WCC compiles a program as follows:

1. WCC takes as input ANSI C files;

2. the *parser* performs lexical, syntax, and semantic analyses and produces the high-level intermediate representation ICD-C developed in Informatik Center Dortmund [Dor09]. *ICD-C* represents the structures of the C source code – e.g. functions, loops, variables, etc. – and is unaware of the target architecture;

3. the *code selector* transforms ICD-C into the low-level intermediate representation ICD-LLIR developed in Informatik Center Dortmund [Dor05]. *ICD-LLIR* represents machine instructions and contains characteristics of the target architecture – e.g. the number of registers;

4. the *code generator* generates an assembly file and a linker script required to create the final binary for the target architecture.

WCC contains some additional parts which allow computing the WCET and energy consumption of a program at compile time:

1. the *loop analyser* coupled to ICD-C and the *parametric loop bound annotations* coupled to ICD-LLIR provide flow facts; more details follow in Section 3.2;
2. the *static WCET and energy consumption analysers* compute WCET and energy consumption; more details about the integration of the analysers into WCC are provided in Section 3.3;
3. the *memory hierarchy specification* describes the memory structures of the target architectures.

The static analysers and code size computation require characteristics of the target architecture, so WCET, energy consumption, and code size are computed at ICD-LLIR, the low-level intermediate representation. To enable WCET, energy consumption, and code size aware optimizations at ICD-C, the high-level intermediate representation, WCC features the *backannotation*. It translates WCET, code size, and energy consumption data from ICD-LLIR to ICD-C.

Similar to widely-used compilers like `GCC` and `LLVM`, WCC offers standard optimizations targeting to optimize the average case performance: high-level optimizations – e.g. function inlining or loop unrolling – and low-level optimizations – e.g. register allocation and instruction scheduling. WCC also features high- and low-level optimizations focusing on minimizing WCET and energy consumption; they are standard optimizations formulated in terms of WCET or energy consumption. The thesis extends standard compiler-based optimizations towards multiobjective optimizations with WCET, code size, and energy consumption as objectives and discusses a possible way to find trade-offs between the objectives.

More details on WCC are given by Falk and Lokuciejewski [FL10], Lokuciejewski and Marwedel [LM11], and Oehlert, Luppold, and Falk [OLF18].

3.2 Flow Facts

As mentioned in Section 2.1, the static WCET and energy consumption analyses require flow facts such as the number of loop iterations or recursion depth. Kirner defined flow facts in his Ph.D. thesis as follows:

Definition 3.1 ([Kir03])
Flow facts *are meta-information that provides hints about the set of possible control flow paths of a program.*

Listing 3.1: Exemplary annotations of static loop bounds.

```
1  int sum( int lim )
2  {
3    int res = 0;
4
5    _Pragma("loopbound min 5 max 50")
6    for( int i = 0; i < lim; ++i )
7    {
8      _Pragma("loopbound min 0 max 50")
9      for( int j = 0; j < i; ++j )
10       res += j;
11   }
12   return res;
13 }
14
15 int main()
16 {
17   int a = sum( 5 );
18   int b = sum( 50 );
19 }
```

Definition 3.2 ([Kir03])
Flow facts that exist due to the structure and semantics of a program code are called implicit flow facts*. Flow facts provided by a user are called* annotated flow facts*.*

Flow facts provided by a user are important for a precise WCET-analysis because

- implicit flow facts are usually only of limited quality;
- the control flow of a program depends on input data ignored while computing implicit flow facts.

WCC features a *loop analyser* introduced by Cordes, Falk, and Marwedel [CFM09]. It automatically derives an upper bound of the maximum number of loop iterations. The loop analyser might fail to derive the upper bound if the program to be compiled is too complex or the number of loop iterations depends on, e.g. pointer arithmetic. A programmer can manually describe flow facts directly in the source code using ANSI-C pragmas. Pragmas do not change the semantics of the C program, and compilers can ignore them if they are not interested in the annotated information.

Listing 3.2: Exemplary annotations of parametric loop bounds.

```
1  int sum( int lim )
2  {
3    int res = 0;
4
5    _Pragma("loopbound min user_reg max user_reg")
6    for( int i = 0; i < lim; ++i )
7    {
8      _Pragma("loopbound min 0 max user_reg")
9      for( int j = 0; j < i; ++j )
10       res += j;
11   }
12   return res;
13 }
14
15 int main()
16 {
17   _Pragma("user_reg = 5")
18   int a = sum( 5 );
19   _Pragma("user_reg = 50")
20   int b = sum( 50 );
21 }
```

The program from Listing 3.1 shows an example of loop bound annotations. It contains a function *sum* with two annotations of *static loop bounds* at Lines 5 and 8; the function *main* calls the function *sum* twice with different input parameters. In Section 2.1.3, we describe AbsInt's static analysers aiT and EnergyAnalyser. They estimate the WCET and energy consumption of a loop by computing the objectives for the basic blocks (see Section 2.1) belonging to the loop and multiplying them by the number of loop iterations. For the exemplary program, the static analysers overestimate the WCET and energy consumption since they assume the worst-case scenario: 50 iterations of the outer and inner loops for both calls of the function *sum* as we annotated in the pragmas.

AbsInt also supports *parametric loop bounds* to specify loop bounds depending on a context; parametric loop bounds allow tighter estimations of WCET and energy consumption. Listing 3.2 shows the code from Listing 3.1 with parametric loop bounds. For each call of *sum* in the function *main*, a *user register* is defined. A user register is a variable feasible only in pragmas (not in C code). We can assign different values to it depending on a context. In the function *sum*, we specified the loop bounds annotated in pragmas by using the user register. The lower and

upper bounds of the outer loop are equal to 5 instead of 50 for the first call and 50 for the second call; the lower bound of the inner loop remains unchanged, and the upper bound is equal to 5 for the first call and to 50 for the second call.

At the time of writing this thesis, WCC does not support the specification of parametric loop bounds in the source code as depicted in Listing 3.2. Ansari [Ans20] integrated annotations of parametric loop bounds into WCC at ICD-LLIR, so Listing 3.2 illustrates the concept of parametric loop bounds as it is exploited within WCC at ICD-LLIR. We utilize ICD-LLIR's parametric loop bounds in Chapter 5 to estimate tighter WCET.

3.3 Static Analyser Framework

WCC supports

- an internal static WCET analyser developed by Kelter [Kel15],
- an energy consumption analyser described by Roth, Luppold, and Falk [RLF18],
- AbsInt's tools aiT [Abs22] and EnergyAnalyser [Tea], which estimate WCET and energy consumption.

In order to evaluate the techniques presented in the upcoming chapters of the thesis, we use AbsInt's tools since AbsInt allows us to estimate both objectives, and moreover, aiT remains the state-of-the-art tool to estimate WCET.

To get the WCET and energy consumption of a program from AbsInt's tools, WCC

1. translates ICD-LLIR – including memory hierarchy and flow fact annotations – into an intermediate AbsInt format;
2. invokes AbsInt's timing and energy analysers to estimate the WCET and energy consumption of the program;
3. imports the computed WCET and energy consumption back and annotates ICD-LLIR with them.

4 Fundamentals

This chapter introduces main concepts required to follow approaches presented in the thesis. Section 4.1 describes principles of multiobjective optimization used in Chapters 6–9, Section 4.2 describes thr main concepts and components of evolutionary algorithms – methods widely used to solve multiobjective problems, and Section 4.3 explains machine learning concepts used in Chapters 7 and 9.

4.1 Multiobjective Optimization

Chapter 1 gives a general workflow to design an embedded system: a system designer defines requirements for the embedded system, creates a graphical model of the system, generates C code out of the model, and tries to optimize the code concerning the defined objectives. But the requirements of modern real-time systems (e.g. WCET, energy consumption, code size) contradict each other: improving one objective worsens the others. E.g. by minimizing the WCET or energy consumption of a program, we usually increase its code size and vice versa. Optimizations problems with contradicting objectives are called *multiobjective optimization problems*. In contrast to single-objective problems, multiobjective problems do not result in unique optimal objective values but a set of trade-offs between them.

Modern compilers can optimize a C code concerning a single objective to simplify system designer's tasks. In the thesis, we aim to extend a compiler with the concept of multiobjective optimization. Then, the compiler can automatically optimize the C code concerning several objectives and provide the system designer with a set of trade-offs between them. In the thesis, we consider multiobjective compiler-based optimizations for hard real-time systems with three objectives: WCET, energy consumption, and code size.

This section introduces the concepts of multiobjective optimization that are relevant to the thesis; we refer to Ehrgott [Ehr06] for more details on the topic. Section 4.1.1 formulates a general multiobjective problem. Section 4.1.2 introduces Pareto-optimality that defines a dominance relation for solutions of multiobjective problems. Section 4.1.3 describes quality indicators that we use in evaluations to assess the quality of solution sets.

4.1.1 Problem Formulation

If we denote by $X \subset \mathbb{R}^n$ the *decision* or *search space* of a multiobjective problem and by $Y \subset \mathbb{R}^m$ the *objective space* of the problem, an *objective function* $\vec{f}: X \to Y$ is defined as follows

$$\vec{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_m(\mathbf{x})) \quad . \tag{4.1}$$

E.g. for problems considered in the thesis, the decision space X depends on the considered compiler-based optimization, whereas the objective space Y is a 3-dimensional vector space representing WCET, energy consumption, and code size, i.e. $\vec{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x}))$, where f_1 represents WCET, f_2 – energy consumption, and f_3 – code size.

Without loss of generality, we formulate the multiobjective optimization problem without constraints as a minimization problem

$$\min_{\mathbf{x} \in X} \vec{f}(\mathbf{x}) \quad . \tag{4.2}$$

If any objective $f_i, i = \overline{1, m}$ [1] is to be maximized, then one should consider the function $-f_i$ which would have to be minimized.

4.1.2 Pareto-Optimality

In single-objective minimization problems, one compares the quality of two solutions by using a simple comparison between two numbers: a smaller value of the objective function is preferred. In a multiobjective problem, its objectives contradict each other, so one uses the concept of *Pareto-optimality* described by Ehrgott [Ehr06] to compare the quality of solutions.

Definition 4.1
For given $\mathbf{x}_1, \mathbf{x}_2 \in X$ *and* $\vec{f}$ *being defined in Equation* (4.1), $\mathbf{x}_1$ dominates $\mathbf{x}_2$ *or in symbols* $\mathbf{x}_1 \prec \mathbf{x}_2$, *if*

$$\forall i \in \{1, 2, \dots, m\} \quad f_i(\mathbf{x}_1) \leqslant f_i(\mathbf{x}_2) \tag{4.3}$$

and

$$\exists j \in \{1, 2, \dots, m\}: \quad f_j(\mathbf{x}_1) < f_j(\mathbf{x}_2) \quad . \tag{4.4}$$

A vector $\mathbf{x}_1$ dominates a vector $\mathbf{x}_2$, if for all objectives, the objective values at $\mathbf{x}_1$ are not worse than the objective values at $\mathbf{x}_2$ and there exists at least one objective such that its value at $\mathbf{x}_1$ is better than the value at $\mathbf{x}_2$. E.g. Figure 4.1 presents an

[1] The notation $i = \overline{1, m}$ is equivalent to $\forall i : i \in \{1, 2, \cdots, m\}$.

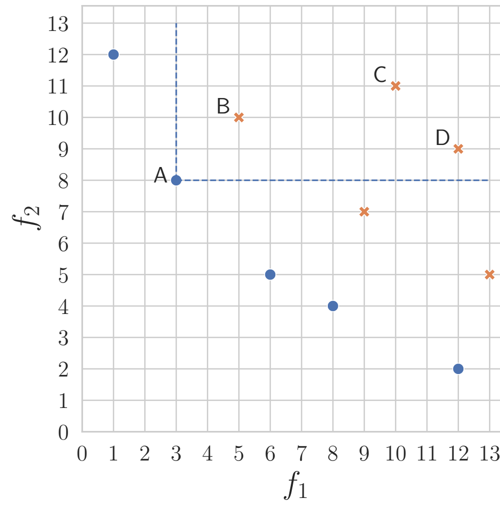


Figure 4.1: Exemplary Pareto optimal front (blue) for a minimization problem with two contradicting objectives f_1 and f_2. The search vectors of the blue points are not dominated by any other search vector, and the search vector corresponding to any orange point is dominated by at least one other search vector. E.g. the search vector of the blue point A dominates the search vectors of the orange points B, C, and D.

example for a minimization problem with two objectives f_1 and f_2. The point $\mathbf{x}_1$ with $f_1(x_1) = 3$ and $f_2(x_1) = 8$ (point A) dominates the points x_2, x_3, and x_4 with the following objective values: $f_1(x_2) = 5$ and $f_2(x_2) = 10$ (point B); $f_1(x_3) = 10$ and $f_2(x_3) = 11$ (point C); $f_1(x_4) = 12$ and $f_2(x_4) = 9$ (point D).

Definition 4.2
For given $\mathbf{x}_1, \mathbf{x}_2 \in X$ *and* $\vec{f}$ *being defined in Equation* (4.1), $\mathbf{x}_1$ weakly dominates $\mathbf{x}_2$ *or in symbols* $\mathbf{x}_1 \preceq \mathbf{x}_2$ *if*

$$\forall i \in \{1, 2, \ldots, m\} \quad f_i(\mathbf{x}_1) \leqslant f_i(\mathbf{x}_2) \ . \tag{4.5}$$

The dominance relation $\mathbf{x}_1 \prec \mathbf{x}_2$ is called *Pareto dominance* and $\mathbf{x}_1 \preceq \mathbf{x}_2$ is called *weak Pareto dominance*. Pareto dominance is stronger than weak Pareto dominance, i.e. any nondominated point is weakly nondominated but not vice versa. (A weakly nondominated point is not necessarily nondominated.)

Definition 4.3
A solution $\mathbf{x} \in X$ *is called* Pareto optimal, *if it is not dominated by any other solution.*

In the example from Figure 4.1, the point $\mathbf{x}_1$ with $f_1(x_1) = 3$ and $f_2(x_1) = 8$ (point A) is Pareto optimal.

Definition 4.4
Pareto optimal set $P_{set} \subset X$ *is a set of all Pareto optimal solutions:*

$$P_{set} = \{\mathbf{x} \in X : \mathbf{x} \textit{ is Pareto optimal}\} \ . \tag{4.6}$$

The goal of any multiobjective optimization is to find Pareto optimal front:

Definition 4.5
Pareto optimal front $PF \subset Y$ *is defined as follows:*

$$PF = \{\vec{f}(\mathbf{x}) : \mathbf{x} \in P_{set}\} \ . \tag{4.7}$$

Pareto optimal set is a subset of the decision space X, whereas Pareto optimal front is a subset of the objective space Y. Figure 4.1 shows in blue the Pareto optimal front for the exemplary minimization problem. The search vector of any orange point is dominated by at least one Pareto optimal solution.

For many real-world problems, the aim is to find approximation of Pareto optimal front for two reasons:

- the search space of a multiobjective problem is often large and determining a single Pareto optimal solution might be NP-hard;
- the proof of optimality is computationally demanding or even infeasible.

4.1.3 Quality Indicators

We evaluate the quality of approximated Pareto fronts using quality indicators. Most well-known quality indicators presented in the literature are appropriate to assess the quality of uniformly distributed Pareto fronts, e.g. hypervolume introduced by Zitzler and Thiele [ZT98b], ϵ-indicator introduced by Zitzler et al. [Zit+03], or R-indicator introduced by Hansen and Jaszkiewicz [HJ98]. In our evaluation, we usually observe Pareto fronts that are nonuniformly distributed, i.e. there are many points in one region of the objective space and some stand-alone points, so we consider other quality indicators, namely nondominance ratio and coverage.

When defining the quality indicators, we assume that an approximated Pareto front contains unique points from the objective space without duplicates. We denote by PF_A an approximated Pareto front returned by a solver, by PF the true Pareto front, and by $|\cdot|$ the size of a set.

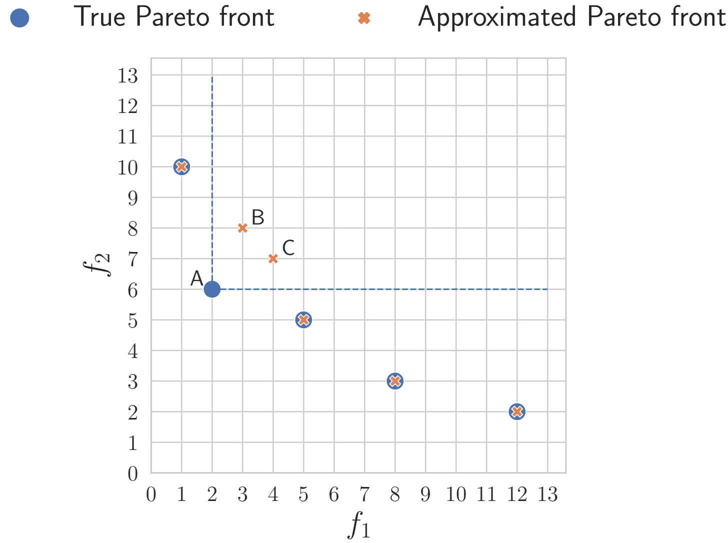


Figure 4.2: Exemplary true Pareto front (blue) and approximated Pareto front (orange) for a minimization problem with two contradicting objectives f_1 and f_2.

Definition 4.6
Nondominance ratio *represents the ratio of points in an approximated Pareto front that are also in the true Pareto front:*

$$NR(PF_A) = \frac{|PF_A \cap PF|}{|PF|} \quad . \tag{4.8}$$

Zitzler introduced a quality indicator called coverage in his Ph.D. thesis:

Definition 4.7 ([Zit99])
Coverage *represents the ratio of dominated points in an approximated Pareto front:*

$$C(PF_A) = 1 - \frac{|PF_A \cap PF|}{|PF_A|} \quad . \tag{4.9}$$

The values of the considered quality indicators are in the interval $[0, 1]$. The aim is to maximize nondominance ratio NR and to minimize coverage C. E.g. Figure 4.2 shows the true Pareto front (blue) and an approximated Pareto front (orange) for a minimization problem with two objectives f_1 and f_2. Computing the coverage and nondominance ratio for the approximated Pareto front, we notice:

- the approximated Pareto front intersects with the true Pareto front on four points, i.e. $|PF_A \cap PF| = 4$;

Algorithm 1 General framework of evolutionary algorithms.

1: **Input:** initial population, stopping criterion.
2: **Output:** approximated Pareto front.
3:
4: **while** stopping criterion is not reached **do**
5: *Generate new individuals:*
6: crossover;
7: mutation;
8: fitness evaluation.
9: *Select individuals for the next iteration:*
10: selection.
11: **return** approximated Pareto front.

- the true Pareto front consists of five points ($|PF| = 5$);
- the approximated Pareto front consists of six points ($|PF_A| = 6$);
- the point x_1 from the true Pareto set with $f_1(x_1) = 2$ and $f_2(x_1) = 6$ (point A) dominates two points x_2 and x_3 from the approximated Pareto set with $f_1(x_2) = 3$, $f_2(x_2) = 8$ (point B) and $f_1(x_3) = 4$, $f_2(x_3) = 7$ (point C), respectively.

As a result, $C(PF_A) = 1/3$, i.e. $1/3$ of the solutions from the approximated Pareto set is dominated by at least one solution from the true Pareto set; $NR(PF_A) = 4/5$, i.e. $4/5$ of the true Pareto front is covered by the approximated Pareto front.

4.2 Evolutionary Algorithms

Evolutionary algorithms are widely used to solve multiobjective problems. Behind all evolutionary algorithms, the idea of evolution is: if an environment has limited resources, individuals compete for them, and natural selection happens, i.e. the fittest individuals survive. In evolutionary algorithms, an objective (vector) function defines the fitness of individuals. The following definition mathematically introduces populations and individuals in the context of evolutionary algorithms:

Definition 4.8
If $X \subset \mathbb{R}^n$ is a decision space, $Y \subset \mathbb{R}^m$ is an objective space, and $\vec{f} : X \to Y$ is an objective function, a population *is a set of pairs $(\mathbf{x}, \mathbf{y})$ with $\mathbf{x} \in X$ and $\mathbf{y} = \vec{f}(\mathbf{x})$. The pairs $(\mathbf{x}, \mathbf{y})$ are called* individuals.

Algorithm 1 presents the general framework of evolutionary algorithms. An evolutionary algorithm starts with an initial population, which consists of random candidate solutions, and a specified stopping criterion to terminate the algorithm, e.g. a number of iterations.

At each iteration, the algorithm tries to introduce diversity and novelty to the population by utilizing crossover and mutation operators. To create a new individual, the *crossover operator* combines several individuals (called parents), whereas the *mutation operator* changes some characteristics of one individual. After evaluating the fitness of the new individuals, the algorithm tries to improve the quality of the individuals in the next generation by using a *selection operator*. Based on the values of the fitness function, it selects top-scoring individuals from the old and new individuals to survive in the next generation (iteration).

Evolutionary algorithms are stochastic methods due to the following reasons:

- crossover operators randomly choose individuals and their pieces to be combined;
- mutation operators randomly choose an individual and its piece to be changed as well as they randomly change the chosen piece;
- selection operators not deterministically choose the best individuals and weak individuals may survive in next generations.

Knowles, Thiele, and Zitzler [KTZ06] stated that for many real-world problems, the search spaces of the problems are large, so it might be NP-hard to determine a single Pareto optimal solution and the proof of optimality becomes infeasible or computationally demanding. Thus, any multiobjective optimizer aims to determine an approximated Pareto front that is as close as possible to the true Pareto front.

According to Eiben and Smith [ES15], the core components of evolutionary algorithms are:

- individual;
- population;
- initialization;
- parent selection;
- crossover;
- mutation;
- fitness function;
- survivor selection;
- termination condition.

Individual. Individuals consist of two parts as presented in Definition 4.8: a search vector and the corresponding objective vector. The search vector is an encoded candidate solution representing the problem context, e.g. the search

vector can be a bit string representing some Boolean statements. The objective vector represents criteria to be optimized, e.g. WCET or energy consumption.

Population. A population is the unit of evolution that holds individuals (not necessarily unique individuals). In evolution process, individuals remain unchanged, whereas the population evolves. Population size is predefined and remains unchanged during algorithm's execution to limit resources needed to compare individuals at the selection stage. The number of distinct search vectors or the number of distinct objective vectors usually represents the diversity of a population.

Initialization. One usually generates an initial population randomly or uses problem-specific heuristics to initialize it, e.g. one can

- initialize the population with known solutions obtained from other techniques;
- select top-scoring individuals from a large set of randomly generated solutions as Bramlette [Bra91] described;
- initialize the population with local optima obtained from a local search applied to random solutions.

Surry and Radcliffe [SR96] showed that heuristics may fail to provide a sufficient diversity of the initial population for evolution. Eiben and Smith [ES15] stated that the effort to generate the initial population other than random is unnecessary since evolutionary algorithms usually reach the population quality obtained from initialization heuristics in a few generations (iterations).

Parent selection. Crossover and mutation operators require individuals from the current population called parents to create new individuals. The parents are selected based on their quality, i.e. more fitted individuals become parents of the next generation. Parent selection is usually probabilistic: top-scoring individuals are more likely to become parents but low-quality individuals also have a small chance to be parents to avoid getting stuck in a local optimum.

Mutation. Mutation operators modify the search vectors of individuals to deliver new candidate solutions called children. Mutation operators are stochastic, i.e. mutations in children are random and representation dependent, i.e. different mutation operators must be defined for different search spaces. E.g. if a search vector is encoded as a bit string, a mutation operator flips each bit with a given probability but this mutation operator cannot be used with, e.g. integer-encoded vectors.

Crossover. The idea behind crossover is that by merging two individuals with desirable features, the operator may produce a new child with combined features. Crossover operators combine the search vectors of two parents to generate a new child in a stochastic way: the parts of the parents are randomly chosen and combined. Eiben and Smith [ES15] stated that crossover operators with more than two parents are possible, e.g. operators presented by Eiben, Raué, and Ruttkay [ERR94] or by Eiben and Bäck [EB97], but they are rarely used due to the absence of biological equivalents.

Fitness function. Fitness functions measure the quality (fitness) of individuals and form the basis for survivor selection operators. A fitness function is composed of an objective (vector) function and a quality measure, which is typically based on the concept of Pareto-optimality introduced in Section 4.1.2.

Survivor selection. Since population size usually remains constant during algorithm execution and crossover and mutation operators generate new individuals, evolutionary algorithms require a mechanism to select survivors for the next generation. Selection operators select individuals with high values of the fitness function and frequently prefer younger individuals.

Termination condition. According to Eiben and Smith [ES15], two types of termination conditions exist:

1. If an optimal fitness is known (probably from a known optimum of the objective function), evolutionary algorithms stop after finding a solution with fitness within a given precision;

2. Since any evolutionary algorithm is stochastic, it is not guaranteed that the algorithm reaches the optimum, i.e. it might never satisfy the first stopping condition. Also, the optimum is often unknown for real-world problems, so additional criteria to certainly stop the algorithms are used, e.g.
   - the maximum number of generations;
   - the number of generations without improvement of fitness function;
   - the number of fitness evaluations;
   - the drop of population diversity under a given value;
   - the maximum CPU time, etc.

We use evolutionary algorithms to solve a multiobjective problem in Chapters 6–9. Section 6.2 describes in detail evolutionary algorithms used in our evaluations.

4.3 Machine Learning

Machine learning is a computational method that aims to improve performance or to predict objective values at a new search vector by using the past available information collected into a training set. E.g. James et al. [Jam+21] considered a machine learning task of predicting the wage of an employee based on the employee's age and education.

The accuracy of predictions heavily depends on the quality and size of the training data: more samples provide more information about the data, but noisy data might mislead a learner. E.g. knowing the age, education, and wage of many employees allows predicting accurately the wage of a new employee based on the age and education but if the wages of the known employees are incorrectly labelled, the wage of the new employee will be predicted wrongly.

Supervised and unsupervised learning are two main classes of learning tasks. Supervised learning deals with data represented by pairs of variables and their labels. A learner aims to predict the label of a new variable based on information received from seen samples. E.g. given a set of houses described by the number of rooms, the number of floors, location, etc., and the corresponding prices, the task is to predict the price of a new house.

Unsupervised learning deals with unlabelled data. E.g. given a set of products' features like colour, size, and shape, the task is to identify the features that influence the prices of the products.

As described by Mohri [Moh18], some standard types of machine learning problems exist:

- *Classification*: This task deals with categorical or qualitative data. A classifier assigns a class or category to a new item given training data. E.g. given the air temperature of the previous 15 days, the task is to predict whether the temperature will increase or decrease tomorrow. In such problems, the number of categories is often limited to a few hundred, as mentioned by Mohri [Moh18].

- *Regression*: The task deals with real-valued labels. A regressor predicts a real value for a new item given training data. E.g. given the air temperature of the previous 15 days, the task is to predict the temperature tomorrow. In contrast to classification tasks, in regression, the closeness of the predicted values to the true values represents the quality of the predictions.

- *Ranking*: The task is to assign ranks to items. E.g. given a list of web pages, the task is to order the web pages according to their relevance to a search query.

- *Clustering*: The task is to partition a set of items into homogeneous clusters (subsets). E.g. clustering helps to identify a target group for selling a product by grouping people by their traits and purchases.
- *Dimensionality reduction*: The task is to transform data into a lower-dimensional representation. E.g. a bank wants to predict the capability of an applicant to repay a loan. The bank collects much information about the applicant but only some of it is relevant to the credit risk score (average monthly income and credit history are probably relevant, whereas gender is not). In this case, unsupervised learning identifies the features that influence the credit risk score.

Classification, regression, and ranking are the examples of supervised learning, whereas clustering and dimensionality reduction belong to unsupervised learning.

In Chapter 7, we apply supervised learning techniques, namely classification, to predict WCET and energy consumption at compile time. Section 7.2 presents mathematical formulation of supervised learning and describes learning algorithms considered in the thesis.

5 Code Compression for Hard Real-Time Systems

Many embedded systems have a limited memory space, so it is crucial to optimize embedded applications concerning code size. Compression techniques allow distribution of compact applications. They often extract the entire application into memory at execution time for PC systems. For embedded systems, decompression of the entire program may lead to memory overflow, so we propose compressing chunks of an application that are independently decompressed at execution time.

Compression approaches differ in implementation techniques: software and hardware methods. Hardware compression is usually faster and achieves a better compression rate than software compression, but it requires expensive specialized hardware. Software compression is usually a more appropriate option because it is cheap and easily configurable, i.e. one can specify how data is compressed, whereas, in the case of hardware compression, the manufacturer integrates compression into the hardware without possibility to be changed. So we focus on a software-based approach where compression happens at compile time, whereas decompression takes place at runtime. The runtime decompression influences the performance of an embedded system, so in the proposed approach, the compiler pays attention to a WCET constraint while selecting chunks for compression:

- if we compress many small chunks, we might violate the WCET constraint because of the runtime decompression;
- if we compress a few large chunks, we might cause memory overflow.

We reduce the program size of an application – the sum of its code size and data size – with as little WCET penalty as possible. The compiler finds a trade-off between the decreasing program size and increasing WCET of the final program. The proposed compression guarantees that WCET constraints are satisfied. To the best of our knowledge, it is the first compiler-based compression that can be safely used for hard real-time systems. Moreover, it can be easily extended to guarantee that other constraints, e.g. energy consumption, are also satisfied.

Chapter 1 describes that a widely used method to solve a multiobjective problem is to reformulate it as a single-objective problem with constraints. This approach is preferable if the aim is to optimize one objective as much as possible

and keep the others below certain limits, since it results in a unique objective optimum, and it is usually easier to solve a single-objective problem than a multiobjective one. The main goal of any compression technique is to decrease program size as much as possible, so we formulate the multiobjective compression considered in this thesis as a single-objective optimization problem with program size as the objective and other objectives like WCET or energy consumption as constraints.

The proposed compression was presented at the International Workshop on Software and Compilers for Embedded Systems (SCOPES) in Sankt Goar, Germany 2019 [MLF19].

The chapter is organized as follows: Section 5.1 discusses related work, Section 5.2 introduces the proposed compression/decompression compiler framework, Section 5.3 presents evaluation results.

5.1 Related Work

Many compression/decompression techniques were proposed to save the memory storage of embedded systems. There are two main classes of compression techniques: hardware-based and software-based approaches. Hardware-based approaches achieve better timing performance, whereas software-based techniques are less expensive since improving hardware is usually a cost-demanding task.

Hardware-based compression schemes. A fundamental hardware-based compression technique is to use shorter instructions. Some processors support 16-bit and 32-bit instructions: 16-bit instructions decrease code size, whereas 32-bit instructions improve performance. The best known dual-width instruction set is ARM Thumb described by Goudge and Segars [GS96]. In the decode stage, 16-bit instructions are translated into 32-bit code. A binary code with 16-bit Thumb instructions is usually 30 % smaller than the regular code but requires a longer execution time since expressiveness of 16-bit instructions is more limited than in the case of 32-bit instructions. To trade off code size reduction and performance degradation, 32-bit instructions are used to compile the most frequently executed code and 16-bit instructions are used for less frequently executed code.

Wolfe and Chanin [WC92] developed the Compressed Code RISC Processor which is the first hardware decompression for a RISC processor. They considered a line of instruction cache (the blocks of the main memory transferred to a memory cache) as a compression unit and utilized Huffman coding [Huf52] to compress code. The authors used cache misses to trigger decompression: for every cache miss, the instructions are fetched from the main memory, decom-

pressed, and put into a cache line. They achieved a significant degree of compression with slight performance degradation.

Later, researchers – e.g. Debray and Evans [DE02] or Xie, Wolf, and Lekatsas [XWL03] – have shown that hardware-based compression schemes can reduce not only code size but also energy consumption and improve performance by positioning a decompression engine between the processor and a cache. The technique allows storing compressed instructions in the cache to increase its capacity. In such techniques, one often uses a small instruction dictionary; it stores short indices for the most frequently appearing instructions, then the original instructions are replaced by their indices. Usually, indices are 8-bit long for 256-entries dictionaries, whereas the instructions of modern processors are 32-bit long. To identify the most frequently executed instructions which are stored in a dictionary, Lekatsas, Henkel, and Wolf [LHW01] and Lekatsas, Henkel, and Jakkula [LHJ02] statically counted instructions' occurrences, whereas Benini, Macii, and Nannarelli [BMN01] used instructions' dynamic profiling information.

Netto et al. [Net+03] combined both static and dynamic profiling since static counts allow compressing the most frequently appearing instructions and achieving the best compression ratio (the final compressed size over the original size), whereas dynamic profiling information allows identifying the most frequently fetched instructions and improving performance and energy consumption. The authors used a dictionary with 256 entries and filled it by adding the most frequently fetched and the most frequently appearing instructions one by one avoiding duplicates in the dictionary. They preferred *post-cache code compression*, i.e. code is compressed in the main memory and cache, over *pre-cache code compression*, i.e. code is compressed in the main memory but not in the cache, for two reasons:

1. post-cache code compression saves more space in the main memory and cache;
2. in the case of post-cache code compression, buses between the main memory and cache and between the cache and processor require fewer bit toggles to transfer data and, as a result, less energy.

For considered benchmarks, the authors obtained average compression ratio of 73 % and decreased bus accesses to the cache by 31 % for the Leon processor. (Fewer bus accesses improve performance and minimize energy consumption.)

Ozaktas et al. [Oza+09] studied the impact of hardware-based code compression on estimated WCET. To compress code, they used a dictionary with the most frequently executed and the most frequently appearing instructions and replaced two or three successive instructions present in the dictionary with one 32-bit instruction. In the paper, the authors considered post-cache code compression and

placed decompression in the processor pipeline between the fetch and decode stages, so decompression time penalty was hidden by the pipeline execution and experiments showed code size reduction together with WCET improvement.

A drawback of hardware-based approaches is that they employ expensive specialized hardware, so less cost-demanding alternatives are software-based compression techniques.

Software-based compression schemes. Software-based approaches are more flexible and much cheaper than hardware-based ones, but they must pay attention that a compressed code does not cause a memory leak, whereas timing and energy constraints are satisfied for two reasons:

1. free space in data memory decreases because of compressed fragments stored in the code as data objects;

2. execution time and energy consumption increase since a decompression engine is a software library invoked at runtime to decompress the compressed code into a buffer.

Software-based compression schemes differ in the following main aspects:

1. a method to select code fragments for compression;

2. the granularity of code fragments selected for compression, i.e. basic blocks, sets of basic blocks, functions, etc.;

3. a method to decompress code fragments before their execution.

Some software-based compression techniques utilize a cache management instruction to write decompressed code into an instruction cache considered as a buffer, e.g. Kirovski, Kin, and Mangione-Smith [KKMS99]. In the case of a cache miss, the compressed instruction is read from the main memory, decompressed, and put into the cache. These approaches make two assumptions about the target processor:

1. an instruction cache miss raises an exception – it allows to invoke decompression for a cache miss;

2. the instruction set contains an instruction to modify the contents of the instruction cache – it allows to put decompressed code into the cache.

Lefurgy, Piccininni, and Mudge [LPM00] presented a compression technique relying on such software-managed instruction caches. Their compression works on the granularity of cache lines. The authors compared two compression/decompression methods: a dictionary-based compression and IBM's CodePack. The

dictionary-based approach stores each unique 32-bit instruction in a dictionary and replaces it with its 16-bit index. CodePack [IBM98] compresses 16 instructions into a group of unaligned variable-length codewords. The authors showed that the dictionary-based approach is faster, whereas CodePack improves compression ratio by 5 % to 25 %.

Pinter and Waldman [PW07] presented a software-based code compression that takes into account overheads in terms of runtime and memory consumption. To identify compression regions defined by sets of basic blocks, the authors used profiling information (e.g. the execution frequency of a region). They utilized BICOM BIjective COMpressor developed by Timmermans [Tim00] to compress selected regions. The compressed regions are decompressed into a buffer at runtime and are executed from it. The authors embedded the compressed regions, the decompression library, and library calls in the code to enable runtime decompression. The method requires a training set of benchmarks to estimate the overhead of runtime decompression. The technique cannot optimize a real-time application because it does not guarantee that its timing constraints are satisfied. On SPEC CPU2000 and MediaBench suites [LPMS97], the authors achieved average reduction of 18.5 % in code size, average overhead of 3.8 % in memory consumption, and average overhead of 7.8 % in runtime.

Knuth [Knu71] described the "80-20 rule" which states that a program spends most of its execution time in a small portion of the code. Debray and Evans [DE02] exploited the rule to compress infrequently executed code to reduce code size and avoid a significant runtime penalty due to runtime decompression. The authors performed the depth-first search – described in Even [Eve15] – in a control flow graph to select sets of basic blocks from a single function for compression. Witten, Moffat, and Bell [WMB99] mentioned that canonical Huffman encoding uses little memory and permits fast decompression, so Debray and Evans used it to compress and decompress selected regions. The authors replaced a compressed code with a short sequence of instructions to call a decompression library. Experiments showed average code size reduction of 16.3 % and average execution time slowdown of 13.5 %.

Summary. Although hardware-based approaches can decrease the code size and improve the performance and energy consumption of an embedded system, they are not a popular choice for many real-world problems since they require expensive hardware modifications. In contrast, software-based approaches are much cheaper and more flexible in the sense that they allow changing easily a compression algorithm without any hardware changes. Since software-based approaches invoke software decompression at runtime, in the case of real-time systems, it is important to pay attention to additional constraints like performance and energy consumption. Hard real-time systems must, in addition, satisfy its WCET constraint, but, to the best of our knowledge, none of software-based

approaches presented in the literature can guarantee that WCET remains within an acceptable range despite timing overheads introduced by software runtime decompression. In this thesis, we propose a novel compiler-based compression technique that guarantees that WCET constraints are satisfied.

5.2 WCET-Aware Compiler-Based Code Compression

As motivated in the previous section, hard real-time systems need a compression method that can guarantee that the WCET of a program is always below a predefined limit. We aimed to develop a new compression technique that minimizes program size and has two main characteristics:

1. it guarantees that WCET constraints are satisfied;

2. it is flexible and can be easily extended with additional constraints, e.g. energy constraint, if required.

Since the new compression method must be flexible, we follow the methodology of software-based approaches: a decompressor decompresses compressed code into a buffer at runtime.

Decompression takes place at runtime and increases WCET, so it is crucial to choose an appropriate compression algorithm. Any compression technique takes a string of bytes as input and encodes it by using fewer bytes. We utilized an *asymmetric compression algorithm* due to its main characteristics:

- compression achieves a high compression ratio;

- decompression is fast compared to the compression.

The fast decompression saves us from a dramatic increase in runtime, whereas the compression reduces memory usage.

We use the Lempel-Ziv 77 compression algorithm (LZ) developed by Ziv and Lempel [ZL77], which continues to arouse the interest of researchers like, e.g. Policriti and Prezza [PP17], Puglisi [Pug16], or Kosolobov et al. [Kos+20]. The algorithm is used in well-known compressors like, e.g. p7zip, zip, rar. LZ is a lossless compression algorithm, i.e. it perfectly reconstructs the original data from the compressed data. LZ compression follows the next steps:

1. it parses a source string into phrases; each phrase is the shortest phrase not seen earlier;

2. it encodes each new phrase by using the index of a previously encountered phrase and a new character at the end of the new phrase.

Example 5.1
Given the following bit string to be compressed:

$$110111000110111 \tag{5.1}$$

The LZ algorithm parses the string into the following phrases:

$$1, 10, 11, 100, 0, 110, 111 \tag{5.2}$$

Each phrase is a previously encountered phrase plus a new character at the end.

Assuming that an empty string has index 0, LZ encodes the first phrase 1 *as* $(0, 1)$*, where* 0 *corresponds to the index of the previously encountered empty string and* 1 *corresponds to the new character at the end of the first phrase. The encoding for the second phrase* 10 *is* $(1, 0)$*, where* 1 *corresponds to the index of the previously encountered phrase* 1 *and* 0 *corresponds to the new character* 0*. Continuing in the same manner, the LZ encode for the entire string is*

$$(0, 1), (1, 0), (1, 1), (2, 0), (0, 0), (3, 0), (3, 1)\ . \tag{5.3}$$

In our evaluation, we use the `FastLZ` compression library implemented by Hidayat [Hid07]. It represents the LZ algorithm focusing on fast compression and decompression. Hidayat implemented the library in the C programming language and optimized it to speed up compression and decompression. (The decompression is still faster than the compression since the LZ algorithm is asymmetric.)

To the best of our knowledge, none of the compression techniques discussed in Section 5.1 has been considered as a compiler-based optimization problem but many software-based approaches have great potential to be integrated into a compiler. E.g. Pinter and Waldman [PW07] presented compression that can be integrated into a compiler if the compiler can provide the profiling information of input code. We integrated our compression technique into the WCET-Aware C Compiler (WCC). WCC automatically identifies compression regions, compresses the selected code, and makes necessary code modifications to enable runtime decompression. WCC has two intermediate representations of code: the high-level intermediate representation (ICD-C) and the low-level intermediate representation (ICD-LLIR) described in Section 3.1. We integrated the proposed compression technique at the low-level representation since it supports description of the memory hierarchy for target architectures, which is necessary to extract data from a binary executable for compression.

Different levels of abstraction of code can be considered to define chunks for compression: functions, loops, basic blocks, etc. In our approach, we consider functions as compression candidates, since if the size in bytes of a chunk is too small, the size of corresponding compressed data might be larger than the

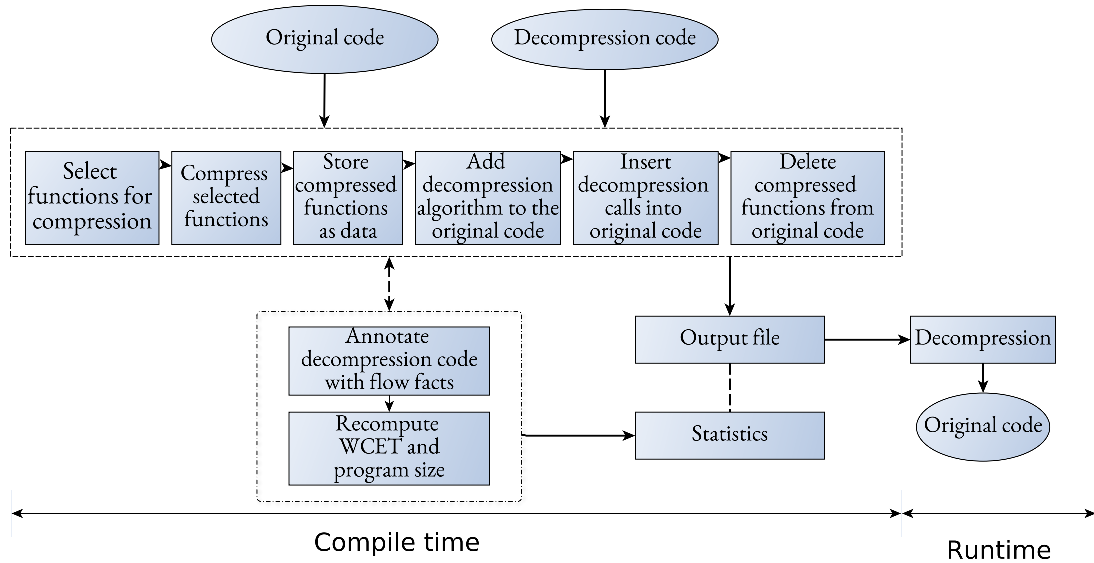


Figure 5.1: Workflow of compile-time code compression and runtime decompression. We pass an original code to be compiled and the code of decompression to the compiler which compresses selected functions taking into account the WCET and program size of the resulting code. The decompressor restores the original functions at runtime.

original size, i.e. considering basic blocks or loops as chunks for compression might be meaningless.

We made three main **assumptions**:

1. We focus on code compression without considering data compression since data compression differs from code compression as noticed by Bonny [Bon10]. The main difference between code compression and data compression is the size of code and data to be compressed. Code is split into smaller blocks that are independently compressed and can be decompressed before being executed, whereas data is usually compressed as one larger block. Since data may be too large, e.g. video stream, and do not fit into memory, compressing larger blocks allows improving compression ratio by utilizing repetitions in the data. It raises the question of how to decompress compressed data at runtime before utilizing them, if the decompressed data do not fit into the memory. Since data compression is another large scientific topic, it is left for future work.

2. We consider any function of a code to be compiled as a compression candidate except the entry point of the program, so that at least one function remains in the code to call decompression routine;

3. We decompress all compressed functions at the beginning of program's execution. The assumption restricts the usage of a buffer because all decompressed functions must fit into it. If some functions have disjoint lifetimes during execution, they could share a memory slot within the buffer, so the compiler can compress more functions. To relax this assumption, one requires a liveness analysis of functions to decompress a function only when it is called and missed in the buffer. This assumption is strict and must be relaxed in future work, but we show that the proposed compression technique can achieve a high compression ratio even with this assumption.

We consider two objectives in our approach: WCET and program size. We consider program size instead of code size as an objective because memory space consists of two sections: a code section and a data section. The code section contains executable code, while the data section contains data, e.g. global variables. The compiler stores compressed functions as data occupying more space in the data section and removes them from the code releasing more space in the code section.

Remark 5.1
In the thesis, we focus on three objectives: WCET, code size, and energy consumption. Unfortunately, the compression technique is aware only of WCET and program size (code size is included in program size), since it was developed for WCC targeting the TriCore TC1797 processor and WCC lacks an energy analyser for this processor. But in this chapter, a multiobjective problem is formulated as a single-objective ILP problem by considering program size as an objective and other design criteria as constraints, so the proposed ILP model can be easily extended with additional constraints, if it is required and a compiler supports necessary analysers.

Figure 5.1 shows the workflow of the proposed code compression. The WCC compiler takes an original code and a decompression code as input. It selects functions for compression using an ILP model described in Section 5.2.1, compresses the selected functions as described in Section 5.2.2, extends the original code with the decompression code to enable runtime decompression as described in Section 5.2.3, and deletes the compressed functions from the original code. The compiler output includes the WCET and program size statistics of an output file. To compute the final WCET, the decompression code is annotated with flow facts, which are crucial for WCET analysis, as described in Section 5.2.4.

5.2.1 Selection Model

This section describes steps to select functions for compression. The selection process consists of two parts:

1. a prephase: the compiler selects functions whose size in bytes is decreased after compression. If the size of a function is small, compression might increase the size;

2. an ILP selection model: the compiler selects functions – from those selected at the prephase – for compression by solving an ILP problem.

Prephase

To select functions for compression at compile time, the compiler collects all functions of an original program with the size of a compressed function being less than the size of the original function.

Definition 5.1
The compression ratio R *of a function is defined by*

$$R = \frac{size(compressed\ function)}{size(original\ function)} \quad . \tag{5.4}$$

We call *compression candidates* all functions with $R < 1$.

ILP Selection Model

After the prephase, a set $U = \{f_1, f_2, .., f_N\}$ contains all compression candidates. We define binary *decision variables* $x_i \in \{0, 1\}$ that correspond to the functions f_i, $i = \overline{1, N}$ [1] as follows:

$$x_i = \begin{cases} 1, & f_i \text{ is being compressed,} \\ 0, & \text{otherwise.} \end{cases} \tag{5.5}$$

We deal with a multiobjective optimization problem considering WCET and program size as objectives, but the primary aim of compression is to minimize program size, so we forbid solutions that lead to an increased program size. The aim is to reduce program size as much as possible without violating WCET constraints. In this case, the problem becomes single-objective, and we can solve it by using ILP.

To model the overall WCET of a final program, we follow an approach introduced by Falk and Kleinsorge [FK09]. We denote by K the total number of functions in the program. We also distinguish between the execution time of the main memory and a buffer because one may use a buffer intended to speed up execution, e.g. a Scratchpad Memory (SPM) described in Section 2.2 is a valid choice for the buffer.

[1] The notation $i = \overline{1, N}$ is equivalent to $\forall i : i \in \{1, 2, \cdots, N\}$.

If a function f_k is a compression candidate, i.e. $f_k \in U$, its WCET differs depending on whether

- the function is executed from the main memory, i.e. the function is not selected for final compression and remains in the main memory, or
- the function is executed from the buffer, i.e. the function is selected for final compression, it is compressed at compile time, and it is decompressed to the buffer at runtime.

Then, the WCET of f_k is modelled as follows:

$$WCET_k^{base} = (1 - x_k) \cdot WCET_k^{main} + x_k \cdot WCET_k^{buffer}, \quad (5.6)$$

where $WCET_k^{main}$ is the overall WCET of f_k executed from the main memory and $WCET_k^{buffer}$ is the overall WCET of f_k executed from the buffer.

If a function f_k is not a compression candidate, i.e. $f_k \notin U$, it is executed from the main memory and its WCET is equal to $WCET_k^{main}$.

Finally, we define the WCET of a function $f_k, k = \overline{1, K}$ as follows:

$$WCET_k^{base} = \begin{cases} (1 - x_k) \cdot WCET_k^{main} + x_k \cdot WCET_k^{buffer}, & f_k \in U, \\ WCET_k^{main}, & f_k \notin U. \end{cases} \quad (5.7)$$

If a function f_k calls a function f_j, function call penalty must be modelled, since it depends on whether the functions f_k and f_j are placed in the same or different memories. If the function f_k is placed in the main memory and the called function f_j is placed in the buffer, a jump instruction ensures that f_j is reached from the f_k's basic block that calls f_j. According to Falk and Kleinsorge [FK09], a jump usually requires several machine instructions for the following reasons:

- the target address of a jump instruction is often computed and stored in an address register and a register-indirect jump instruction is used;
- the distance between the address spaces of the main memory and buffer may be large;
- the displacement which can be encoded as the target of a jump instruction is limited.

Similar to Falk and Kleinsorge [FK09], we model the jumping overhead for f_k and f_j as follows:

$$WCET_{kj}^{call} = \begin{cases} (x_k \oplus x_j) \cdot P_{high} + (1 - (x_k \oplus x_j)) \cdot P_{low}, & f_k, f_j \in U, \quad (5.8a) \\ x_j \cdot P_{high} + (1 - x_j) \cdot P_{low}, & f_k \notin U, f_j \in U, \quad (5.8b) \\ x_k \cdot P_{high} + (1 - x_k) \cdot P_{low}, & f_k \in U, f_j \notin U. \quad (5.8c) \end{cases}$$

If both functions f_k and f_j are compression candidates, Equation (5.8a) defines the function call penalty, where

- the operator $\oplus$ is the Boolean XOR,
- P_{high} is a constant representing a high penalty when the functions are placed in different memories and a large jumping overhead occurs;
- P_{low} is a constant representing a low penalty when the functions are placed in the same memory.

If the function f_k is not a compression candidate, i.e. it is placed in the main memory, and the called function f_j is a compression candidate, then Equation (5.8b) models the call penalty which depends on whether

- the function f_j is selected for compression by the ILP model and is decompressed to the buffer at runtime or
- the function f_j is not selected for compression by the ILP model and it remains in the main memory.

Similarly, Equation (5.8c) models the case when the function f_k is a compression candidate and the called function f_j is not a compression candidate.

Remark 5.2
The nonlinear operator XOR in Equation (5.8a) is expressed by using linear inequalities as follows:

$$z = x_1 \oplus x_2 \iff \begin{cases} z \leqslant x_1 + x_2, \\ z \geqslant x_1 - x_2, \\ z \geqslant x_2 - x_1, \\ z \leqslant 2 - x_1 - x_2. \end{cases} \tag{5.9}$$

Finally, for a function $f_k, k = \overline{1, K}$, its WCET is defined as follows:

$$\begin{aligned} WCET_k = WCET_k^{base} + \sum_{j: f_k \text{ calls } f_j} WCET_{kj}^{call} \cdot C_{kj}^{call} \\ + \begin{cases} WCET_k^{decomp} \cdot x_k, & f_k \in U, \\ 0, & f_k \notin U, \end{cases} \end{aligned} \tag{5.10}$$

where

- $WCET_k^{base}$ is defined in Equation (5.7),
- $WCET_{kj}^{call}$ is defined in Equations (5.8a)–(5.8c),

- C_{kj}^{call} is the number of calls of the function f_j inside the function f_k,
- $WCET_k^{decomp}$ is the WCET of the LZ decompression routine when decompressing the function f_k, if it is compressed at compile time and must be decompressed at runtime.

Then, the overall WCET of the final program is

$$\sum_{k=1}^{K} WCET_k \ . \tag{5.11}$$

We call ΔPS_i the change in program size after compressing a function f_i, $i = \overline{1, N}$:

$$\Delta PS_i = CS_i^{decomp} - (CS_i - DS_i) \,, \tag{5.12}$$

where

- CS_i^{decomp} is code size increase due to a call of decompression code to decompress the function f_i;
- CS_i is the code size of the original function f_i;
- DS_i is the data size of the compressed data corresponding to the function f_i.

We define ΔPS_i such that it usually results in a negative value and has to be minimized to achieve better compression of a final executable file.

We distinguish between a call and the body of decompression routine. Both affect WCET and program size. For each compressed function, we assume that $WCET_k^{decomp}$ from Equation (5.10) represents the WCET of the call and the body of the decompression routine when decompressing the function f_k, whereas CS_i^{decomp} from Equation (5.12) represents the code size of the call since the code size of the body is constant and affects the code size of the final program only once. We reflect the code size of the body and the data size of the decompression routine in the ILP constraint (5.15) described later.

Objective function

We minimize the following *objective function*:

$$G(x) = \sum_{i=1}^{N} \Delta PS_i \cdot x_i, \tag{5.13}$$

subject to the following *constraints*:

1. The final WCET is less than or equal to a predefined value $WCET_{limit}$ to satisfy timing constraints despite runtime decompression of compressed functions:

$$\sum_{k=1}^{K} WCET_k \leqslant WCET_{limit} \ . \tag{5.14}$$

2. Program size never increases even though decompression code is inserted in the final binary file:

$$PS_{decomp} \cdot \max_{1 \leqslant i \leqslant N}(x_i) + \sum_{i=1}^{N} \Delta PS_i \cdot x_i \leqslant 0, \tag{5.15}$$

   where PS_{decomp} is the program size of the LZ decompression routine.

   Remark 5.3
   To rewrite the nonlinear operator max in Equation (5.15) *as a linear operator, we define an artificial variable* $y \in \{0, 1\}$*. Equation* (5.15) *is equivalent to the following system of inequalities:*

$$\begin{cases} PS_{decomp} \cdot y + \sum\limits_{i=1}^{N} \Delta PS_i \cdot x_i \leqslant 0, \\ y \geqslant x_i, \qquad i = \overline{1, N}. \end{cases} \tag{5.16}$$

3. The total size of compressed data is less than or equal to a predefined value DS_{limit} due to a limited data storage:

$$\sum_{i=1}^{N} DS_i \cdot x_i \leqslant DS_{limit}, \tag{5.17}$$

   where DS_i is the data size of the compressed data for the function f_i.

4. The total code size of decompressed functions is less than or equal to a predefined value $BUFF_{limit}$ because of a limited buffer size. We assume that all functions are decompressed right upfront executing the final program, so all decompressed functions must fit to the buffer:

$$\sum_{i=1}^{N} CS_i \cdot x_i \leqslant BUFF_{limit} \ . \tag{5.18}$$

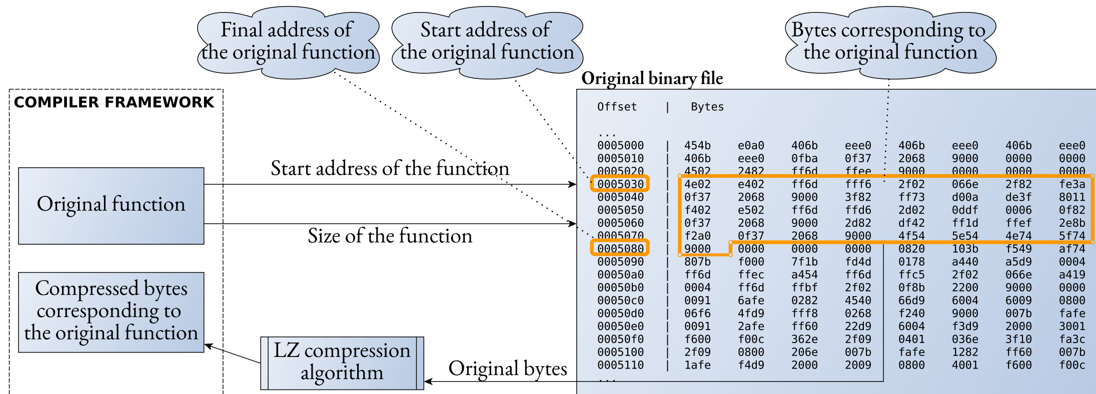


Figure 5.2: Compression of a function at compile time. The compiler extracts bytes corresponding to the function from an original binary file, passes the bytes to the LZ compression algorithm and gets compressed bytes.

5.2.2 Compression

After selecting functions for compression as described in the previous section, the compiler compresses the functions as shown in Figure 5.2. To provide a compression algorithm with input bytes corresponding to a function to be compressed, the compiler must:

1. compile the original code and produce a binary file;

   Remark 5.4
   Since compressed functions are decompressed to a buffer, we must guarantee that the program flow is not broken, e.g. while calling a decompressed function, the call instruction must refer to the function's buffer address but not to the original memory address. For this reason, before generating a binary file, functions selected for compression are moved to the buffer and jumps to the buffer are corrected.

2. identify bytes corresponding to the function in the original binary file;

3. pass the extracted bytes to the compressor and get compressed bytes.

The compiler uses the start address and the code size of the function to identify the bytes of the original function. This information is known within WCC thanks to its memory hierarchy infrastructure supported at the low-level representation (ICD-LLIR, see Figure 3.1). The compiler stores the compressed bytes as a data object in the final binary file, and the decompressor uses them to restore the original function at execution time. We compress functions at compile time, and com-

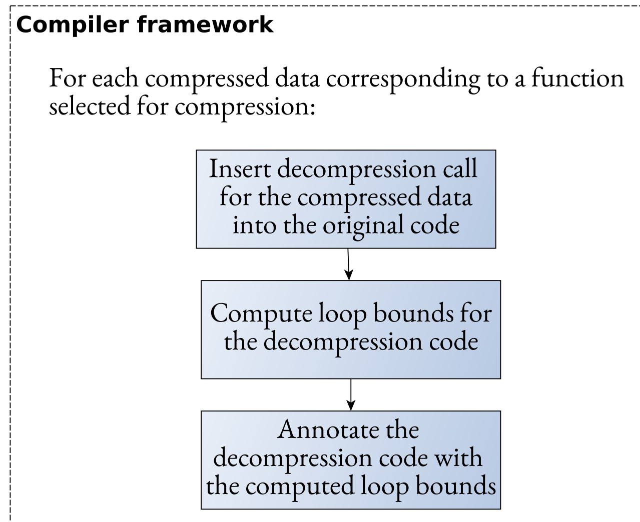


Figure 5.3: Compiler-based preparation phase for runtime decompression. The compiler inserts calls of decompression to the original code for each compressed data, computes loop bounds of the LZ decompression routine, and annotates them to accurately compute the final WCET.

pressed bytes are always available during program execution, so the compiler does not include the code of the compression routine in the final executable file.

5.2.3 Decompression

After the compression phase described in the previous section, compressed data can be used by the LZ decompression routine at execution time. Figure 5.3 shows a preparation phase for runtime decompression.

For every compressed function, the compiler inserts a call of the decompression routine into the original code with the compressed data passed as a parameter. We integrate the compression technique at the low-level intermediate representation of WCC (ICD-LLIR) but add calls of the decompression routine at WCC's high-level intermediate representation (ICD-C, see Section 3.1). Since ICD-C is closer to C code, whereas ICD-LLIR is closer to assembly code, it is easier to insert a function call at ICD-C than at ICD-LLIR.

In the next step, the decompression code is analysed in terms of WCET: the compiler computes loop bounds for each call of the decompression routine and annotates the decompression code with them to enable a precise static WCET analysis described in the next Section 5.2.4.

As discussed in Assumption 3, to evaluate the potential of our approach, we insert all decompression calls right at the beginning of the program's entry point.

We assume that compressed functions are decompressed to a buffer which is any dedicated memory space suitable to store code.

5.2.4 WCET Estimation

To estimate the WCET of a final program, the WCET of an input code and the WCET of the LZ decompression code must be computed due to runtime decompression. We estimate WCETs at compile time by using the static WCET analyser aiT described in Section 2.1.3. aiT requires the input code annotated with flow facts to estimate its WCET more precisely. We assume that a user provides flow facts for the original program and the compiler automatically computes flow facts for the LZ decompression routine.

Listing 5.1 presents a code snippet of the LZ decompression routine called at runtime. The original decompression code has four input parameters:

- *input* is data to be decompressed;
- *length* is the length of the input data;
- *output* is a buffer (target memory address) for decompressed data;
- *maxout* is the maximum size of the decompressed data.

Loop bounds are flow facts that influence the precision of WCET estimation and can be easily computed for the LZ decompression code if input parameters are known. Since the WCC compiler knows the input information after the compression phase, it computes exact loop bounds for each call of the decompression routine as described later and annotates the program with parametric loop bounds described in Section 3.2. aiT estimates the WCET of the decompression code by utilizing the provided parametric loop bounds.

The decompression routine of the `FastLZ` library contains four loops, but in Listing 5.1, we show only two of them to demonstrate how the compiler computes the loop bounds of the decompression code. We describe computation of loop bounds for the loop at Lines 19–22, the loop bounds of the remaining loops are computed in the same way. The loop bounds of the considered loop depend on a variable *len* defined at Line 14 and modified inside an if-statement (see Line 17) before the loop execution. Since *len* is defined through a variable *ctrl*, which is defined through a variable *ip* at Line 7 and *ip* is defined through the input parameter *input* at Line 5, the value of *len* depends on the input parameter, i.e. the loop bounds also depend on the decompressor input. To compute the lower and upper bounds of loop iterations, we slightly modify the input code of the decompressor:

- we add a new input parameter *loopbounds* which stores computed loop bounds and makes them available outside the decompression routine;

Listing 5.1: Code snippet of FastLZ library's decompression [Hid07].

```
1  int fastlz_decompress( const void* input, int length,
2                          void* output, int maxout,
3                          unsigned int *loopbounds )
4  {
5    const unsigned char* ip = (const unsigned char*)input;
6    ...
7    unsigned int ctrl = (*ip++) & 31;
8    int loop = 1;
9    unsigned int loopboud_min = 0;
10   unsigned int loopbound_max = 4294967295;
11
12   do{
13     ...
14     unsigned int len = ctrl >> 5;
15     ...
16     if( ctrl >= 32 ){
17       /*Some modifications of len*/
18       unsigned int loopbound_current = 0;
19       for( ; len; --len ){
20         *op++ = b;
21         loopbound_current++;
22       }
23
24       //Update lower and upper bounds, if necessary
25       if( loopbound_current < loopbound_min )
26         loopbound_min = loopbound_current;
27       else if( loopbound_current > loopbound_max )
28         loopbound_max = loopbound_current;
29     }
30     ...
31   }
32   while(loop);
33   ...
34   loopbounds[0] = loopbound_min;
35   loopbounds[1] = loopbound_max;
36   ...
37 }
```

Table 5.1: Server specifications.

CPU	Dual CPU Intel XEON Gold 6146
RAM	1.48 T
Number of CPU cores	48
CPU frequency	3.30 GHz
Operation system	Ubuntu 18.04.5 LTS

- we define two new variables *loopbound_min* and *loopbound_max* at Lines 9 and 10 to store the lower and upper bounds of the loop, respectively;
- for each execution of the loop, we compute the exact number of iterations as shown at Lines 18–22;
- we update the lower and upper bounds at Lines 25–28, if necessary.

The modified decompression code is only used by the compiler to get loop bounds for each compressed function at compile time, but the original decompression routine is used at runtime to avoid unnecessary timing overheads due to computation of loop bounds.

5.3 Evaluation

5.3.1 Experimental Setup

To evaluate the proposed compression technique, we used WCC for the Infineon TriCore TC1797 micro-controller with 88K of data section and 39K of SPM. We ran evaluations on a server with specifications presented in Table 5.1. To compute WCET, we used `aiT` version 20.10i, and to solve the ILP selection problem, we used `Gurobi Optimizer` version 8.1.0. We considered benchmarks from the following benchmark suites: JETBENCH, MRTC, MediaBench, PolyBench, UTDSP, linear-algebra, and misc with loop bounds annotated by TACLeBench project [Fal+16]. Appendix A presents all tested benchmarks.

To decompress a compressed function at runtime before its execution, the LZ decompressor is stored in the main memory and the SPM is utilized as a buffer because it is a fast software-controllable memory – SPM takes one cycle to access data whereas the main memory takes six cycles – and code can be directly executed from it (see Section 2.2).

We set the constants of the selection ILP model to the following values:

- $WCET_{\mathtt{limit}} = 1.5 \cdot WCET_{\mathtt{orig}}$, i.e. the final WCET of a program cannot increase by more than 50 % compared to the original WCET;

- $DS_{limit} = DS_{free}$, where DS_{free} is the available free space of the data section;

- $BUFF_{limit} = SPM_{free}$, where SPM_{free} is the available free space of the SPM;

Modeling function call penalties in Equations (5.8a)–(5.8c), we use two constants P_{high} and P_{low} which represent the penalties when a function calls another function placed in a different or the same memory, respectively. Kleinsorge [Kle] showed that for the Infineon TriCore TC1796 architecture, appropriate values of P_{high} and P_{low} are 16 and 8, respectively. We use these values in our evaluations, since the considered TriCore TC1797 architecture features the same internal structure of memories and pipeline as the TC1796.

After the prephase described in Section 5.2.1, for each function that is a compression candidate, WCC computes

- the WCET of the LZ decompression routine ($WCET_i^{decomp}$) by compressing the function, computing the loop bounds of the decompression code for the compressed function (see Section 5.2.4), annotating the decompression code with the loop bounds, and passing the decompression code to aiT;

- the WCET of all executions of the function from the main memory ($WCET_i^{main}$) by passing the original code to aiT and getting the WCET of the function from aiT;

- the WCET of all executions of the function from SPM ($WCET_i^{buffer}$) by moving the function to SPM, passing the modified code to aiT, and getting the WCET of the function from aiT.

5.3.2 Compression Results

For 71 out of 143 tested benchmarks, the prephase resulted in at least one function that was a compression candidate, the other 72 benchmarks contained only small functions that could not gain from compression.

Figure 5.4 shows the statistics of functions for the benchmarks with at least one compression candidate. Each graph depicts the results for specific benchmark suites. The x-axis of the graphs lists the benchmarks, whereas the y-axis shows the number of functions for each benchmark. The bars represent the total number of functions in a benchmark (blue), the number of functions chosen as compression candidates after the prephase described in Section 5.2.1 (orange), and the number of compressed functions after solving the ILP selection model formulated in Section 5.2.1 (green).

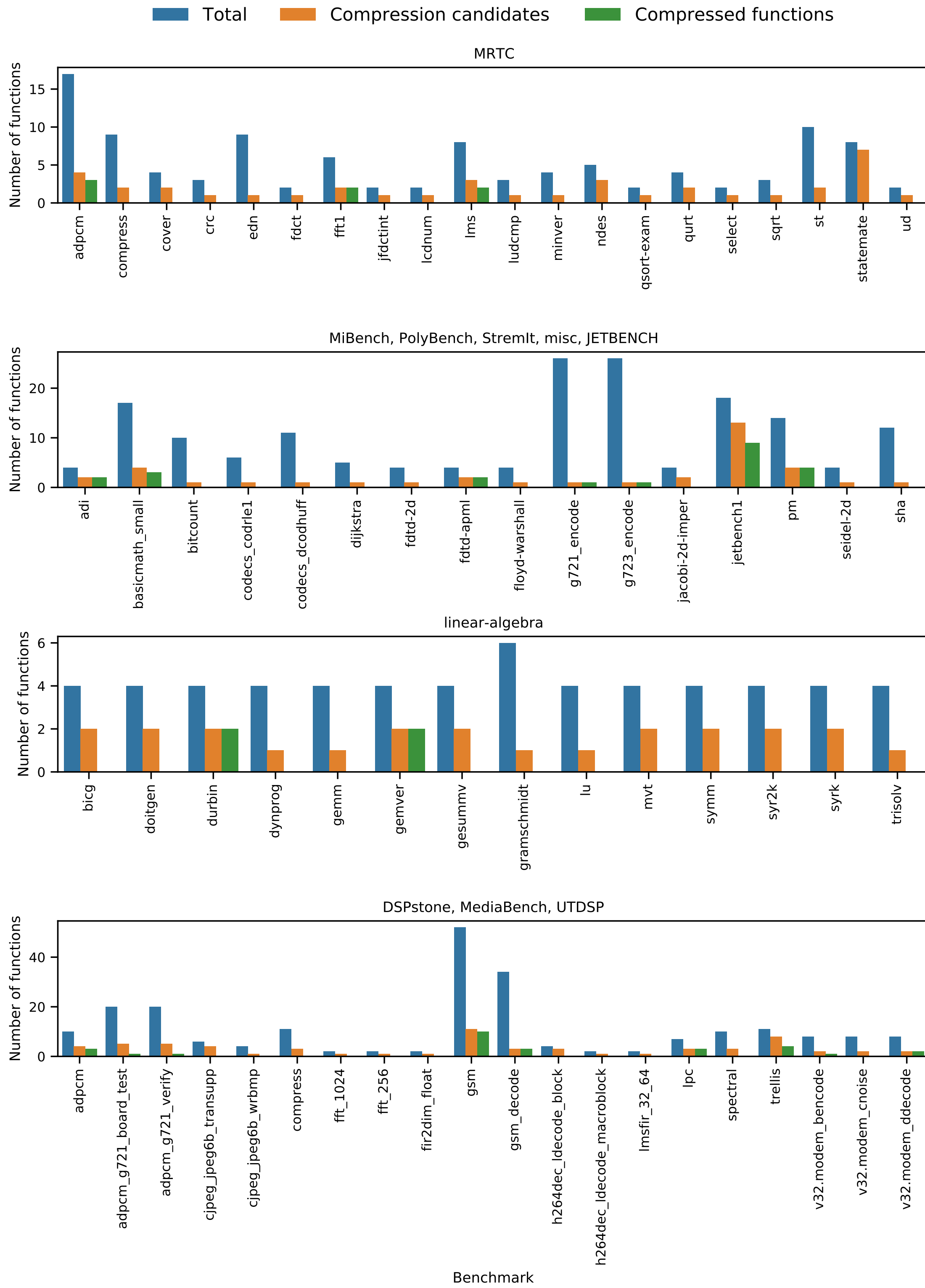


Figure 5.4: Statistics for functions subject to compile-time compression for benchmarks with at least one function selected as a compression candidate after the prephase

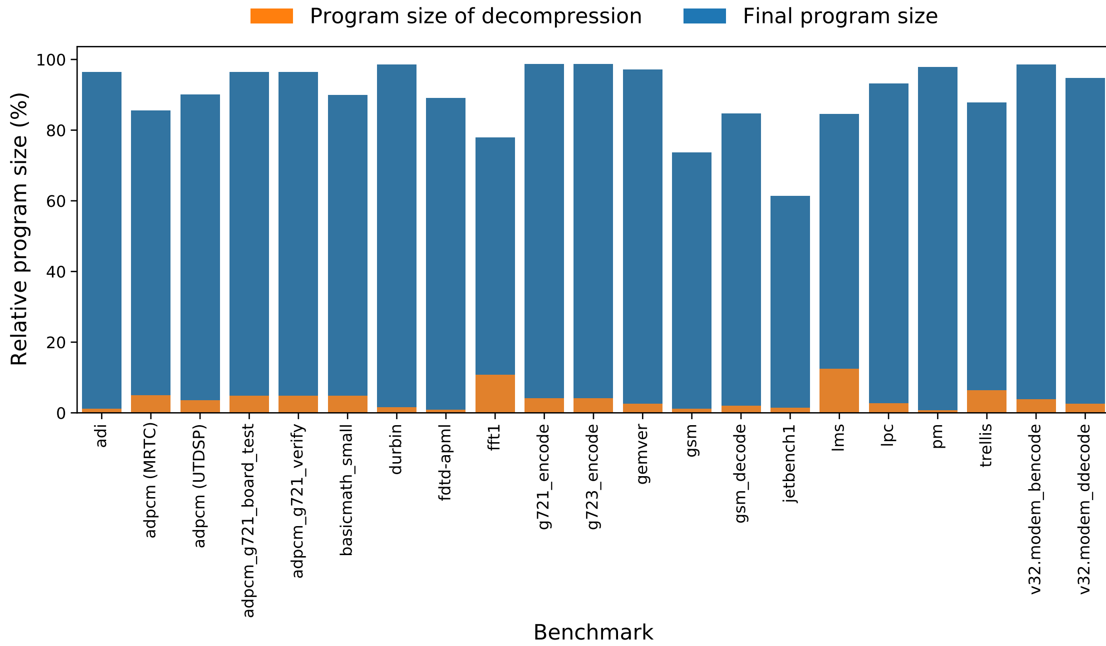


Figure 5.5: Statistics for the program size of benchmarks after compression. 100 % corresponds to the original program sizes of benchmarks.

During the prephase, a few functions were selected as compression candidates for many benchmarks, since the benchmarks contain many functions with the size of compressed data being greater than the size of the original function, i.e. $R > 1$. For the benchmarks from Figure 5.4, the total number of functions differs from 2 to 52; three functions, on average, were selected as candidates for compression; and in 19 benchmarks, three functions, on average, were finally compressed. For many benchmarks, after solving the ILP problem from Section 5.2.1, none of prephase candidates were chosen for compression because of the violated program size constraint (5.15), i.e. the program size of the decompression code (290 bytes) added to the final binary file was larger than program size decrease due to decompression, which led to overall program size increase.

Figure 5.5 shows the statistics for the program size of benchmarks that profited from compression; the x-axis shows the benchmarks, whereas the y-axis presents relative program size, ranging from 0 % to 100 %, where 100 % corresponds to the original program size of a benchmark. The stack diagram represents

- the program size of the LZ decompression routine inserted into the final binary file at compile time (orange);
- the overall program size of the final program after compression (blue).

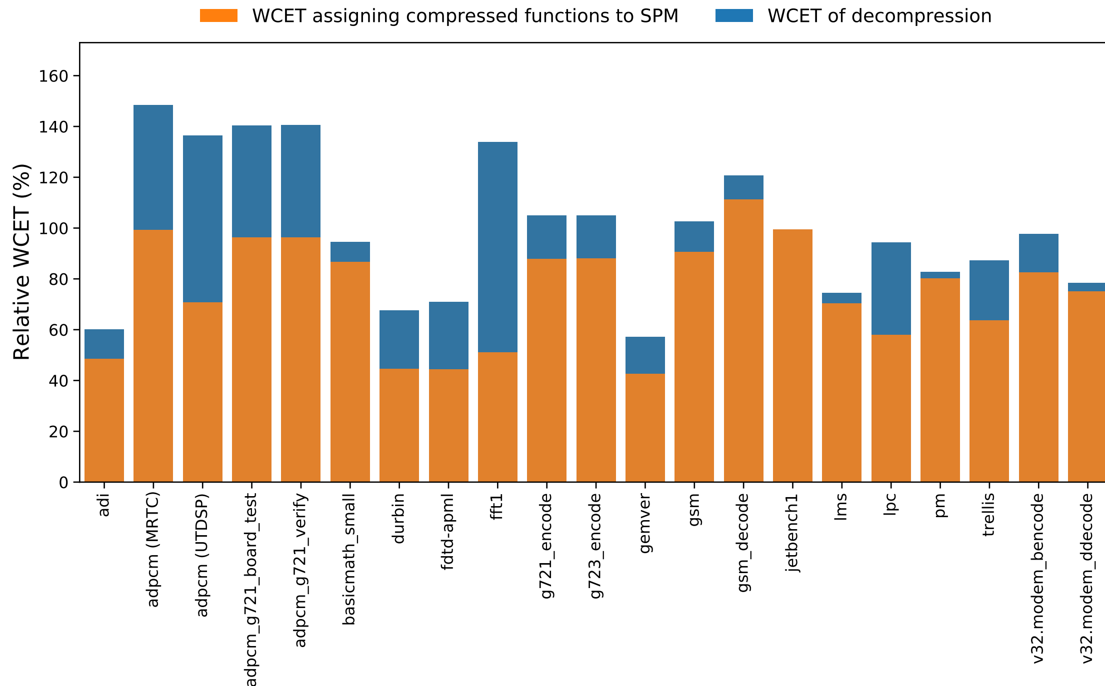


Figure 5.6: Statistics for WCET when decompressing compressed functions to the SPM. 100 % corresponds to the original WCETs of benchmarks.

Program size decreased by 11 %, on average, but for the benchmark *jetbench1*, which is the largest considered benchmark, its program size decreased by 39 %. For all benchmarks, the program size of the decompression routine was less than 10 % of the original program size – even for the smallest benchmarks *fft1* and *lms*.

Figure 5.6 shows relative WCET when compressed functions are decompressed to the fast SPM, i.e. the SPM is a buffer. The x-axis lists the benchmarks and the y-axis presents relative WCET, ranging from 0 % to 100 %, where 100 % corresponds to the original WCET of a benchmark. We present the final WCET of a benchmark as a stack diagram consisting of two parts:

- the WCET of the benchmark after assigning functions selected for compression to the SPM without compressing and decompressing them. It represents WCET decrease due to utilization of the fast SPM;
- the WCET of the decompression routine when decompressing compressed functions at runtime. It represents impact of the runtime decompression on WCET.

The average increase in WCET was 9 %. For some benchmarks like *adpcm*, the final WCET was close to the predefined limit of 150 %, whereas for other benchmarks like *adi*, the WCET decreased by 40 %.

Table 5.2: Statistics for the benchmark *jetbench1*.

Total number of functions	18
Number of compression candidates	13
Number of compressed functions	9
Original program size (bytes)	20,068
Final program size (bytes)	12,325
Program size of decompression (bytes)	290
Original code size of compressed functions (bytes)	16,886
Final size of compressed functions (bytes)	7,834
Original WCET (cycles)	10,065,929,314
Final WCET (cycles)	10,001,377,330
WCET of decompression (cycles)	3,438,441

For the largest benchmark *jetbench1*, the WCET slightly decreased, the WCET of the decompression routine was almost negligible compared to the final WCET of the benchmark, but according to Figure 5.5, the final program size decreased by 39 %. To explain such results, Table 5.2 shows more detailed statistics for the benchmark *jetbench1*. The benchmark has 13 large functions, their compression potentially decreases the program size of the benchmark. Half of all functions were finally compressed without violating any ILP constraint.

The final program size decreased by 39 %, and the program size of the decompression routine was only 2 % of the final program size. The original code size of nine compressed functions amounts to 83 % of the original benchmark's program size, and their total size decreased by 54 % after compression.

Although the compiler compressed many functions which had to be decompressed at runtime, since they were executed from the fast SPM and the original WCET of the benchmark is large enough, the WCET decreased by 1 %. The WCET of the decompression routine is less than 1 % of the final WCET.

In our evaluations, we used the SPM as a buffer, which is faster than the main memory and is often utilized to improve WCET. Figure 5.7 shows relative WCET if compressed functions are decompressed to the slower main memory (i.e. a part of the main memory is a buffer) instead of the faster SPM. The x-axis presents the benchmarks and the y-axis shows relative WCET with 100 % corresponding to the original WCETs of a benchmark. While decompressing functions to the main memory, the figure shows that WCET can only increase due to additional overhead because of runtime decompression, so for all benchmarks, the final WCET was greater than 100 %. For these experiments, we disabled the WCET constraint in the ILP model to demonstrate benchmarks for which the constraint is violated due to a slower buffer (the main memory). For many benchmarks,

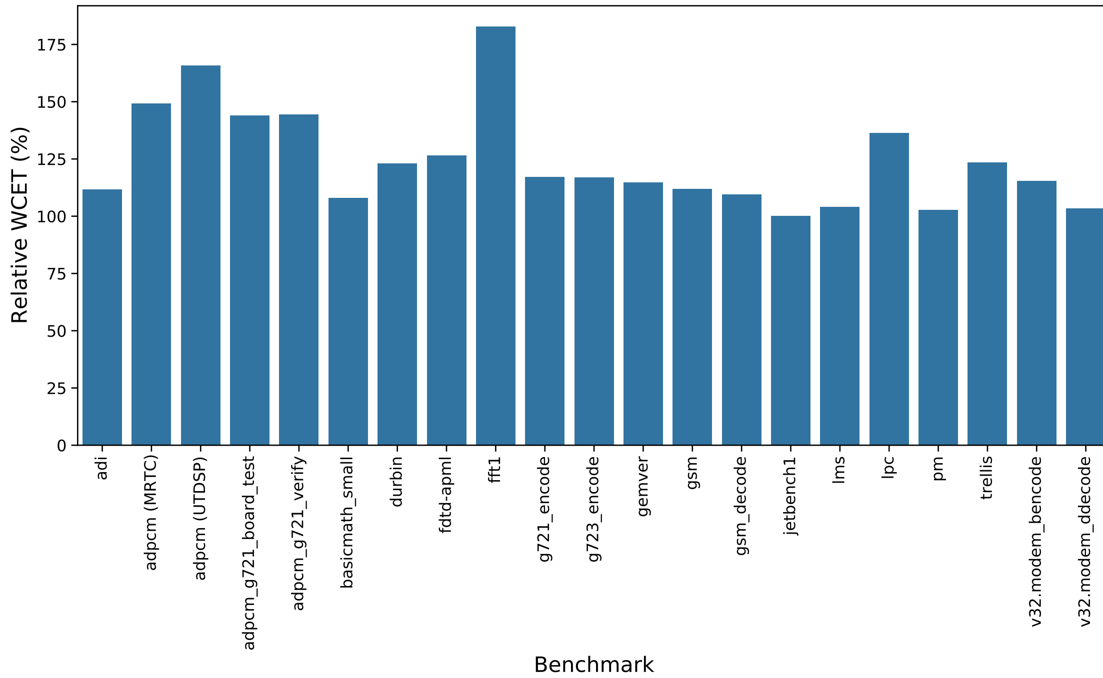


Figure 5.7: Statistics for WCET when decompressing compressed functions to the main memory. 100 % corresponds to the original WCETs of benchmarks.

the WCET slightly increased and the WCET constraint was satisfied, only for two benchmarks – *adpcm* from the benchmark suite UTDSP and *fft1* – the constraint would be violated. It means that the selected decompression is fast enough to be used at runtime.

Table 5.3 shows the runtime of the LZ compression including the runtime of the following steps:

- prephase: compress functions and identify compression candidates whose size decreases after compression;
- preparing ILP:
  - compute ILP constants: the available space of data memory (DS_{limit}) to store compressed functions and the available free space of the SPM to decompress functions ($BUFF_{limit}$);
  - compute original WCETs ($WCET_{orig}$), original program sizes (PS_{orig}), and the program size of the LZ decompression code (PS_{decomp});
  - for each compression candidate, compute loop bounds of the LZ decompression routine (see Section 5.2.4), annotate it, and compute its WCET ($WCET_i^{decomp}$);

Table 5.3: Compression runtime.

Benchmark	Prephase (sec)	Preparing ILP (sec)	Solving ILP (sec)	Finalizing (sec)	Total (sec)
adi	<1	68	<1	28	96
adpcm (MRTC)	<1	27	<1	12	39
adpcm (UTDSP)	<1	3	<1	1	4
adpcm_g721_board_test	<1	2	<1	2	4
adpcm_g721_verify	<1	7	<1	2	9
basicmath_small	<1	3	<1	1	4
durbin	<1	15	<1	8	23
fdtd-apml	5	82	<1	50	137
fft1	1	26	<1	17	44
g721_encode	<1	4	<1	2	6
g723_encode	<1	5	<1	2	7
gemver	<1	22	<1	8	30
gsm	2	8	<1	5	15
gsm_decode	1	2	<1	3	6
jetbench1	65	248	<1	259	572
lms	<1	2	<1	3	5
lpc	<1	18	<1	7	25
pm	<1	47	<1	12	59
trellis	<1	<1	<1	1	1
v32.modem_bencode	<1	3	<1	1	4
v32.modem_ddecode	<1	7	<1	2	9

- for each compression candidate, compute its WCET when executing it from the main memory ($WCET_i^{main}$) and the SPM ($WCET_i^{buffer}$);
- for each compression candidate, compute the code size of the function (CS_i), code size increase due to calls of the LZ decompression (CS_i^{decomp}), and the data size of the compressed data (DS_i);

In total, this step requires at least three WCET estimations of a program to be compiled:

- one estimation to compute the original WCET ($WCET_{orig}$) of the program and WCETs when functions are executed from the main memory ($WCET_i^{main}$);
- one estimation to compute the WCET of the LZ decompression routine for compression candidates ($WCET_i^{decomp}$): we insert decompression calls for all compression candidates, estimate the WCET of the result-

ing program, and extract the WCET of the decompression code for each individual call;

- one estimation to compute WCETs when compression candidates are executed from the SPM ($WCET_i^{buffer}$), if all compression candidates fit into the SPM. If the code size of the compression candidates is larger than SPM size, the compression candidates are grouped such that each group fits into the SPM, and WCET estimation is repeated for each group.

- solving ILP: solve the ILP problem by using `Gurobi Optimizer` version 8.1.0;

- finalizing: insert calls of the decompression routine into the final executable file for functions selected by the ILP model, store compressed functions as data objects in the final executable file, delete the compressed functions from the final executable file, and compute the WCET and program size of the final program.

The column "Total" in Table 5.3 represents the sum of the steps listed above. For 18 out of 21 benchmarks, the compression runtime was less than 60 sec. In the case of the largest benchmark, *jetbench1*, compression took around 9.5 min which is acceptable, since we compress functions at compile time and it does not affect the WCET of a final program.

For many benchmarks, the prephase took less than 1 s to compress all functions. For *jetbench1*, the prephase took 65 s, since it contains 18 functions (the compiler compresses each function to get compression ratio and identify 13 compression candidates) and the `jetbench1`'s functions are larger than functions present in the other benchmarks. The most time-demanding steps are preparing the ILP and finalizing compression to enable runtime decompression since both steps invoke a static analyser to compute WCETs, which is a time-consuming process. For *jetbench1*, the finalizing step was a little bit slower than the ILP preparation step due to the invoked WCET analyser and the necessity to generate WCC's low-level representation (ICD-LLIR) from the high-level representation (ICD-C), since we insert calls of the decompression routine at ICD-C as motivated in Section 5.2.3. Generation of ICD-LLIR can be time-consuming for complex benchmarks like *jetbench1*. For all benchmarks, `Gurobi Optimizer` solved constructed ILPs in less than 1 s.

5.4 Conclusion

This chapter has shown that a multiobjective compiler-based optimization can be formulated as a single-objective optimization problem if the goal is to opti-

mize one objective as much as possible and keep other objectives within predefined limits. We demonstrated the approach by presenting a novel compiler-based compression technique where compression takes place at compile time and decompression at runtime. The aim was to solve a multiobjective compression problem with two objectives: WCET and program size. The main goal of any compression is to decrease program size, so we reformulated the problem as a single-objective ILP problem with program size as an objective and WCET as a constrained parameter. We showed that the approach has several advantages:

- a single-objective optimization problem results in a solution that can be used to produce a final executable file without any further analysis, in contrast, a multiobjective problem results in a set of trade-offs between the objectives, and one requires a decision maker to select a final solution;
- a single-objective problem can be easily solved at compile time by using an appropriate solver, e.g. for evaluated benchmarks, `Gurobi Optimizer` solved the ILP compression problem in less than 1 s, and the WCC compiler produced a compressed file in less than 1 min for many benchmarks.

The disadvantage of reformulating a multiobjective problem as a single-objective one is that if any constraint is violated in the search space, then no solution is produced, and one can only manually try to find trade-offs between the objectives by relaxing constraints or changing objective functions. A pure multiobjective problem is usually more challenging to solve, but it allows avoiding unnecessary constraints and producing a set of possible trade-offs between the objectives more automatically, so in the following sections, we focus on pure multiobjective compiler-based optimization problems.

6 Evolutionary Algorithms for Multiobjective Compiler-Based Optimizations

Compiler-based optimizations are efficient techniques to improve a program to be compiled concerning a single objective according to Muchnick [Muc98]. Most original compiler-based optimizations minimize average-case performance but they can also improve WCET as was shown, e.g. by Falk and Schwarzer [FS06] or Oehlert, Luppold, and Falk [OLF17].

In addition to WCET constraints, modern hard real-time systems must regard additional design criteria such as code size and energy consumption. Since these three objectives contradict each other, it is impossible to optimize them simultaneously, and optimization problems become multiobjective. The previous chapter demonstrates how a compiler-based multiobjective problem can be reformulated as a single-objective problem if a meaningful single-objective function can be constructed out of multiple objectives.

In this chapter, we focus on compiler-based multiobjective optimizations that cannot be reformulated as single-objective problems because none of the objectives is preferable over the others. This chapter aims to solve a multiobjective problem at compile time and identify bottlenecks of the solution process. Since a unique solution does not exist, a compiler must return a set of possible trade-offs between objectives. Final decision regarding the most preferred solution from a solution set is out of the scope of this thesis.

To demonstrate our approach, we consider a well-known compiler-based optimization called *function inlining*. It substitutes function calls by the body of the function and was originally proposed to reduce average-case performance, but Lokuciejewski et al. [Lok+09] successfully applied it to minimize WCET. We formulate function inlining as a multiobjective problem by considering WCET, code size, and energy consumption as objectives.

Evolutionary algorithms are widely used population-based heuristics to solve multiobjective problems. They extensively explore the search space of a problem and return a set of trade-off solutions. In this chapter, we show the advantages and disadvantages of applying evolutionary algorithms to solve compiler-based optimization problems for hard real-time systems. For this purpose, we solve the multiobjective function inlining problem by using evolutionary algo-

rithms. We propose and evaluate extension of a well-known evolutionary algorithm, Generalized Differential Evolution (GDE), to solve multiobjective problems with binary-encoded search spaces. Since any evolutionary algorithm contains some user-defined control parameters, we also present a sensitivity analysis for the considered function inlining problem which answers the question of how changes in control parameters affect the quality of a solution set.

The approach described in this chapter was presented at the International Conference on Real-Time Networks and Systems (RTNS) in 2020 [MF20b].

The chapter is organized as follows: Section 6.1 presents related work regarding methods to solve multiobjective problems and their usage at compile time, Section 6.2 describes evolutionary algorithms considered in the thesis, Section 6.3 formulates the multiobjective function inlining problem, Section 6.4 contains evaluation results, and Section 6.5 gives a conclusion.

6.1 Related Work

Most methods to solve multiobjective problems can be divided into two classes: *iterative scalarization methods* and *evolutionary algorithms*[1]. Scalarization methods explore an algorithm that produces one Pareto optimal solution (see Section 4.1.2) in each run and repeats the algorithm with some changes to get a set of trade-off solutions. In contrast, evolutionary algorithms produce a set of solutions in one run.

Scalarization methods. The most commonly used scalarization method is *the weighted sum method* used, e.g. by Koski [Kos88], or Jahn, Klose, and Merkel [JKM92]. A multiobjective problem is converted into a single-objective problem by optimizing a convex linear combination of the objectives. The method produces a set of solutions by changing the weights in the linear combination. Das and Denis [DD97] or Messac and Ismail-Yahaya [MIY01] showed that if a Pareto front is nonconvex, there do not exist weights for which a solution to the single-objective problem lies in the nonconvex part. Later, Das and Denis [DD98], Messac and Mattson [MM02; MIYM03], Motta, Afonso, and Lyra [SMAL12], and others proposed mathematical programming-based methods that overcome the issue of the weighted sum method and can produce a nonconvex Pareto front.

According to Coello et al. [Coe+19], scalarization methods have the following limitations: they are sensitive to the shape and continuity of Pareto front and produce one solution per execution. An alternative to such approaches is evolutionary algorithms which are more flexible, less domain specific, and generate a set of trade-off solutions in one execution.

[1] Section 4.2 describes evolutionary algorithms in detail.

Evolutionary algorithms. Many original evolutionary algorithms are single-objective, but they have been extended to solve multiobjective problems. In each iteration, any evolutionary algorithm deals with a set of solutions called population. In 1985, Shaffer proposed the first multiobjective evolutionary algorithm called the Vector Evaluated Genetic Algorithm (VEGA) [Sch85]: a population is divided into as many subpopulations as the number of objectives; in each subpopulation, optimal solutions are selected concerning a single objective; a new population is created by recombining optimal solutions from all subpopulations. A drawback of VEGA is that each solution is evaluated with one objective ignoring other objectives.

Goldberg proposed *Pareto ranking* [Gol89] to select solutions by using the concept of Pareto-optimality, i.e. solutions are compared by taking not just one objective but all objectives into account. Many implementations of Pareto ranking are presented in the literature, e.g.

- Fonseca and Fleming proposed the Multi-Objective Genetic Algorithm (MOGA) [FF93] which compares each individual to all other individuals in terms of Pareto-optimality;

- Srinivas and Deb developed the Nondominated Sorting Genetic Algorithm (NSGA) [SD94] which ranks a population several times and creates layers of solutions.

Elitism is another mechanism incorporated in many evolutionary algorithms. It retains top-ranking solutions found by an evolutionary algorithm and avoids their elimination when the algorithm creates new solutions. E.g. Zitzler and Thiele proposed the Strength Pareto Evolutionary Algorithm (SPEA) [ZT98a] that stores nondominated solutions in an archive and prunes the archive if it exceeds its user-defined limit.

Many other evolutionary algorithms are presented in the literature [ZK04; EBN05; ZL07; BZ11] but according to Coello et al. [Coe+19], NSGA-II proposed by Deb et al. [Deb+02a] remains the most widely used algorithm. NSGA-II improves the ranking scheme of NSGA and uses a crowded comparison operator to guide the algorithm towards a uniformly spread Pareto front. The crowded comparison operator relies on the crowding distance that measures the density of solutions surrounding a given solution. Many researchers use NSGA-II nowadays despite limitations of the crowded comparison operator when solving a multiobjective problem with more than two objectives. Deb et al. [Deb+02b] stated that NSGA-II provides good diversity for bi-objective problems but fails for so-called many-objective problems. Kukkonen and Deb [KD06] showed that the crowding distance fails to estimate the crowdedness of solutions in the case of many-objective problems.

In this thesis, we consider evolutionary algorithms to solve a multiobjective problem, since they are more flexible and less domain-dependent than scalarization methods. Since evolutionary algorithms have been rarely used in the context of solving compiler-based multiobjective optimization problems, in this chapter, we evaluate them in terms of solving multiobjective optimizations at compile time and show their limitations in this case. In the next chapters, we present possible ways to tackle these limitations.

Parameters of evolutionary algorithms. Most evolutionary algorithms rely on user-defined parameters (e.g. crossover and mutation parameters described in Section 4.2). A user sets them before running an algorithm. These parameters control the behaviour of evolutionary process and can significantly influence algorithm's performance.

Hinterding, Michalewicz, and Eiben [HME97], Eiben and Smit [ES11b], and many other researchers stated that one should tune an evolutionary algorithm to a particular problem. Jong [Jon07] mentioned that many early evolutionary algorithms dealt with search spaces encoded by using real-valued vectors, so a Gaussian operator with a mean of zero and a variance of σ^2 was used to create new search vectors. Fixing σ for an evolutionary run was suboptimal and dynamically adapting mechanisms were developed. Rechenberg and Schwefel proposed the first adapting rule called the "1/5th rule" [Sch77]. They monitored the ratio of new vectors that improve fitness to the ones that degrade it. If the ratio became below 1/5, they decreased σ and increased it otherwise.

Later, researchers started to distinguish between two approaches to set the parameters: parameter tuning and parameter control. Parameter tuning seeks parameter values before running an algorithm and they remain fixed during algorithm execution, whereas parameter control adapts parameters during algorithm execution. Branke [Bra02], Kramer [Kra10], and other researchers proposed many parameter control strategies but almost none of them are used in practice. Jong [Jon07] explained this by the fact that it is difficult to estimate performance improvements achieved through dynamic parameter settings. In contrast, parameter tuning strategies are widely used in practice.

Rechenberg [Rec94] tuned parameters by following a two-level evolutionary algorithm: a top-level evolutionary algorithm seeks parameter values for a low-level evolutionary algorithm, which solves a problem with the found fixed parameters. In this case, the question remains about how to set parameters of the top-level evolutionary algorithm.

The parameter space of any evolutionary algorithm is often large, so it is often explored offline as Goldberg [Gol13] described: parameter values are selected by using sample problems from a class of problems and used to solve new problems from the same class. Jong [Jon07] stated that in practice, to find the best parameter values, various parameters are tested by running an evolutionary algorithm

multiple times. E.g. Nannen and Eiben [NE07] proposed the REVAC tool that automatically searches for optimal parameter values following a generate-and-test heuristic. In each iteration, new parameter values are tested by executing an evolutionary algorithm with given parameters and measuring algorithm's performance. (A performance measure is specified by a user.)

Multiobjective compiler-based optimizations. Multiobjective approaches are rarely used to perform compiler-based optimizations. Lokuciejewski et al. [Lok+11] considered a problem of finding optimal compiler optimization sequences; interactions between optimizations are complex and a smartly chosen sequence of optimizations prevents miss of optimization potential and performance degradation. The authors considered two bi-objective problems trading off WCET and average-case performance as well as WCET and code size. They exploited evolutionary algorithms (SPEA2 [AE07], NSGA-II [Deb+02a], and IBEA [ZK04]) to approximate a set of Pareto optimal compiler optimization sequences for each pair of objectives.

Jadhav and Falk [JF19] presented a multiobjective static SPM allocation with two objectives: WCET and energy consumption. The static SPM allocation identifies parts of a program that should be allocated to a fast SPM (instead of a slow main memory) at compile time to improve the objectives. The authors proposed to utilize the Flower Pollination Algorithm (FPA) developed by Yang [Yan12] to solve the multiobjective allocation problem.

6.2 Differential Evolution

As motivated in the previous section, we focus on evolutionary algorithms. Section 4.2 describes the core components of any evolutionary algorithm, whereas this section describes evolutionary algorithms used in our evaluation.

Many variants of evolutionary algorithms are presented in the literature: genetic algorithms [Gol89], evolution strategy [BS02], evolutionary programming [FF96], differential evolution [SP97], particle swarm optimization [WTL17], etc. In this section, we focus on differential evolution which we use in the thesis to demonstrate proposed approaches. For more details about other variants, we refer to Eiben and Smith [ES15, Chapter 6].

Differential Evolution (DE) is an evolutionary algorithm proposed by Storn and Price [SP97]. According to Eiben and Smith [ES15], DE is a simple but efficient global optimization technique that does not require any gradient information and can solve nonlinear nondifferentiable nonconvex problems. Vesterstrom and Thomsen [VT04], Rekanos [Rek08], Ponsich and Coello [PC11] showed that DE outperforms particle swarm optimization and genetic algorithms in real-world

problems. In this section, we describe the standard DE for single-objective optimization problems followed by its extension toward multiobjective optimization.

Standard differential evolution. A population of the standard DE consists of individuals with real-valued search vectors. The initial population is randomly generated.

In the literature, several *mutation* schemes have been presented; each scheme is described through the notation $DE/a/b$, where DE stands for differential evolution, a denotes a strategy to choose a base vector (e.g. $rand$ or $best$), and b specifies the number of different vectors in a perturbation vector (introduced in Equation (6.1)). We describe the most commonly used strategy $DE/rand/1$ while assuming a minimization problem. We denote by NP a constant population size. A mutation operator creates NP mutated search vectors $\mathbf{u}_i, i = \overline{1, NP}$ from three randomly chosen individuals r_1, r_2, and r_3 with $r_1 \neq r_2 \neq r_3 \neq i$ as follows:

$$u_{j,i,g} := x_{j,r_3,g} + F \cdot (x_{j,r_1,g} - x_{j,r_2,g}), \tag{6.1}$$

where

- j is a coordinate of a vector;
- g is the index of a generation;
- $\mathbf{x}_{r_1,g}, \mathbf{x}_{r_2,g}, \mathbf{x}_{r_3,g}$ are the search vectors of chosen individuals;
- $\mathbf{x}_{r_3,g}$ is a base vector;
- $\mathbf{x}_{r_1,g} - \mathbf{x}_{r_2,g}$ is a perturbation vector;
- F is a positive real constant called *scaling factor* defined by a user. It controls the robustness and speed of exploring a search space. The algorithm converges faster to an optimum with a lower value of F but may get stuck in a local optimum.

A *crossover* operator generates a new vector by crossing a search vector $\mathbf{x}_i$, which corresponds to the i-th individual of the current population, and the mutated vector $\mathbf{u}_i$ coordinate-wise:

$$v_{j,i,g} := \begin{cases} u_{j,i,g}, & \text{if } rand[0,1) < CR \text{ OR } j = j_{rand}, \\ x_{j,i,g}, & \text{otherwise,} \end{cases} \tag{6.2}$$

where

- $rand[0, 1)$ is an operator returning a random number from $[0, 1)$;

- j_{rand} is a random number from $\{1, 2, \cdots, NP\}$;
- CR is a user-defined probability of choosing the mutated vector $\mathbf{u}_{i,g}$ or the old vector $\mathbf{x}_{i,g}$.

The condition $j = j_{rand}$ in Equation (6.2) guarantees that at least one coordinate of the new vector $\mathbf{v}_{i,g}$ differs from the coordinate of the old vector $\mathbf{x}_{i,g}$.

A *selection* operator chooses between the newly generated vector $\mathbf{v}_{i,g}$ and the old vector $\mathbf{x}_{i,g}$ based on the values of a fitness function f:

$$\mathbf{x}_{i,g+1} = \begin{cases} \mathbf{v}_{i,g}, & \text{if } f(\mathbf{v}_{i,g}) < f(\mathbf{x}_{i,g}), \\ \mathbf{x}_{i,g}, & \text{otherwise.} \end{cases} \tag{6.3}$$

Generalized differential evolution. Many extensions of the standard DE have been proposed to solve multiobjective problems, e.g. Chang, Xu, and Quek presented a Pareto-based Differential Evolution approach [CXQ99], Babu and Jehan developed Vector Evaluated Differential Evolution (VEDE) [BJ03], Lampinen proposed Generalized Differential Evolution (GDE) [Lam01], etc. In this thesis, we utilize The Third Version of GDE (GDE3) proposed by Kukkonen and Lampinen [KL05], since Durillo et al. [Dur+10] showed that it outperforms the widely used NSGA-II algorithm. We use GDE3 to demonstrate the advantages and disadvantages of evolutionary algorithms in terms of solving multiobjective problems at compile time, but our findings remain valid for any other evolutionary algorithm.

Lampinen introduced the first version of GDE [Lam01]. It extends the standard DE for multiobjective optimization by changing the selection rule as follows:

$$\mathbf{x}_{i,g+1} = \begin{cases} \mathbf{v}_{i,g}, & \text{if } \mathbf{v}_{i,g} \preceq \mathbf{x}_{i,g}, \\ \mathbf{x}_{i,g}, & \text{otherwise.} \end{cases} \tag{6.4}$$

The selection rule is based on the concept of Pareto-optimality described in Section 4.1.2. It replaces the old vector $\mathbf{x}_{i,g}$ in the next generation if the new vector $\mathbf{v}_{i,g}$ weakly dominates it. All other parts of the standard DE are preserved.

Kukkonen and Lampinen [KL04b] showed that the first version of GDE lacks a mechanism to maintain distribution of solutions and is too sensitive to its control parameters (scaling factor F and crossover parameter CR) but it can handle any number of objectives. They proposed The Second Version of GDE (GDE2) [KL04a] that improves the GDE's selection operator and selects a survivor vector based on Pareto dominance and crowdedness introduced by Deb et al. in NSGA-II [Deb+02a]. The *crowding distance* estimates the density of surrounding solutions for each individual in a given population. Figure 6.1 shows

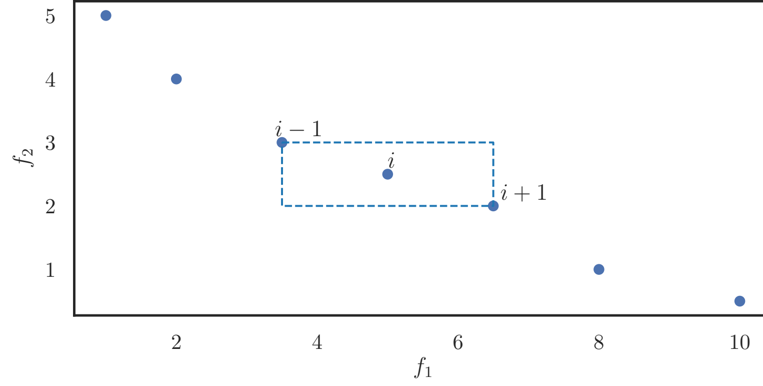


Figure 6.1: Example of a cuboid to calculate the crowding distance of the i-th individual in a 2-dimensional objective space. The cuboid is formed by the nearest neighbours $i-1$ and $i+1$. The plot is adapted from Deb et al. [Deb+02a].

an exemplary Pareto front in a 2-dimensional objective space: the x-axis corresponds to the first objective f_1, whereas the y-axis corresponds to the second objective f_2, the blue points represent the Pareto front, the blue dashed lines show a cuboid formed for the i-th individual by the nearest neighbours $i-1$ and $i+1$. The crowding distance is the average side length of the cuboid. Algorithm 8 in Appendix C presents a procedure to compute the crowding distance for individuals from a given population.

The selection operator of GDE2 prefers the new vector $\mathbf{v}_{i,g}$ over the old vector $\mathbf{x}_{i,g}$ if the new vector weakly dominates the old vector or the vectors are incomparable, i.e. none of the vectors dominates the other, and the crowding distance of the new vector is larger than the crowding distance of the old vector:

$$\mathbf{x}_{i,g+1} = \begin{cases} \mathbf{v}_{i,g}, \text{ if } \mathbf{v}_{i,g} \preceq \mathbf{x}_{i,g} \text{ OR } (\mathbf{x}_{i,g} \not\prec \mathbf{v}_{i,g} \text{ AND } \mathtt{CDist}(\mathbf{x}_{i,g}) < \mathtt{CDist}(\mathbf{v}_{i,g})), \\ \mathbf{x}_{i,g}, \text{ otherwise,} \end{cases} \tag{6.5}$$

where

- $\mathbf{x}_{i,g} \not\prec \mathbf{v}_{i,g}$ means that the vectors are incomparable in terms of Pareto dominance;
- $\mathtt{CDist}$ denotes the crowding distance.

Kukkonen and Lampinen [KL05] stated that GDE2 improves distribution of solutions but degrades convergence of a population to Pareto front and is still too sensitive to control parameter values.

The third version of GDE improves further the GDE2's selection operator while preserving the remaining parts of the standard DE. The selection operator of

GDE2 keeps the population size unchanged since it selects only one out of two vectors, but the selection operator of GDE3 allows a population to grow. In the next generation, it keeps

- the new vector $\mathbf{v}_{i,g}$ if the new vector weakly dominates the old vector $\mathbf{x}_{i,g}$;
- the old vector $\mathbf{x}_{i,g}$ if the old vector weakly dominates the new vector $\mathbf{v}_{i,g}$;
- both vectors $\mathbf{v}_{i,g}$ and $\mathbf{x}_{i,g}$ if the vectors are incomparable.

Since the population grows, after selecting individuals to be kept for the next generation, GDE3 prunes the population by using nondominated sorting. The individuals of the population are sorted by rank and crowding distance. The *rank* of an individual represents the number of individuals in the population dominating the current individual. A lower rank indicates a higher quality of the individual. Algorithm 7 in Appendix C describes a procedure to compute individuals' ranks.

To prune a population, GDE3

1. assigns ranks to individuals;
2. sorts the individuals by rank in ascending order;
3. sorts the individuals by crowding distance in descending order for each rank;
4. removes the last individuals that exceed a predefined population size.

We present the full algorithm of GDE3 in Appendix C. Input parameters $\mathbf{x}^{min}$ and $\mathbf{x}^{max}$ represent bounds of a search space. They are used to initialize an initial population with random feasible search vectors.

Kukkonen and Lampinen [KL05] showed that GDE3 improves the diversity of the previous versions of GDE in terms of solutions' distribution and it is less sensitive to changes in the control parameters.

Binary differential evolution. Most compiler-based optimizations are binary-encoded problems, i.e. the search space of a problem consists of binary vectors and objective functions are defined only on binary vectors. The standard DE and its multiobjective extensions handle real-valued variables, i.e. the decision space of a problem is a subspace of a real-valued vector space.

In this thesis, we demonstrate our approaches by solving the multiobjective function inlining problem described in Section 6.3. It is a binary-encoded problem, so we cannot apply the original GDE3 to solve it. This issue appears because of the DE's mutation operator defined in Equation (6.1): the scaling factor F is a real number, so the mutation operator creates a new real-valued vector $\mathbf{u}_{i,g}$.

Wang et al. presented the Modified Binary Differential Evolution (MBDE) [Wan+10] for single-objective problems; it preserves the mutation, crossover, and selection operators of the standard single-objective DE and adds a probability estimation operator to map real-valued vectors produced by the mutation operator to binary-encoded vectors. The authors showed that MBDE outperforms the discrete binary particle swarm optimization developed by Kennedy and Eberhart [KE97], the binary ant system introduced by Kong and Tian [KT05], and the discrete binary differential evolution presented by Peng, Jian, and Zhiming [PJZ08] in terms of accuracy and convergence speed.

The MBDE's probability estimation operator is a sigmoid function which is often used to map real numbers into the interval $[0, 1]$:

$$P_{DE}(x) = \frac{1}{1 + e^{-(2b \times (x-0.5))/(1+2F)}}, \tag{6.6}$$

where x is a real number, $b \in \mathbb{R}_{\geqslant 0}$ is a *bandwidth* parameter that describes the smoothness of the operator and F is the scaling factor introduced in Equation (6.1).

MBDE applies the probability estimation operator to the new vector $\mathbf{u}_{i,g}$ created by the mutation operator introduced in Equation (6.1) coordinate-wise and maps the real coordinates of the vector $\mathbf{u}_{i,g}$ to binary values as follows:

$$bu_{j,i,g} = \begin{cases} 1, & \text{if } rand[0,1) \leqslant P_{DE}(u_{j,i,g}), \\ 0, & \text{otherwise.} \end{cases} \tag{6.7}$$

The new vector $\mathbf{bu}_{i,g}$ is binary-encoded and can be used further by the crossover and selection operators.

The advantage of MBDE's extension for binary-encoded problems is that it requires no changes in the original DE and minimal additional effort to map real-valued vectors into binary-valued vectors.

In our evaluation, we integrate the same mechanism into GDE3 to solve multiobjective binary-encoded problems; we call Binary GDE3 (BGDE3) GDE3 with the mutation operator extended by the probability estimation operator from Equation (6.6) and the mapping from Equation (6.7). Since such modification of GDE3 has never been evaluated, we compare its results to the Multiobjective Binary Probability Optimization Algorithm (MBPOA) proposed by Wang et al. [Wan+14]. The algorithm is also based on GDE3 and was specifically designed for binary-encoded problems.

MBPOA follows the procedure of GDE3 except the mutation operator, where a new vector $\mathbf{u}_{i,g}$ may become a real-valued vector. MBPOA substitutes the linear combination in Equation (6.1) by the following linear combination:

$$u_{j,i,g} = x_{j,best,g} + x_{j,r_2,g} + x_{j,r_3,g} \tag{6.8}$$

with randomly selected r_1, r_2 such that $r_2 \neq r_3$ and $\mathbf{x}_{best,g}$ denoting an individual with the highest index h in the current population. The mutation operator does not have a scaling factor but it includes an individual selected by using the index h. The index h – used also by Farina and Amato [FA04] and Peng, Sun, and Guo [PSG10] – is defined as follows:

$$h = \frac{1}{m} \sum_{i=1}^{m} h_i \tag{6.9}$$

with

$$h_i = \begin{cases} 1, & f_i \leqslant f_i^{min}, \\ \frac{f_i^{max} - f_i}{f_i^{max} - f_i^{min}}, & f_i^{max} > f_i > f_i^{min}, \quad i = \overline{1, m}, \\ 0, & f_i \geqslant f_i^{max}, \end{cases} \tag{6.10}$$

where m is the dimension of an objective space, f_i^{min} and f_i^{max} represent the minimum and maximum values of the i-th objective function in the current population, respectively. The index h represents the average fitness of an individual over all objectives with $h_i = 1$ denoting a top-scoring individual in terms of the objective i.

MBPOA uses the following probability estimation operator P_{MBPOA}:

$$P_{MBPOA}(u_{j,i,g}) = \begin{cases} \frac{1}{n}, & u_{j,i,g} = 0, \\ 0.667, & u_{j,i,g} = 1, \\ 0.333, & u_{j,i,g} = 2, \\ 1 - \frac{1}{n}, & u_{j,i,g} = 3, \end{cases} \tag{6.11}$$

where n is the dimension of a search space. Wang et al. showed that it is sufficient to assign the probability estimation operator to constant values for four possible values of $u_{j,i,g}$ (the summands in Equation (6.8) are binary numbers). For more details about derivation of the constants, we refer to the original paper of Wang et al. [Wan+14].

Binary coordinates of the vector $\mathbf{u}_{i,g}$ are mapped to binary values similar to MBDE:

$$bu_{j,i,g} = \begin{cases} 1, \text{ if } rand[0,1) \leqslant P_{MBPOA}(u_{j,i,g}), \\ 0, \text{ otherwise.} \end{cases} \tag{6.12}$$

Both algorithms BGDE3 and MBPOA preserve all parts of GDE3 except the mutation part where additional efforts are added to map newly created real-valued vectors to binary-valued vectors. We present the original GDE3 algorithm in Appendix C; its initialization part at Line 10 creates real-valued vectors. In

the case of BGDE3 and MBPOA, we replace GDE3's original initialization with an initial population containing random binary-valued vectors.

Parameter tuning. Similar to most other evolutionary algorithms, BGDE3 and MBPOA have user-defined parameters that influence algorithms' behaviour and performance. Both algorithms have a crossover parameter CR presented in Equation (6.2) and BGDE3 has two additional control parameters: a scaling factor F from Equation (6.1) and a bandwidth parameter b from Equation (6.6).

Eiben and Smit [ES11a] showed that by smartly setting control parameters, one can improve the performance of an evolutionary algorithm. The parameters can be set before running an algorithm by using parameter tuning or they can be adapted during algorithm execution by using parameter control mentioned in Section 6.1. In this thesis, we follow a parameter tuning approach, since it is usually used in practice as stated by Jong [Jon07].

The most commonly used approach to tune parameters is to perform many runs of an algorithm and to compare its performance for different parameter values [Jon07], similar to the REVAC tool [NE07] described in Section 6.1. In this thesis, we follow the same logic to identify the best parameter values for BGDE3 and MBPOA. We set the parameters to several possible values, solve the function inlining problem by using fixed parameters, and measure the performance of the algorithm by using the following quality indicators: nondominance ratio NR and coverage C described in Section 4.1.3. We select the best values of the parameters based on quality indicators' values.

For the function inlining problem, an exemplary optimization considered in the thesis and described in the next section, we compare the algorithms BGDE3 and MBPOA by running them with various parameter values and choose the best parameters values for them in Section 6.4.2.

6.3 Multiobjective Function Inlining

To demonstrate the advantages and disadvantages of evolutionary algorithms when solving multiobjective compiler-based optimization problems, we consider a well-known compiler-based optimization called *function inlining* but critical observations regarding evolutionary algorithms remain valid for all other multiobjective compiler-based optimizations.

Originally, function inlining – described by Muchnick [Muc98] – is a compiler-based optimization that aims to improve average-case performance, e.g. Wörteler et al. [Wö+15] applied function inlining to a query language for XML databases, XQuery, to achieve better performance.

Lokuciejewski et al. [Lok+09] presented a WCET-aware function inlining approach based on machine learning heuristics. The authors showed that WCET-

```
1: function FUNC_1( int a )
2:     res = 0
3:     if a > 0 then
4:         res = 2 · a + 5
5:     else
6:         res = 4 · a + 7
7:     return res
8:
9: function FUNC_2( int b )
10:    res = 0, i = 0
11:    Loop bound min 2 max 300.
12:    while i < b do
13:        res = res + 3 · i
14:        i = i + 1
15:    return res
16:
17: Function calls.
18: FUNC_1( 1 )
19: FUNC_2( 40 )
20: FUNC_2( 2 )
21: FUNC_2( 300 )
```

(a) Original program.

```
1: function FUNC_2( int b )
2:     res = 0, i = 0
3:     Loop bound min 2 max 300.
4:     while i < b do
5:         res = res + 3 · i
6:         i = i + 1
7:     return res
8:
9: Inlined func_1.
10: a = 1, res_func_1 = 0
11: if a > 0 then
12:     res_func_1 = 2 · a + 5
13: else
14:     res_func_1 = 4 · a + 7
15:
16: Inlined func_2.
17: b = 40, res_func_2 = 0, i = 0
18: Loop bound min 40 max 40.
19: while i < b do
20:     res_func_2 = res_func_2 + 3 · i
21:     i = i + 1
22:
23: Remained original calls of func_2.
24: FUNC_2( 2 )
25: FUNC_2( 300 )
```

(b) Program with applied function inlining.

Figure 6.2: Example of function inlining.

driven inlining heuristics based on *random forests* outperform standard heuristics in terms of WCET. In their paper, code size increase was also taken into account when choosing functions for inlining.

Function inlining is a single-objective optimization problem, but in this section, we formulate it as a multiobjective problem with three objectives: WCET, code size, and energy consumption. We motivate why function inlining cannot optimize the objectives simultaneously and trade-offs between them must be considered.

Function inlining is a high-level compiler-based optimization: a compiler replaces a function call by the body of the function. Figure 6.2 shows an exemplary input program before and after applying function inlining. Figure 6.2(a) presents the original program with two functions func_1 and func_2 and four

```
1: function FUNC_2( int b )
2:     res = 0, i = 0
3:     Loop bound min 2 max 300.
4:     while i < b do
5:         res = res + 3 · i
6:         i = i + 1
7:     return res
8:
9: Inlined func_1.
10: res_func_1 = 2 · 1 + 5
11:
12: Inlined func_2.
13: res_func_2 = 0, i = 0
14: Loop bound min 40 max 40.
15: while i < 40 do
16:     res_func_2 = res_func_2 + 3 · i
17:     i = i + 1
18:
19: Remained original calls of func_2.
20: FUNC_2( 2 )
21: FUNC_2( 300 )
```

Figure 6.3: Program from Figure 6.2(b) with applied redundant path elimination and constant propagation.

function calls. The function func_1 is called once, whereas the function func_2 is called three times with different values of its input parameter b. The function func_2 contains a loop with a static loop bound depending on the input parameter b (Line 11), so as explained in Section 3.2, when computing the WCET of the original program, the static analyser aiT assumes the worst-case scenario for all function calls, i.e. it considers the loop bound equal to 300, which leads to an overestimation of the WCET in the case of the calls from Lines 19 and 20.

Figure 6.2(b) presents a program with two inlined function calls. Since the original program contains only one call of the function func_1 which is inlined, the compiler removes the original function from the code, whereas the function func_2 remains in the code due to the calls at Lines 24 and 25. By inlining a function call, the compiler stores function's input parameters to local variables, inserts the function body into the code, and removes the return instruction.

Function inlining enables more possibilities for subsequent optimizations like, e.g. redundant path elimination or constant propagation. Figure 6.3 shows a program when the compiler applied these additional optimizations to the program from Figure 6.2(b). E.g. at Lines 11–14 in Figure 6.2(b), the compiler prop-

agated the constant value of a variable a and eliminated an if-else condition due to $1 > 0$. It demonstrates that function inlining combined with other optimizations might decrease code size. By propagating the constant value of the variable b at Lines 17–19 in Figure 6.2(b), the compiler provides static analysers like aiT and EnergyAnalyser with a constant loop bound as shown at Line 14 in Figure 6.3. It tightens estimation of WCET and energy consumption, which is critical for hard real-time systems.

Function inlining may decrease WCET and energy consumption since it enables

- reduction of call and return instructions as well as of parameter handling;
- potentially smoother pipeline behaviour due to removed call and return instructions;
- new possibilities for subsequent optimizations;
- tighter WCET and energy consumption estimations.

The disadvantages of function inlining are the following:

- increasing code size due to duplicates of function bodies;
- possibly degraded energy consumption due to inserted additional local variables which increase register pressure.

We formulate function inlining as a binary-encoded problem with a decision space $X = [0, 1]^n$. The dimension n corresponds to the total number of function calls in a program. We define the coordinates of a search vector $\mathbf{x} \in X$ as follows:

$$x_i = \begin{cases} 1, & \text{if the function is to be inlined at a function call } i, \\ 0, & \text{otherwise.} \end{cases} \tag{6.13}$$

From the nature of function inlining, it is clear that code size contradicts WCET and energy consumption, but it is unclear whether WCET contradicts energy consumption. Figure 6.4 shows dependence between WCET and energy consumption when performing function inlining for four exemplary benchmarks: `adpcm`, `codecs_codrle1`, `codecs_dcodhuff`, and `gsm_encode`. The x-axis shows relative WCET and the y-axis presents relative energy consumption. 100 % represents the WCET and energy consumption of the original programs (without function inlining).

To generate data for the plots,

- for each benchmark, WCC randomly generates 100 search vectors as defined in Equation (6.13);

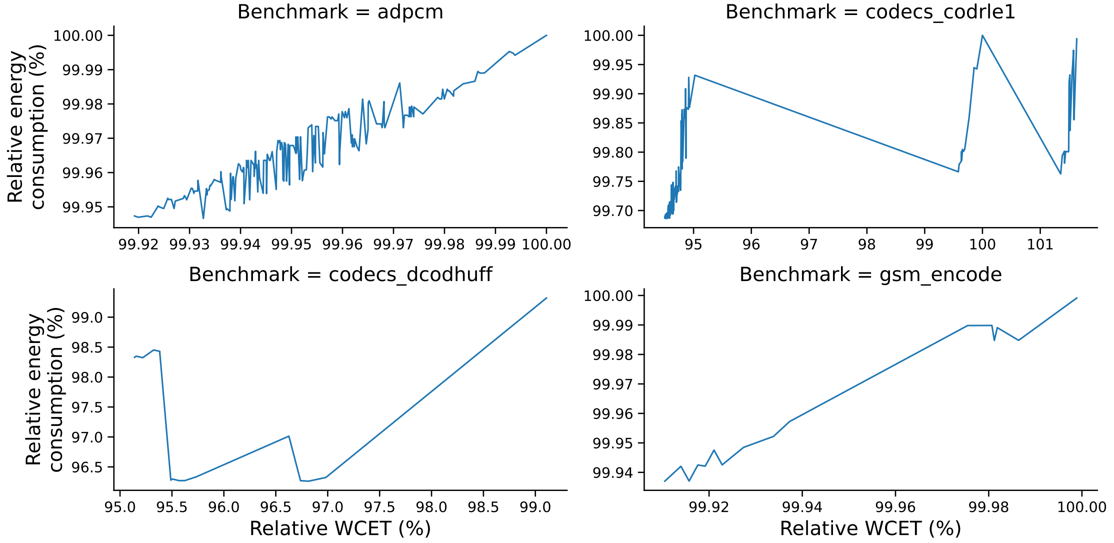


Figure 6.4: Dependence between relative WCET and energy consumption while performing function inlining. 100 % corresponds to the original WCET or energy consumption of a benchmark.

- for each search vector, WCC inlines function calls specified by the entries of the search vector set to 1 and evaluates the WCET and energy consumption of the resulting program by using the tools aiT and EnergyAnalyser.

Figure 6.4 shows a nonlinear dependence between WCET and energy consumption, so function inlining with the considered objectives is multiobjective. We showed dependence between WCET and energy consumption only for four random benchmarks. Since the dependence is nonlinear for these benchmarks, we cannot state that it is linear for other benchmarks, so we must assume that it is nonlinear for all benchmarks. We thus define a 3-dimensional objective function

$$\vec{f} = (\text{WCET},\ \text{Code Size},\ \text{Energy Consumption}) \tag{6.14}$$

and formulate an optimization problem as follows:

$$\min_{\mathbf{x} \in X} \vec{f}(\mathbf{x}) \ . \tag{6.15}$$

The formulated multiobjective function inlining problem is simple but in the next section, we show that even solving such a simple problem with evolutionary algorithms described in Section 4.2 arises issues crucial for multiobjective compiler-based optimizations. Namely, any evolutionary algorithm extensively explores a search space and evaluates all objectives of each search vector, but estimation of WCET and energy consumption is very time-consuming at compile

time, which leads to a drastic increase in compile time. In the next chapters, we propose some approaches to tackle these issues.

6.4 Evaluation

In this section, we evaluate BGDE3 and MBPOA in terms of solving the multiobjective function inlining problem. Section 6.4.1 describes experimental setups, Section 6.4.2 shows parameter tuning for the considered evolutionary algorithms, and Section 6.4.3 presents results regarding runtime and quality of solutions of the evolutionary algorithms.

6.4.1 Experimental Setup

For the purpose of evaluation, we use the WCC compiler described in Chapter 3 for the ARM Cortex-M0 microcontroller described in Section 2.2.1. We compute WCET and energy consumption by using AbsInt's aiT and EnergyAnalyser 20.10i described in Section 2.1.3 and code size by WCC. We run all evaluations on a server with specifications listed in Table 5.1.

We demonstrate our approach by considering the compiler-based optimization function inlining formulated in Section 6.3. It shows the most significant improvement of a final executable in combination with other optimizations, e.g. constant propagation and dead code elimination, so we perform all evaluations with the compiler optimization level O2. The Cortex-M0 microcontroller lacks a hardware floating-point unit, so we use the WCC software math library to tackle this issue. We consider functions of the floating-point library also as candidates for inlining.

We evaluate the approach by using benchmarks from the publicly available test suites PolyBench, MediaBench, MRTC, DSPstone, and UTDSP with annotated loop bounds from the TACLeBench project [Fal+16]. We consider benchmarks with the dimension of the search space (the dimension of the search space corresponds to the total number of function calls that can be inlined) greater than 5: to solve a problem with a search space dimension less than or equal to 5, brute force search requires only at most $2^5 = 32$ evaluations of each objective. All considered benchmarks are listed in Appendix B.

We consider a function call as a candidate for inlining if

- the function is not recursive;
- the argument list of the function does not contain a variable number of arguments;
- the function does not declare any static symbols.

WCC supports a standard function inlining which is invoked at the compiler optimization level O2. The standard function inlining follows a simple strategy: it inlines a function if

1. it satisfies the three assumptions listed above;

2. it consists of fewer than 30 expressions[1] to prevent inlining of large functions, which leads to a drastic increase in code size.

To be consistent with the WCC's standard function inlining, in our evaluation, we also ignore functions with more than 30 expressions. It should be mentioned that even without these functions, the search spaces of the benchmarks are large and challenging for evolutionary algorithms.

We solve the multiobjective function inlining problem defined in Equation (6.15) by utilizing BGDE3 and MBPOA algorithms from Section 6.2.

6.4.2 Parameter tuning

The performance of any evolutionary algorithm depends on values of its control parameters, so we tune them as described on Page 78 to identify the best parameters' values for BGDE3 and MBPOA. We compare the quality of approximated Pareto fronts by using nondominance ratio NR and coverage C defined in Equations (4.8) and (4.9), respectively.

Both nondominance ratio and coverage estimate the quality of an approximated Pareto front found by an evolutionary algorithm by comparing it to the true Pareto front of a problem. For most real-world problems and in our case as well, the true Pareto fronts are unknown, so in order to compute the quality indicators, we substitute the true Pareto front PF by a reference set PF_R. To find the reference set PF_R, we pool together all available approximated Pareto fronts (in this chapter, all Pareto fronts produced by BGDE3 and MBPOA with all different parameters) and denote by PF_R a set of nondominated points of the pooled set. PF_R represents the best approximation of the true Pareto front found so far.

We consider the following values of MBPOA's crossover parameter:

- $CR \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$.

BGDE3 has three control parameters: crossover parameter CR, scaling factor F, and bandwidth b. We compare the following values of them, similar to Wang et al. [Wan+12]:

[1]Expression is a component of ICD-C to model code to be compiled. E.g. $b = 4$ is modelled by ICD-C as follows: "=" is an assignment expression that contains two subexpressions, a symbol expression b and an integer constant expression 4. For more details about the ICD-C internal structure, we refer to the ICD-C developer manual [Dor09].

- crossover $CR \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$;
- scaling factor $F \in \{0.2, 0.6, 1.0, 5.0, 10.0\}$;
- bandwidth $b \in \{2, 6, 10, 50, 100, 500, 1000\}$.

We randomly generate an initial population of the algorithms and repeat each experiment with fixed control parameters 10 times. We evaluate MBPOA with nine possible crossover parameters and BGDE3 with 175 combinations of the control parameters, so for each benchmark, we ran MBPOA $9 \cdot 10 = 90$ times, whereas BGDE3 $175 \cdot 10 = 1,750$ times. Since $1,750$ runs of BGDE3 is already a lot, we consider fewer values of the crossover parameter in the case of BGDE3 compared to MBPOA as can be seen in the lists above.

A population size and the number of generations are problem-specific parameters, and setting them to large numbers allows better exploration of a search space, but one should set them keeping in mind that large values might lead to a long evolutionary process.

The population size is a user-defined parameter that can also be tuned but in our evaluations, we fix it to 50 for two reasons:

- Chen et al. [Che+12] showed that a large population may lead an evolutionary algorithm to a local optimum;
- Georgioudakis and Plevris [GP20] stated that the population size should be set to a value between 30 and 50 for engineering problems.

The number of generations is used as a stopping criterion in BGDE3 and MBPOA. This parameter should also be tuned but we have to limit it to 30 when solving the function inlining problem, since a large number of generations leads to a too long evolutionary process due to time-consuming WCET and energy consumption estimations[2].

E.g. Figure 6.5 presents runtime required to estimate WCET and energy consumption 1,500 times by using the static analysers aiT and EnergyAnalyser (see Section 2.1) for all considered benchmarks; the x-axis shows the benchmarks with their dimensions of the search spaces in parentheses, whereas the y-axis shows runtime in hours.

Many considered benchmarks have large search spaces whose dimensions are greater than 80. If the population size is set to 50, the number of generations is set to 30, and no archive is used to store previously seen individuals (with already evaluated objectives), WCET and energy consumption must be estimated 1,500 times. This means that, e.g. for the benchmark `3mm`, one run of an evolutionary

[2]WCET and energy consumption, which are objectives of the considered function inlining problem, require time-consuming analyses (e.g. control flow or microarchitectural analyses) as described in Section 2.1.

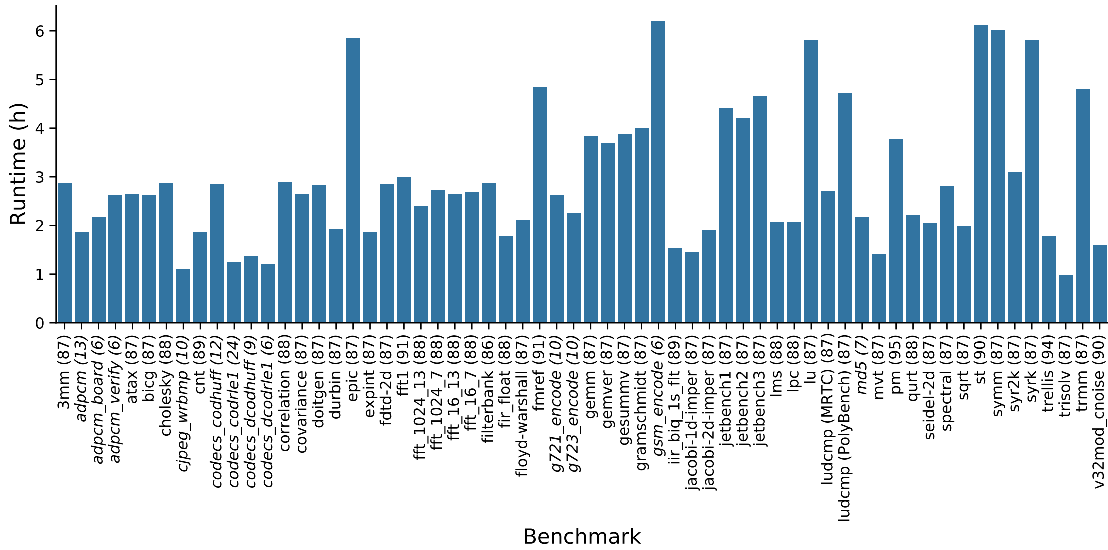


Figure 6.5: Runtime required to estimate WCET and energy consumption 1,500 times by using the static analysers aiT and EnergyAnalyser. The dimensions of the search spaces are specified in parentheses next to the benchmarks' names.

algorithm might take more than 2 h. While tuning the parameters, we run MBPOA 90 times and BGDE3 1,750 times for each benchmark, so the parameter tuning for `3mm` might take $(90 + 1,750) \cdot 2 = 3,680\ \mathrm{h}$ or $153\ \mathrm{days}$.

Limiting the number of generations might be not enough to terminate the evolutionary algorithms within a feasible time frame, so we also set a time limit to 72 h for each benchmark. We store the objectives of explored search vectors in an archive to avoid redundant estimations of WCET and energy consumption.

The limitation of 72 h left only 12 out of 62 benchmarks in evaluation. The successful benchmarks are benchmarks with search space dimensions less than 30. These benchmarks are shown in italics in Figure 6.5, e.g. `adpcm`, `adpcm_board`, `adpcm_verify`, etc. These benchmarks have quite small search spaces and the archive allowed to finish evaluation within 72 h.

To find the best values of the control parameters for both algorithms, we analyse the dependency of the quality indicators from the control parameters. Figure 6.6 presents the mean and 95 % confidence interval for nondominance ratio and coverage over all successful benchmarks when solving the inlining problem with MBPOA; the x-axis shows crossover values, whereas the y-axis shows nondominance ratio (the left plot) and coverage (the right plot); the blue solid line presents the mean of the quality indicators over all benchmarks, whereas the shadowed area shows the confidence interval. We observe the following behaviour of MBPOA:

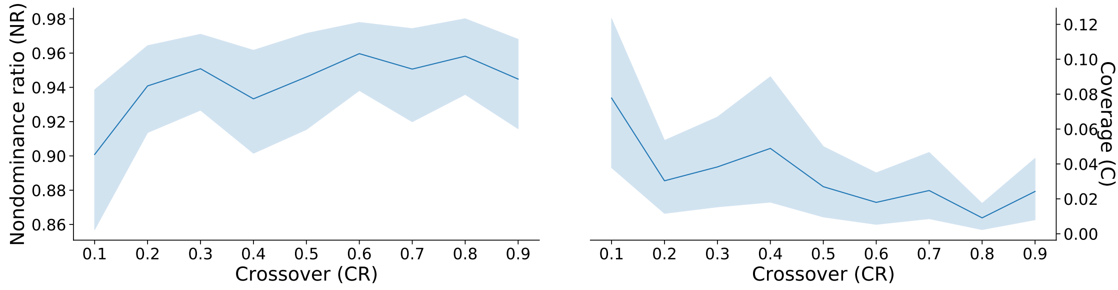


Figure 6.6: The mean and 95 % confidence interval of nondominance ratio (left) and coverage (right) over all successful benchmarks when solving the inlining problem with MBPOA. Nondominance ratio is to be maximized, coverage is to be minimized.

- MBPOA is stable to changes of crossover values: for any crossover value, with 95 % confidence, nondominance ratio is greater than 0.94 (the best value is 1), and coverage is less than 0.05 (the best value is 0). This means that in general, without tuning the parameter but just setting it to any value from the interval $(0, 1)$, the algorithm already produces a high-quality Pareto front;
- for the crossover parameter equal to 0.1 ($CR = 0.1$), MBPOA shows the worst quality of found Pareto fronts since the average nondominance is 0.96 and the average coverage is 0.03;
- according to the results of nondominance ratio, the algorithm produces the best Pareto fronts with the values 0.6 and 0.8 (the average nondominance ratio is 0.98), but according to coverage, the algorithm shows the best results with the crossover parameter equal to 0.8 (the average coverage is 0.003).

Similar to Figure 6.6, Figure 6.7 shows the mean and 95 % confidence interval for the considered quality indicators when solving the problem with BGDE3; the rows of the figure correspond to the BGDE3's control parameters: crossover, scaling factor, and bandwidth, whereas the columns correspond to the quality indicators: nondominance ratio (left column) and coverage (right column). The x-axes show control parameters' values and the y-axes show quality indicators' values; the blue solid line shows the mean of the quality indicators over all benchmarks and the shadowed area presents the confidence interval. We observe the following dependencies between the control parameters and the quality indicators for BGDE3:

- similar to MBPOA, BGDE3 is stable to changes of the control parameters: for any values of the control parameters, with 95 % confidence, nondominance ratio is greater than 0.96 and coverage is less than 0.025. These

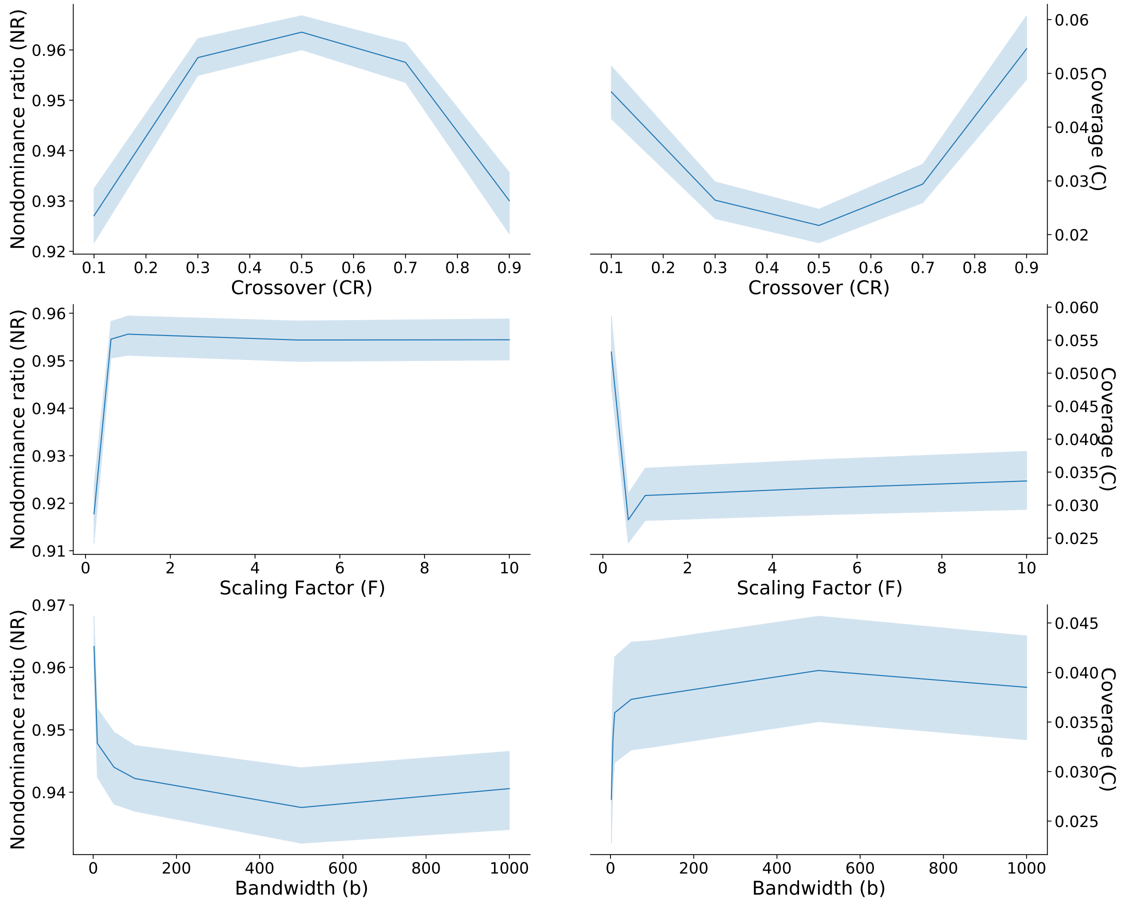


Figure 6.7: The mean and 95 % confidence interval for nondominance ratio (left) and coverage (right) over all considered benchmarks when solving the inlining problem with BGDE3. Each row corresponds to the control parameters: crossover, scaling factor, and bandwidth. Nondominance ratio is to be maximized, coverage is to be minimized.

boundary values of the quality indicators for BGDE3 are slightly better than for MBPOA;

- according to both quality indicators, the best *crossover* value for BGDE3 is 0.5; in general, the crossover's plots are symmetric: BGDE3 shows worse results for small ($CR = 0.1$) and large ($CR = 0.9$) values of the crossover parameter, i.e. a low nondominance ratio and high coverage, and it achieves the best results with the middle value $CR = 0.5$;

- the quality of resulting Pareto fronts becomes stable for *scaling factors* greater than 1.0. The lowest average nondominance ratio is equal to 0.966 and the highest average coverage is equal to 0.022, which are observed

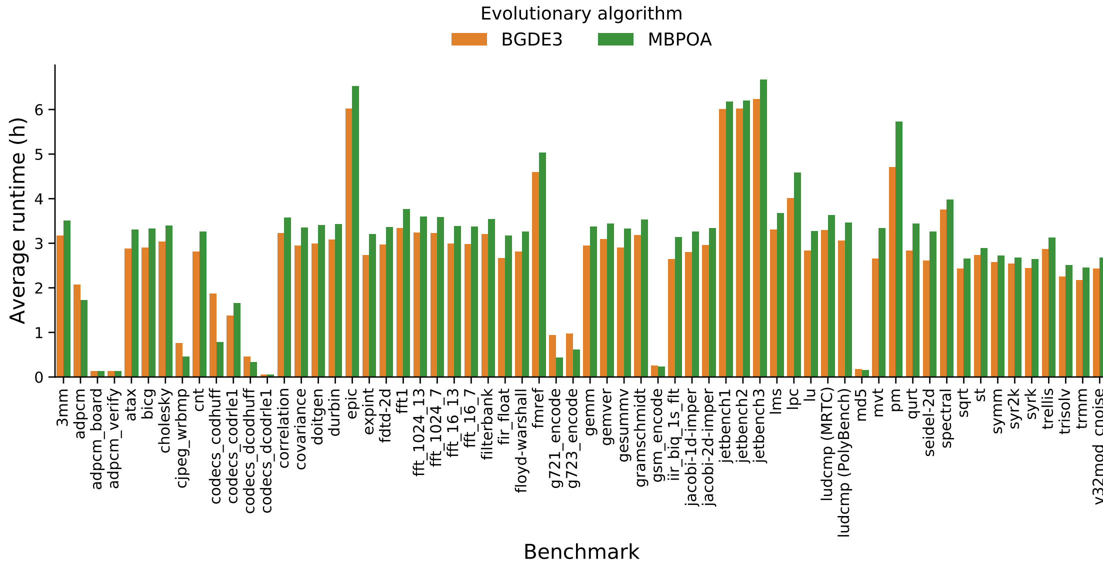


Figure 6.8: Average runtime of BGDE3 and MBPOA.

for 0.2 (the smallest evaluated value of scaling factor). An increase of scaling factor improves (increases) nondominance ratio, and starting from the value of 0.6, the quality indicator remains almost unchanged. Coverage behaves slightly differently: an increase of scaling factor from 0.2 to 0.6 improves (decreases) coverage but starting from scaling factor equal to 1.0, coverage is slightly degraded (increases). For these reasons, the best scaling factor for BGDE3 is 0.6;

- an increase of *bandwidth* degrades the quality indicators: nondominance ratio decreases and coverage increases, so BGDE3 shows the best results with bandwidth equal to 2, which is the smallest tested value.

To summarize, both algorithms result in a high nondominance ratio and low coverage for all considered control parameters' values, but they achieved the best results with the following values:

- MBPOA: $CR = 0.8$;
- BGDE3: $CR = 0.5$, $F = 0.6$, $b = 2$.

6.4.3 Comparison of BGDE3 and MBPOA

From now on, we fix the control parameters to the best values found in the previous section and present results only for the fixed parameters.

Running the algorithms with the fixed control parameters, we remove the time limit of 72 h set during parameter tuning. Figure 6.8 presents the average runtime over 10 runs of BGDE3 (orange) and MBPOA (green): the x-axis lists the benchmarks, the y-axis shows the average runtime. In most cases, BGDE3 was faster than MBPOA because it required fewer evaluations of WCET and energy consumption, i.e. BGDE3 resulted in more search vectors with previously evaluated objectives and stored into an archive than MBPOA. E.g. for the benchmark `3mm`, the average runtime of BGDE3 and MBPOA are 3.2 h and 3.5 h, respectively. For the most time-consuming benchmark `jetbench3`, MBPOA required 6.7 h and BGDE3 required 6.2 h. For the least time-consuming benchmark `codecs_dcodrle1`, MBPOA required 3.3 min and BGDE3 required 3.1 min.

Figure 6.9 shows the quality indicators of BGDE3 (orange) and MBPOA (green) with the fixed control parameters; the x-axis lists the benchmarks and the y-axis presents the quality indicators: nondominance ratio (the top plot) and coverage (the bottom plot); the bars show the mean of the quality indicators when repeating experiments with the fixed control parameters 10 times for each benchmark. For many benchmarks like, e.g. `3mm`, `atax`, `bicg`, etc., BGDE3 resulted in $\mathsf{NR} = 0$ and $\mathsf{C} = 1$, i.e. in 10 runs, the algorithm failed to find any nondominated point from a reference set $\mathsf{PF_R}$. But for these benchmarks, the results determined by MBPOA are also poor, a small nondominance ratio (the average value close to 0.1) and large coverage (the average value close to 0.9) mean that the algorithm found nondominated points from the set $\mathsf{PF_R}$ but only in some runs of the algorithm. For all these benchmarks, the dimension of the search space is larger than 30, i.e. these benchmarks were not used during parameter tuning, since the algorithms exceeded the time limit of 72 h. This might mean that parameters tuned for the benchmarks with the dimension of the search space less than 30 are not suitable for large benchmarks.

For 12 benchmarks (shown in italics) with search space dimensions less than 30, we observed the following results:

- BGDE3 outperformed MBPOA in the case of four benchmarks: `adpcm`, `cjpeg_wrbmp`, `codecs_dcodhuff`, and `g721_encode`;
- MBPOA outperformed BGDE3 in the case of four benchmarks: `codecs_codrle1`, `codecs_dcodrle1`, `g723_encode`, and `gsm_encode`;
- both algorithms returned results of the same quality in the case of four benchmarks: `adpcm_board`, `adpcm_verify`, `codecs_codhuff`, and `md5`.

From the results of Figures 6.8 and 6.9, we cannot clearly state that one algorithm is preferable over the other, since

- both algorithms returned solutions of almost the same quality;
- the runtime of both algorithms is of the same order of magnitude.

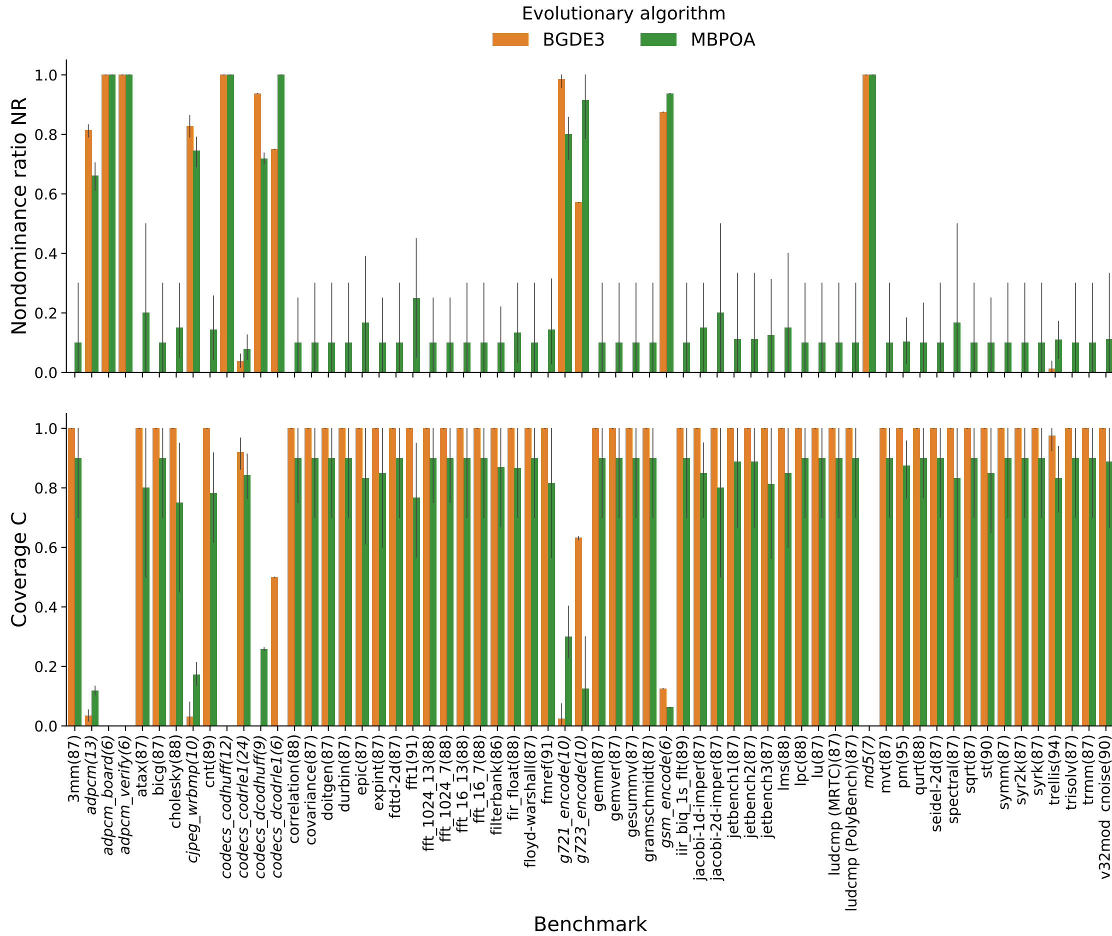


Figure 6.9: Quality indicators of BGDE3 and MBPOA with the following fixed control parameters: MBPOA: crossover CR=0.8; BGDE3: crossover CR=0.5, scaling factor F=0.6, and bandwidth b=2. Nondominance ratio is to be maximized and coverage is to be minimized. The dimensions of the search spaces are specified in parentheses next to the benchmarks' names.

In general, we would suggest using MBPOA, although BGDE3 was faster than MBPOA, since MBPOA found solutions of a little bit higher quality if the dimension of the search space is greater than 30, i.e. it might be less sensitive to choice of parameter values.

Next, we demonstrate solutions returned by the algorithms for two benchmarks: `3mm` and `cjpeg_wrbmp`. Figure 6.10 shows all solutions found by BGDE3 (orange) and MBPOA (green) for `3mm` in 10 runs. The x-axis shows relative WCET, the y-axis – relative energy consumption, and the z-axis – relative code size. 100% corresponds to the benchmark's original WCET, energy consumption, and code

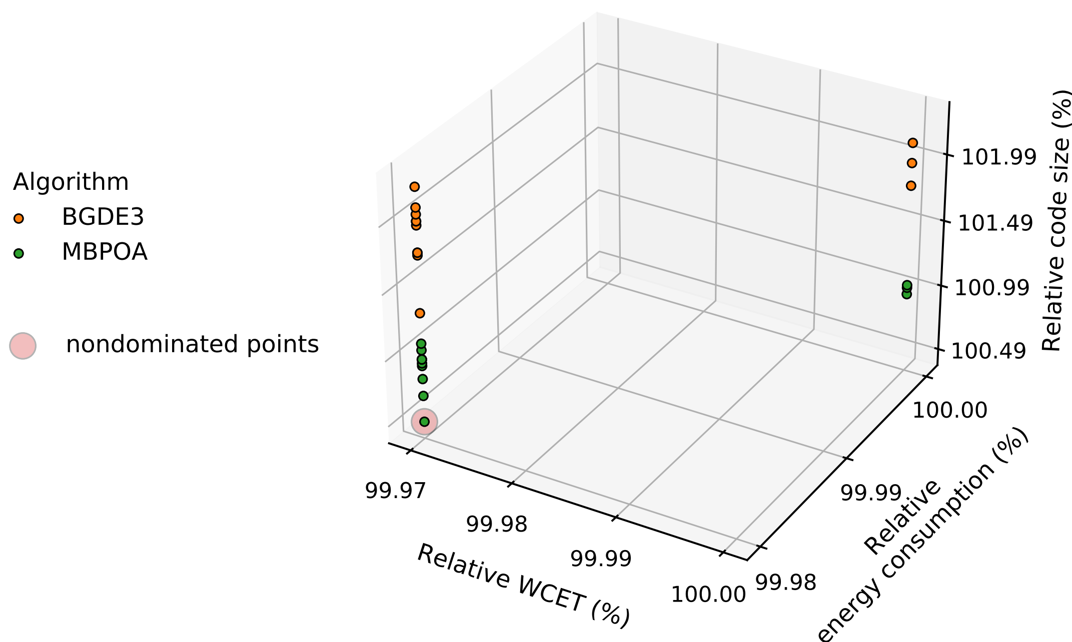


Figure 6.10: Solutions returned by the algorithms BGDE3 and MBPOA for the benchmark `3mm` in 10 runs. 100% corresponds to the original WCET, energy consumption, and code size.

size, which are 1,330,766 cycles, 2.62×10^{11} fj, and 812 bytes, respectively. For this benchmark, function inlining was able to decrease WCET by around 0.03 % and energy consumption by around 0.02 %, while code size was increased by up to 2 %. One nondominated point shown in red was found in 1 out of 10 runs of MBPOA. Both algorithms found two groups of solutions with the same WCET and energy consumption and different values of code size. MBPOA found a solution shown in red that dominates all other solutions. The code size of solutions found by MBPOA is smaller than the code size of solutions found by BGDE3. So MBPOA spent more time exploring the search space according to Figure 6.8 but found solutions of higher quality.

Figure 6.11 presents an example of approximated Pareto fronts found by the evolutionary algorithms for the benchmark `cjpeg_wrbmp`; each algorithm was run 10 times. The x-axis shows relative WCET, the y-axis represents relative energy consumption, and the z-axis represents relative code size. 100% corresponds to the original WCET, energy consumption, and code size, which are 438,467 cycles, 6.94×10^{11} fj, and 598 bytes, respectively. The presented approximated Pareto fronts were found by BGDE3 (orange) and MBPOA (green). Nondominated points are shown in red. For this benchmark, function inlining

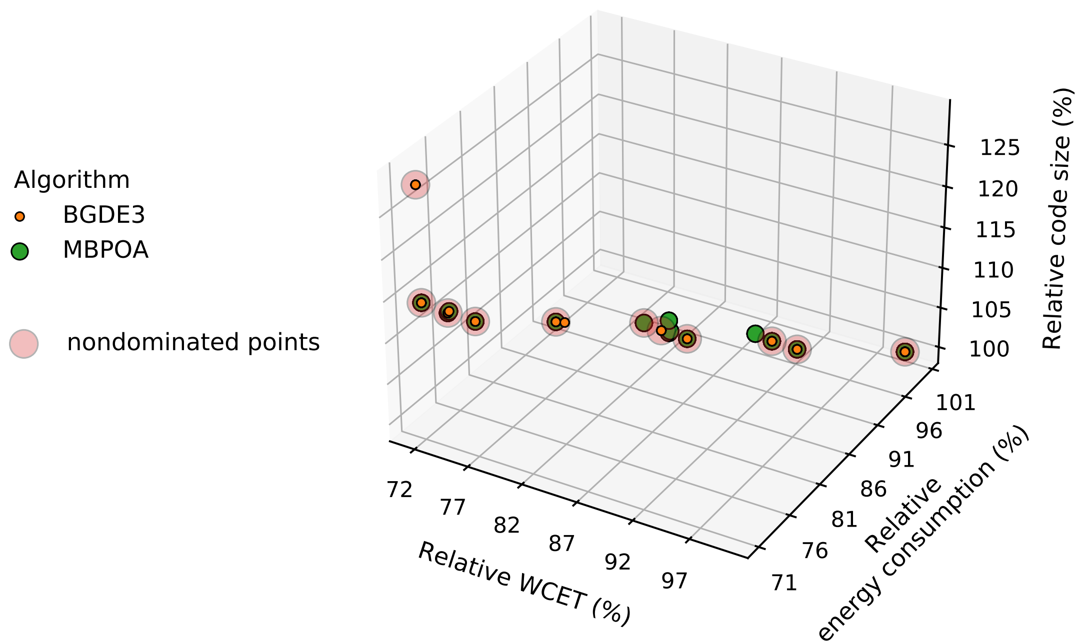


Figure 6.11: Solutions returned by the algorithms BGDE3 and MBPOA for the benchmark `cjpeg_wrbmp` in 10 runs. 100% corresponds to the original WCET, energy consumption, and code size.

decreased WCET and energy consumption up to 73 % but it increased code size up to 125 %. BGDE3 returned 12 solutions, 10 of them are nondominated, and MBPOA returned 14 solutions, 9 of them are nondominated. Since BGDE3 found one more nondominated point than MBPOA, BGDE3's quality indicator results in Figure 6.9 are better than in the case of MBPOA for this benchmark.

6.5 Conclusion

In this chapter, we focused on multiobjective compiler-based optimizations that must be treated as multiobjective problems and should not be reformulated as single-objective problems with constraints. We aimed to find a set of trade-offs between contradicting objectives.

To demonstrate the difficulties of solving multiobjective compiler-based optimizations, we utilized evolutionary algorithms to solve a multiobjective function inlining problem. We tuned user-defined control parameters of considered evolutionary algorithms to achieve approximated Pareto fronts of high quality. It turned out that for the considered problem, the evaluated evolutionary algorithms are stable to changes of control parameters, i.e. changes in the control pa-

rameters lead to only minor changes in the quality of Pareto fronts returned by the algorithms.

We showed that for many benchmarks, considering WCET and energy consumption as objectives leads to a time-consuming solution process for two reasons:

- any evolutionary algorithm extensively evaluates objectives in every newly found point of a search space;
- estimation of WCET and energy consumption at compile time is time-consuming.

To tackle these issues, in the next chapters, we present approaches that substitute time-consuming estimations of WCET and energy consumption with cheaper predictions (see Chapter 7) and reduce the search space of the original problem to provide an evolutionary algorithm with a smaller search space that needs to be explored (see Chapter 8).

7 Predicting Objectives at Compile Time

Chapter 6 exploits evolutionary algorithms to solve a multiobjective optimization problem at compile time. One can easily use evolutionary algorithms to solve complex multiobjective problems, but the algorithms require many evaluations of objectives to find a good approximation of Pareto front due to their stochastic nature. According to Coello et al. [Coe+19], this limits usage of evolutionary algorithms to solve real-world problems where

- objective functions are expensive to evaluate;
- objective functions lack an algebraic representation;
- financial constraints limit the number of evaluations of objective functions.

In the previous chapter, we showed that considering WCET and energy consumption as objectives drastically increases compilation time: evolutionary algorithms extensively evaluate objectives to explore the search space of a problem, whereas estimations of WCET and energy consumption require time-consuming analyses.

To speed up compilation process, in this chapter, we follow a methodology of surrogate models that approximate an original problem by using data gathered from evaluations of objective functions. Many methods based on surrogate models are presented in the literature, e.g. Arias-Montano, Coello, and Mezura-Montes [AMCMM12], Tabatabaei et al. [Tab+15], or Diaz-Manriquez et al. [DM+16]. We build a surrogate model based on machine learning at compile time to quickly predict WCET and energy consumption instead of costly estimating them (by a static analyser) while running an evolutionary algorithm. We do not predict code size since it can be easily computed by the WCC compiler used in the thesis.

One of the most important requirements of estimated WCET is safeness: an estimated WCET must be greater or equal to the exact WCET of a program as shown in Figure 2.1. We aim to predict WCET and energy consumption as precise as possible but we do not require predicted WCETs to be safe in contrast to estimated WCETs for two reasons:

- we use predictions to guide an evolutionary algorithm to promising regions of a search space;

- we do not use predictions to validate timing requirements of hard real-time systems, instead we utilize a static analyser to safely estimate the WCET of final solutions.

Altenbernd et al. [Alt+16], Bonenfant et al. [Bon+17], Huybrechts, Mercelis, and Hellinckx [HMH18], and others proposed early stage approaches to predict WCET by using machine learning. The approaches follow three main steps:

- they build a prediction model by extracting features of an input program like, e.g. the number of arithmetic operations or function calls, and use them as variables in the prediction model;

- they estimate the WCETs of training programs for a specific target architecture;

- they fit the model on the training programs to find dependence between the features and estimated WCET.

Such approaches result in prediction models that predict WCET for a specific target architecture. In our approach, the WCC compiler builds a prediction model for each program independently. Our prediction model is unaware of any specific features of a program and target architecture and relies only on the search and objective spaces of an optimization problem.

The goal of this chapter is to choose the most promising machine learning approach among many widely used methods and to show that it can predict WCET and energy consumption precisely enough to solve a multiobjective problem at compile time. We also tune the parameters of considered machine learning techniques to improve the quality of predictions.

We presented the prediction model described in this chapter at the International Conference on Machine Learning, Optimization, and Data Science (LOD) in 2021 [MF21b].

The chapter is organized as follows: Section 7.1 presents related work; Section 7.2 introduces the model to predict WCET and energy consumption at compile time; Section 7.3 describes integration of the prediction model into evolutionary algorithms; Section 7.4 presents evaluation results; Section 7.5 gives a conclusion.

7.1 Related Work

This section presents the related work concerning the following topics: surrogate models during evolutionary algorithm executions, prediction of WCET and energy consumption.

Surrogate models. Machine learning approaches are often used to build surrogate models to solve expensive multiobjective problems, e.g. Wismans, Berkum, and Bliemer [WBB13], Esfe, Hajmohammad, and Wongwises [EHW17], or Mazumdar et al. [Maz+19].

Many researchers presented surrogate models based on neural networks, e.g. Martínez and Coello [MC13], or Palar and Shimoyama [PS17]. Azzouz, Bechikh, and Said [ABS14] used neural networks to build a surrogate model for the airfoil shape problem presented by Szőllős, Šmíd, and Hájek [SŠH09]. They utilized it within the Indicator-Based Evolutionary Algorithm (IBEA) to minimize the number of expensive evaluations of the objectives. The model estimated the hypervolume – introduced by Zitzler and Thiele [ZT98b] – of each offspring, and the authors evaluated only the exact objectives of the fittest individuals. The surrogate model sped up IBEA from 4.550 s to 1.250 s.

Other proposed surrogate models use Gaussian process to improve evolutionary algorithms, e.g. Zhang et al. [Zha+10] or Bradford, Schweidtmann, and Lapkin [BSL18]. The latter one combined Gaussian processes with NSGA-II [Deb+02a]: for each objective, the authors built an individual Gaussian process model and sampled the objective by using spectral sampling introduced by Hernández-Lobato, Hoffman, and Ghahramani [HLHG14]. NSGA-II solved a multiobjective problem by using sampled objectives. The proposed method outperformed the original NSGA-II in terms of hypervolume indicator for 8 out of 9 test problems with the number of objective evaluations limited to 150.

Researchers have also used support vector machines to reduce the number of objective evaluations, e.g. Zheng, Zhang, and Guo [ZZG09] or Ribeiro and Reynoso-Meza [RRM18]. Yun, Yoon, and Nakayama [YYN08] followed the concept of multiobjective optimization based on meta-modelling: the authors built a support vector regression model for every single objective using some sample data, solved the problem by using predicted objectives and SPEA2 [ZLT01], and chose a new sample to rebuild the prediction models. For evaluated benchmarks, the authors reduced the number of objective evaluations by $1/100$ to $1/10$ of the original SPEA2.

Since it is unclear which machine learning method we should use in our case, in this chapter, we compare different machine learning techniques in terms of predicting WCET and energy consumption at compile time and choose the best one to solve the multiobjective function inlining problem formulated in Section 6.3.

Prediction of WCET. Very few approaches are present in the literature to predict WCET. While developing an embedded system, a system designer often requires preliminary timing estimates, so early stage approaches predict WCET by building a prediction model based on source code features.

Altenbernd et al. [Alt+16] presented a linear model to predict WCET at the source code level. The authors ran training programs on a simulator or real hardware to measure their execution times. They translated an input C code into an intermediate format consisting of virtual instructions, e.g. arithmetic operations, and emulated the code to record the execution counts of instructions. The proposed prediction model is a linear timing model where the execution counts are the coefficients of unknowns. To fit the model, the authors used the least square method described by Gauß in 1809 [Gau09]. For tested benchmarks, evaluation showed the average deviation of 8 % for the ARM7 target architecture.

Bonenfant et al. [Bon+17] described an approach to predict WCET at the C source code level by using static analysis of a source code as well as a machine learning technique. The prediction model relied on a worst-case event count analysis that counts certain events of the input code, e.g. the maximal number of function calls. Finally, a machine learning algorithm based on the extracted counts of attributes builds the WCET prediction model. The paper presents no numerical results.

Huybrechts, Mercelis, and Hellinckx et al. [HMH18] proposed a hybrid approach to predict WCET by using machine learning. The authors divided an input code into blocks to extract attributes like the number of logic operations, global variables, array accesses, etc. To predict WCET, the authors compared eight regression methods with the extracted features considered as input variables. The support vector regression with a linear kernel showed the best results on tested benchmarks with the mean relative error of 27.3 %.

Early stage approaches to predict WCET build a prediction model that is fast and can be easily fitted by using some training programs, but they rely on source code features. In contrast, we aim to build a model that relies on the search space of a problem and misses explicit knowledge about source code or hardware features, so it can be fitted for any compiler-based optimization (source and assembly code level optimizations).

Energy consumption. A hot topic in parallel execution is to find trade-offs between the performance and energy consumption of multi-core systems. The approaches often utilize machine learning techniques to predict performance and energy consumption while configuring the number of used cores and their frequencies.

Mishra et al. [Mis+15] presented LEO, a probabilistic graphical model-based learning system, to estimate online the power consumption and performance of an application. The authors used directed graphical models to exploit conditional dependence between system configurations, e.g. the number of cores or memory controllers. They modelled the objectives – power consumption and performance – as normally distributed random variables with unknown

mean and standard deviation. For 25 tested applications, LEO achieved greater than 97 % accuracy while estimating the objectives.

Sensi, Torquati, and Danelutto [STD16] aimed to derive on-the-fly the performance and power consumption of an application in a multi-core system. The authors utilized linear regression to predict service time and power consumption depending on the number of used cores and their frequencies.

Demetrios et al. [Dem+20] obtained a power model for a heterogeneous multiprocessor platform by representing the power consumption of two clusters: big cluster – consisting of higher performance cores consuming more power – and little cluster – consisting of slower energy-efficient cores. The search space of the model included the number of processing cores for each cluster and their operating frequency. The authors built two independent models for performance and power consumption by using nonlinear regression. Between predicted and measured values for eight evaluated applications, performance and power consumption absolute percentage errors were 5.53 % and 22.30 % on average, respectively.

In contrast to Mishra et al. [Mis+15], Sensi, Torquati, and Danelutto [STD16], and Demetrios et al. [Dem+20], we build and utilize a prediction model at compile time; our model relies on the search space of a compiler-based optimization and is independent of any hardware specifications.

7.2 Prediction Model

The previous section discusses that machine learning techniques are strong mechanisms that allow building a surrogate model to speed up evolutionary algorithms. They have been also used to predict WCET at early stages of system development and the energy consumption of multi-core systems. So, in this thesis, we propose a prediction model based on machine learning that predicts WCET and energy consumption at compile time[1].

Multi-output learning predicts simultaneously multiple outputs, but it requires a large number of samples to build a prediction model according to Xu et al. [Xu+19]. Since for each training sample, we must perform time-consuming WCET and energy consumption analyses, such approaches become infeasible for us. We build two independent single-output prediction models for WCET and energy consumption, which require fewer training samples than a multi-output model.

We denote by $X \subset \mathbb{R}^n$ an input (search) space of a problem and by $Y \subset \mathbb{R}$ an objective space representing WCET or energy consumption. We call *features* the coordinates of a vector $\mathbf{x} \in X$. E.g. for the function inlining problem formulated in Section 6.3, the input space X in this chapter coincides with the search space X of the problem from Chapter 6.

[1] We do not predict code size since WCC can easily compute it.

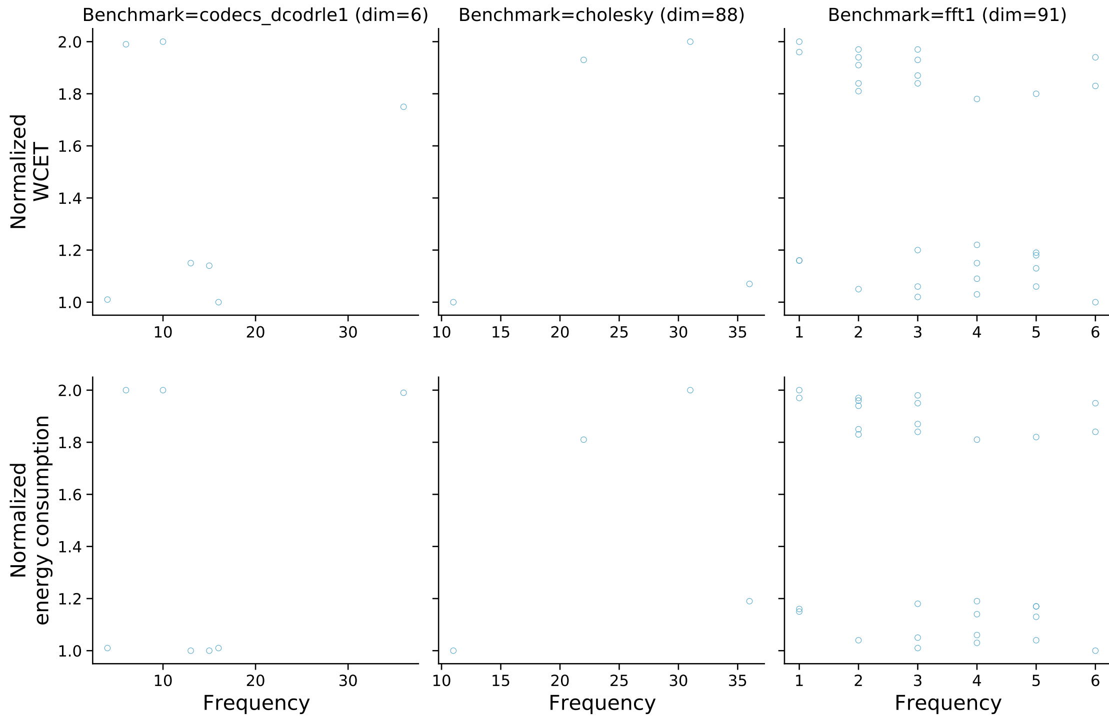


Figure 7.1: Objectives' frequency while performing function inlining for three exemplary benchmarks. For each benchmark, the static analysers aiT and EnergyAnalyser computed the objectives for 100 random samples. The dimensions of the search spaces are denoted by $\mathtt{dim}$. The objectives were normalized to lie in the interval $[1, 2]$ for the sake of legibility.

At compile time, we aim to predict $y \in Y$ at a given $\mathbf{x} \in X$ as accurately as possible. In other words, the goal is to identify a function that maps the input $\mathbf{x}$ to the output y. The formulated problem is a supervised machine learning problem, Section 4.3 gives details on machine learning tasks.

Any supervised machine learning algorithm seeks a function $g : X \to Y$ such that

$$y = g(\mathbf{x}) + \epsilon, \tag{7.1}$$

where ϵ is a noise. The algorithm selects g from a space of possible functions, e.g. a space of linear functions, by using a training set $P \subset X \times Y$. Supervised learning tasks are divided into two classes: regression and classification. In regression problems, objective spaces are continuous, whereas in classification problems, they are discrete.

WCET and energy consumption are, in general, continuous measures, i.e. a regression problem should be solved. But, e.g. while performing function inlining described in Section 6.3, we observe only a limited number of unique val-

ues for the objectives as shown in Figure 7.1. The figure presents the frequency of the unique values of observed WCET (the top row) and energy consumption (the bottom row) for 100 randomly generated samples; the x-axis represents frequency, the y-axis shows normalized WCET and energy consumption computed by the static analysers aiT [Abs22] and EnergyAnalyser [Tea]. The objectives are normalized to lie in the interval $[1, 2]$ for the sake of legibility. The figure presents the results for three benchmarks which are representative for all evaluated benchmarks: `codecs_dcodrle1` (the first column), `cholesky` (the second column), and `fft1` (the third column); Appendix D contains the results for all evaluated benchmarks. In the figure, we specified the dimensions (dim) of the search spaces next to the benchmarks' names.

For all benchmarks, we observed one of the following scenarios:

- similar to the benchmark `codecs_dcodrle1`: a small search space and some unique objective values with high frequency (greater than 10);
- similar to the benchmark `cholesky`: a large search space and a few unique values with high frequency;
- similar to the benchmark `fft1`: a large search space and many unique values with low frequency.

For many evaluated benchmarks, we observed the second scenario, i.e. a limited number of unique values for the objectives was observed. This motivated us to formulate and solve a classification problem instead of a regression problem. In general, we would expect the same behaviour for many compiler-based optimizations like, e.g. function specialization or loop unrolling, since the last step of WCET or energy consumption estimations (see Section 2.1.3) is a path analysis which identifies a path with the largest values of the objectives. When optimizations improve the objectives outside the critical path, WCET or energy consumption might be kept unchanged.

Section 7.2.1 describes shortly classifiers compared in the thesis, for more details, we refer to Hastie, Tibshirani, and Friedman [HTF09]. Section 7.2.2 presents an algorithm to choose the best classifier among the considered ones. Section 7.2.3 describes an algorithm to build a prediction model for a program to be compiled. Section 7.2.4 presents integration of the prediction model within the WCC compiler.

7.2.1 Classification

In this section, we shortly describe classifiers considered in this thesis; in the next section, we present an algorithm to choose the best classifier for a certain problem.

We consider the following most well-known classification methods [HTF09], which are widely used in practice:

1. logistic regression;
2. nearest neighbours classifier;
3. Support Vector Classifier (SVC) with linear kernels;
4. SVC with Radial Basis Function (RBF) kernels;
5. Gaussian process classifier;
6. decision tree classifier;
7. random forest classifier;
8. AdaBoost classifier based on decision trees;
9. naive Bayes classifier for Bernoulli models.

Logistic regression. It is a linear classification model that models the output for an input vector $\mathbf{x} \in X$ as follows [HTF09]:

$$y = g(\mathbf{w} \cdot \mathbf{x} + c), \tag{7.2}$$

where $\mathbf{w}$ represents a vector of weights, c is a constant, and the function $g : \mathbb{R} \to Y$ maps $\mathbf{w} \cdot \mathbf{x}$ to the output (class) $y \in Y$. Originally, logistic regression deals with binary classification – i.e. $y \in \{0, 1\}$ – then the function g is often a *threshold function*

$$g(t) = \begin{cases} 1, & \text{if } p(t) \geqslant C, \\ 0, & \text{if } p(t) < C, \end{cases} \tag{7.3}$$

where the constant $C \in (0, 1)$ and $p : \mathbb{R} \to (0, 1)$ is a *logistic function*, also called a *sigmoid function* sig,

$$p(z) = \mathrm{sig}(z) = \frac{1}{1 + e^{-z}} \ . \tag{7.4}$$

The function describes the probabilities of outcomes in the model.

Remark 7.1
If an objective space consists of J classification classes with $J > 2$, then the logistic function is substituted by a softmax function $g : \mathbb{R}^J \to \mathbb{R}^J$*:*

$$g(\mathbf{x}; W) = \begin{bmatrix} g(y = 1 \mid \mathbf{x}; W) \\ g(y = 2 \mid \mathbf{x}; W) \\ \dots \\ g(y = J \mid \mathbf{x}; W) \end{bmatrix} = \frac{1}{\sum_{j=1}^{J} e^{\mathbf{w}_j \cdot \mathbf{x} + c_j}} \begin{bmatrix} e^{\mathbf{w}_1 \cdot \mathbf{x} + c_1} \\ e^{\mathbf{w}_2 \cdot \mathbf{x} + c_2} \\ \dots \\ e^{\mathbf{w}_J \cdot \mathbf{x} + c_J} \end{bmatrix}, \tag{7.5}$$

where $W \in \mathbb{R}^{J \times n}$ with columns $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_J \in \mathbb{R}^n$ is a matrix of parameters to be found.

Given a training set $P = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_d, y_d)\}$, to find the weights $\mathbf{w}$, one solves one of the following optimization problems[2]:

1. logistic regression

$$\min_{\mathbf{w},c} \sum_{i=1}^{d} \log(e^{-y_i(\mathbf{x}_i \cdot \mathbf{w} + c)} + 1); \tag{7.6}$$

2. logistic regression with l_1 regularization $\|\mathbf{w}\|_1$:

$$\min_{\mathbf{w},c} \|\mathbf{w}\|_1 + L \sum_{i=1}^{d} \log(e^{-y_i(\mathbf{x}_i \cdot \mathbf{w} + c)} + 1); \tag{7.7}$$

3. logistic regression with l_2 regularization $\frac{1}{2}\mathbf{w} \cdot \mathbf{w}$:

$$\min_{\mathbf{w},c} \frac{1}{2}\mathbf{w} \cdot \mathbf{w} + L \sum_{i=1}^{d} \log(e^{-y_i(\mathbf{x}_i \cdot \mathbf{w} + c)} + 1); \tag{7.8}$$

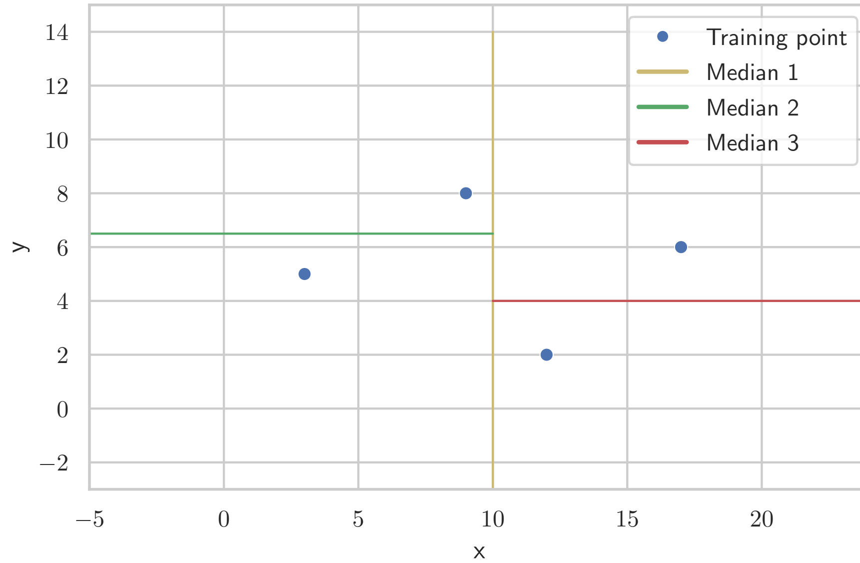
4. logistic regression with elastic-net regularization (combination of l_1 and l_2 regularization terms):

$$\min_{\mathbf{w},c} \frac{1-\rho}{2}\mathbf{w} \cdot \mathbf{w} + \rho \|\mathbf{w}\|_1 + L \sum_{i=1}^{d} \log(e^{-y_i(\mathbf{x}_i \cdot \mathbf{w} + c)} + 1) \ . \tag{7.9}$$

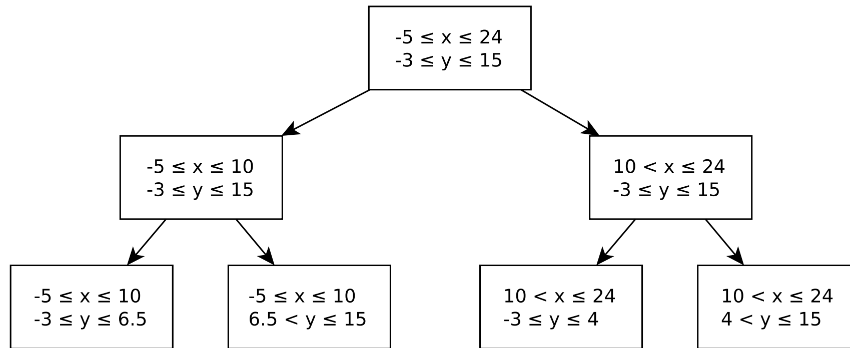
In Equations (7.7) to (7.9), L is a positive constant that defines the inverse of regularization strength: a smaller value means stronger regularization. In Equation (7.9), ρ is a parameter that specifies a ratio between l_1 and l_2 penalties. Regularization penalties allow avoiding overfitting: l_1 regularization limits the size of weights and can yield a sparse model with some weights equal to 0; l_2 regularization shrinks weights without eliminating some of them.

Nearest neighbours classifier. This classifier checks k nearest neighbours of a new sample and assigns the new sample to the most common class among the neighbours [HTF09]. The parameter k is a positive integer number, which typically is small. The following algorithms are presented in the literature to compute distances between points and find the nearest neighbours: Brute Force, K-D Tree, and Ball Tree.

[2] For details of derivation of the problem formulations, we refer to Hastie, Tibshirani, and Friedman [HTF09].



(a) K-D tree decomposition.



(b) K-D tree.

Figure 7.2: Example of K-D tree.

Brute Force computes distances between all pairs of d training points. For a problem with an n-dimensional search space, the computation cost of the algorithm is $\mathcal{O}(nd^2)$, where n is the dimension of the search space of a problem and d is the size of a training set. Brute force is efficient for small data sets and infeasible for large data sets.

K-D tree and ball tree algorithms deal with binary trees which are used to find the nearest neighbours to a new point faster.

Bentley introduced *K-D Trees* [Ben75] in 1975. A K-D tree is built recursively by partitioning a search space along features. E.g. Figure 7.2(a) shows an exemplary 2-dimensional search space with four blue training points and its partitioning, and Figure 7.2(b) presents the resulting K-D tree. The root of the tree is the

whole search space, i.e. the rectangle with $-5 \leqslant x \leqslant 24$ and $-3 \leqslant y \leqslant 15$. The partitioning proceeds as follows:

1. *along the x-axis*: Median 1 (the yellow line in Figure 7.2(a)) divides the root rectangle into two rectangles:
   - $-5 \leqslant x \leqslant 10$ and $-3 \leqslant y \leqslant 15$;
   - $10 < x \leqslant 24$ and $-3 \leqslant y \leqslant 15$.

   These two rectangles form child nodes of the root as shown in Figure 7.2(b).

2. *along the y-axis*: each rectangle obtained in the previous step is divided into two rectangles along the y-axis by Median 2 (green) and Median 3 (red). It forms the following rectangles:
   - $-5 \leqslant x \leqslant 10$ and $-3 \leqslant y \leqslant 6.5$;
   - $-5 \leqslant x \leqslant 10$ and $6.5 < y \leqslant 15$;
   - $10 < x \leqslant 24$ and $-3 \leqslant y \leqslant 4$;
   - $10 < x \leqslant 24$ and $4 < y \leqslant 15$.

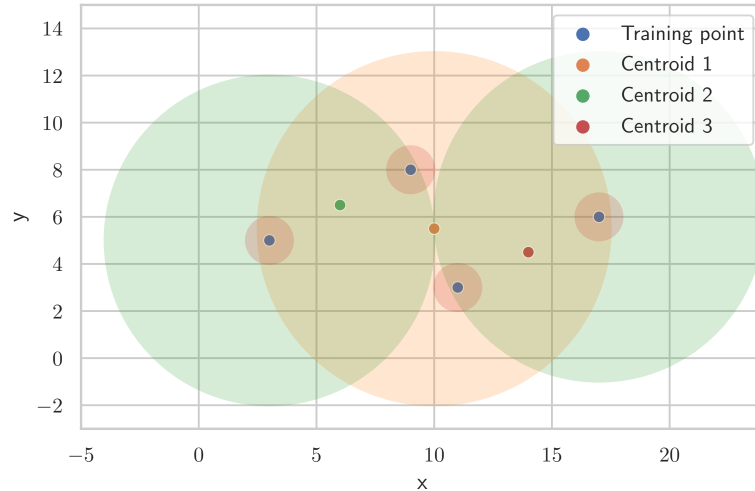
   The newly formed rectangles are leaves in the tree from Figure 7.2(b) since a node in a K-D tree is a leaf if it contains only one training point.

The nearest neighbour can be determined from a K-D tree with $\mathcal{O}(\log(d))$ distance computations, whereas the overall time complexity of the algorithm is $\mathcal{O}(nd\log(d))$ which is faster than brute force for large data sets. The K-D tree approach is fast for low-dimensional search spaces ($n < 20$) and inefficient for large dimensions.

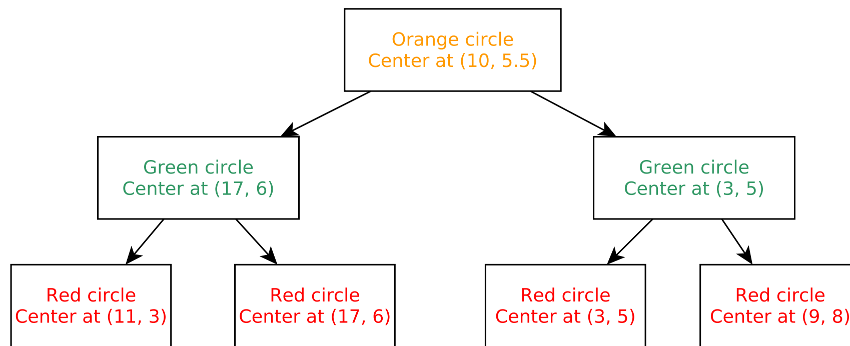
Omohundro presented *Ball Trees* [Omo89] in 1989. A ball tree is built recursively by partitioning a search space into hyperspheres. Figure 7.3(a) presents an exemplary partitioning of a 2-dimensional search space with four blue training points and Figure 7.3(b) shows the corresponding ball tree. We describe the procedure of partitioning by omitting the radii of resulting hyperspheres for the sake of simplicity. How to define the radii is discussed later. The partitioning follows the steps below:

1. the centroid of all training points (Centroid 1) defines the centre of the first hypersphere (the orange ball in Figure 7.3(a)) which is the root of the ball tree shown in Figure 7.3(b);

2. the centre of the first child is a training point with the maximum distance to the centre of the parent hypersphere. In Figure 7.3(a), the training point $(17, 6)$ is the most remote point from the centre of the orange ball, so the green ball with the centre at $(17, 6)$ represents the first child of the root node;



(a) Ball tree decomposition.



(b) Ball tree.

Figure 7.3: Example of ball tree.

3. the centre of the second child is a point with the maximum distance to the centre of the first child within the parent hypersphere. In our example, the training point $(3, 5)$ is the centre of the second child node of the root, namely, the green ball with the centre at $(3, 5)$;

4. each point is assigned to a child hypersphere that has the shortest distance between the point and the child hypersphere's centre. In general, child balls may intersect, but each point is assigned only to one ball. In the example, the points $(11, 3)$ and $(17, 6)$ belong to the first child (the green ball with the centre at $(17, 6)$) and the points $(3, 5)$ and $(9, 8)$ belong to the second child (the green ball with the centre at $(3, 5)$);

5. in the next step, the centroid of each child hypersphere is identified, and Steps 1–3 are repeated for the child nodes. In the example, four red balls

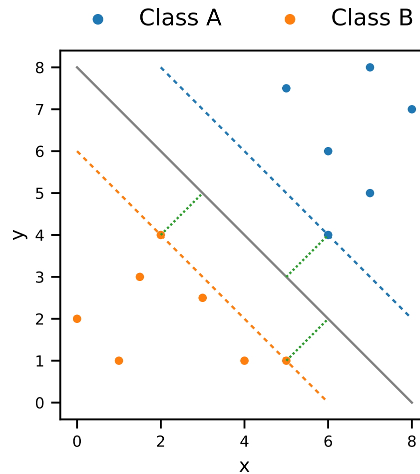


Figure 7.4: Example of support vector classifier. The black line represents a hyperplane separating classes A and B. The points on the dashed lines are called support vectors. The green dotted lines are called functional margins.

from Figure 7.3(a) are formed at this step. The red balls are leaves in the ball tree from Figure 7.3(b) since a node is a leaf in a ball tree if it contains only one training point.

The radius of each ball is the smallest radius such that the ball contains all points assigned to the ball.

The time complexity of the algorithm is $\mathcal{O}(\mathtt{ndlog(d)})$; ball trees can be more efficient than K-D trees in high-dimensional spaces to identify the nearest neighbours, but it highly depends on the structure of a training set.

Support vector classifier. Such classifiers aim to identify hyperplanes separating samples into classes [HTF09]. Functional margin is a distance between a hyperplane and the nearest training points called support vectors. Figure 7.4 presents an example of a hyperplane (the black line), support vectors (the points on the dashed lines), and functional margins (the green dotted lines) for two classes of points: Class A (blue) and Class B (orange). The hyperplane with the largest margin achieves the best separation and leads to the lowest generalization error of the classifier.

For many real-world problems, if a training set is not linearly separable, a *kernel trick* [HTF09] is applied to input data: one maps the input data into a high-dimensional space and finds a hyperplane that separates the samples in the high-dimensional space. E.g. Figure 7.5 shows an example of data that

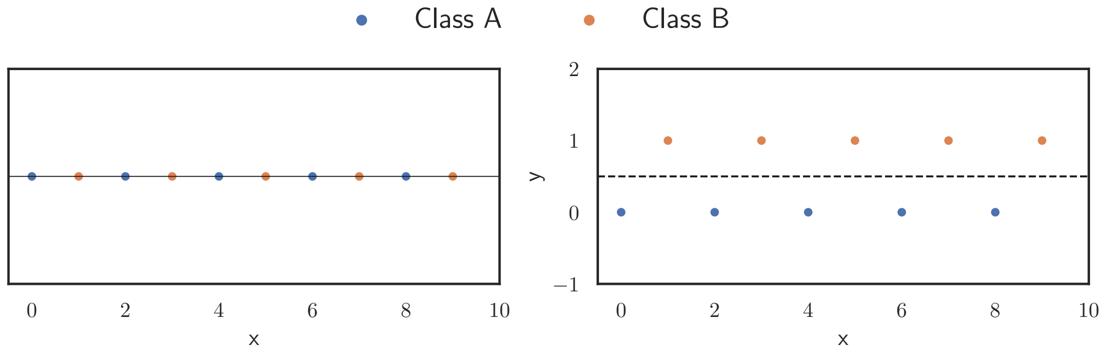


Figure 7.5: Example of kernel trick. The left figure shows data that are not linearly separable in $\mathbb{R}$. The right figure shows the data with the applied transformation $y = x \mod 2$; the resulting data is linearly separable in $\mathbb{R}^2$.

are not linearly separable in $\mathbb{R}$ but linearly separable in $\mathbb{R}^2$ after applying the transformation $y = x \mod 2$.

Given a training set $P = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_d, y_d)\}, \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{1, -1\}$, SVC searches weights $\mathbf{w} \in \mathbb{R}^q$ and a constant c such that $\mathrm{sign}(\mathbf{w} \cdot \phi(\mathbf{x}) + c)$ predicts correctly the objective values of most samples. The function $\phi(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^q$ maps search vectors into a high-dimensional space.

To find $\mathbf{w}$ and c, SVC maximizes the margin by solving the following primal problem:

$$\begin{aligned} &\min_{\mathbf{w},c,\xi} \frac{1}{2}\mathbf{w} \cdot \mathbf{w} + L \sum_{i=1}^{d} \xi_i \\ &\text{subject to } y_i \left(\mathbf{w} \cdot \phi(\mathbf{x}_i) + c\right) \geqslant 1 - \xi_i, \\ &\xi_i \geqslant 0, i = \overline{1, d} \ . \end{aligned} \tag{7.10}$$

Similar to logistic regression, L is the inverse of regularization strength.

If $y_i \left(\mathbf{w} \cdot \phi(\mathbf{x}_i) + c\right)$ would be greater than 1 for all samples, then the problem is perfectly separable and all predictions are correct. Since this is not always the case, the variables ξ indicate distances at which samples are allowed to be from margin boundaries.

To solve the problem with equality constraints defined in Equation (7.10), the following dual problem is formulated[3]:

[3]Lagrange multipliers are used to formulate the dual problem, for more details, we refer to Hastie, Tibshirani, and Friedman [HTF09].

$$\min_{\boldsymbol{\alpha}} \frac{1}{2}\boldsymbol{\alpha}^T Q\boldsymbol{\alpha} - \sum_{i=1}^{d} \alpha_i$$
$$\text{subject to } \sum_{i=1}^{d} y_i \alpha_i = 0, \tag{7.11}$$
$$0 \leqslant \alpha_i \leqslant L, i = \overline{1, d} \ .$$

$Q \in \mathbb{R}^{n \times n}$ is a positive semidefinite matrix with $Q_{ij} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$, where $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$ is a kernel.

In the context of this thesis, we compare two kernels: a *radial basis function* $K_{RBF} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ (it is often used for binary data)

$$K_{RBF}(\mathbf{x}_1, \mathbf{x}_2) = \exp\left(\frac{\|\mathbf{x}_1 - \mathbf{x}_2\|^2}{2\sigma^2}\right) \text{ with } \sigma \in \mathbb{R} \tag{7.12}$$

and a *linear function* $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ (the simplest kernel function)

$$K(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{x}_1 \cdot \mathbf{x}_2 + const \ . \tag{7.13}$$

The constant in the linear kernel is not significant and can be set to 0.

If a search space consists of binary vectors, poor predictions are expected when using a linear kernel.

After solving the optimization problem defined in Equation (7.11), prediction for a given decision vector $\mathbf{x}$ is given by

$$\sum_{i=1}^{d} y_i \alpha_i K(\mathbf{x}_i, \mathbf{x}) + c \ . \tag{7.14}$$

Remark 7.2
In general, SVC supports binary classification. To enable multiclass classification, SVC divides a multiclass problem into binary classification problems and follows one of the following approaches:

1. one-to-one approach*: SVC searches for a hyperplane for each pair of classes;*
2. one-to-rest approach*: SVC searches for a hyperplane that separates one class from the rest classes.*

Gaussian process classifier. This is a probabilistic classification method based on a Bayesian methodology described by Rasmussen [Ras04]. We denote by

$P = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_d, y_d)\}$ a training set, by $\mathbf{y}_{in} = (y_1, y_2, \ldots, y_d)^T$ a vector of the objective values of the training points, and by X_{in} a matrix containing all training vectors $\mathbf{x}_i$. The classifier places a Gaussian process prior on an unobserved latent variable $\mathbf{g} = (g_1, g_2, \ldots, g_d)$:

$$p(\mathbf{g} \mid X_{in}) = \mathcal{N}(\mathbf{g} \mid \mu, K), \tag{7.15}$$

where $\mathcal{N}$ stands for Gaussian distribution and μ and K are the parameters of the Gaussian distribution: the mean vector and covariance matrix. Dealing with a binary-encoded search space, one uses the RBF kernel defined in Equation (7.12).

To compute the posterior distribution of the variable $\mathbf{g}$ given $\mathbf{y}_{in}, X_{in}$, Bayes' theorem is applied:

$$p(\mathbf{g} \mid \mathbf{y}_{in}, X_{in}) = \frac{p(\mathbf{y}_{in} \mid \mathbf{g}) p(\mathbf{g} \mid X_{in})}{p(\mathbf{y}_{in} \mid X_{in})}, \tag{7.16}$$

where $p(\mathbf{g} \mid X_{in})$ is the prior defined in Equation (7.15) and $p(\mathbf{y}_{in} \mid X_{in})$ is a normalizing constant calculated as

$$p(\mathbf{y}_{in} \mid X_{in}) = \int_{\mathbb{R}^d} p(\mathbf{y}_{in} \mid \mathbf{z}) p(\mathbf{z} \mid X_{in}) d\mathbf{z} \ . \tag{7.17}$$

The term $p(\mathbf{y}_{in} \mid \mathbf{g})$ in Equation (7.16) is the likelihood of $\mathbf{y}_{in}$ given $\mathbf{g}$. In the case of binary classification, i.e. $y \in \{0, 1\}$, it is computed by using the sigmoid function sig defined in Equation (7.4):

$$\begin{aligned} p(y = 0 \mid g_i) &= sig(g_i), \\ p(y = 1 \mid g_i) &= 1 - p(y = 0 \mid g_i). \end{aligned} \tag{7.18}$$

Figure 7.6 illustrates an example of the terms from Bayes' theorem (7.16): a prior distribution (the blue curve), a likelihood $p(\mathbf{y}_{in} \mid \mathbf{g})$ (the green curve), a posterior distribution $p(\mathbf{g} \mid \mathbf{y}_{in}, X_{in})$ (the orange curve).

For a test vector $\mathbf{x}^*$, the classifier predicts value at the new point by using the posterior of the output variable

$$y^* \mid \mathbf{y}_{in}, X_{in} = g^* \mid \mathbf{y}_{in}, X_{in} + \epsilon, \tag{7.19}$$

where the distribution for $g^* \mid \mathbf{y}_{in}, X_{in}$ is computed by using Equation (7.16) and ϵ is an added Gaussian noise. The posterior of $\mathbf{g}$ is not Gaussian but can be approximated with Gaussian by using the Laplace approximation as mentioned by Rasmussen [Ras04]. After applying the Laplace approximation, the distribution for $y^* \mid \mathbf{y}_{in}, X_{in}$ is also Gaussian.

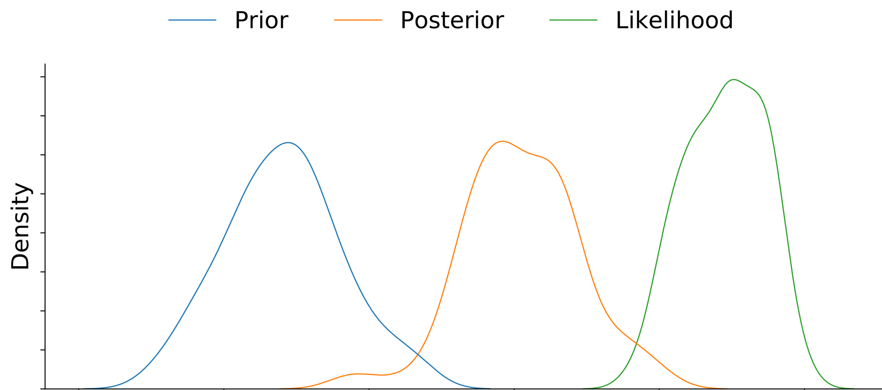


Figure 7.6: Example of a prior distribution, a posterior distribution, and a likelihood function in the Bayes rule.

To extend binary SVC to multi-label classification, two main strategies are presented in the literature: *one-versus-rest* – binary classifier is fitted for each label – and *one-versus-one* – binary classifier is fitted for each pair of labels.

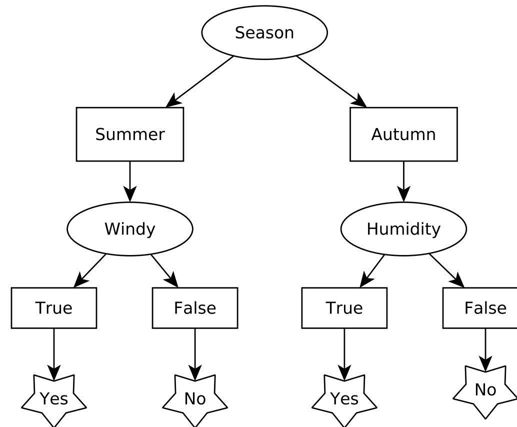
Decision tree classifier. This is a simple classifier that splits data according to a certain criterion [HTF09]. Figure 7.7 shows an example of a decision tree that tries to answer the question of whether it is reasonable to take an umbrella under certain weather conditions. Figure 7.7(a) presents a data set containing three predictors: the season (Autumn or Summer), humidity (High or Normal), and whether it is windy (True or False), and suggestion about an umbrella as a target. Figure 7.7(b) shows the decision tree corresponding to the table; it says that one should take an umbrella if it is windy in summer or humidity is high in autumn.

The most widely used splitting functions in decision tree classifier are the Gini Index function [Gin55] introduced by Gini in 1955 and the Information Gain function [Qui86] presented by Quinlan in 1986. The Gini Index function measures how often a new sample would be incorrectly classified if it was randomly classified according to the distribution of classes from a data set. While building a decision (sub)tree, one chooses a feature with the least Gini index as the root node. The Information Gain function measures the amount of information that a feature renders about a class. While building a decision (sub)tree, a feature with the largest value of the Information Gain function is chosen as the root node.

Random forest classifier. Breiman introduced this classifier in 2001 [Bre01]. It fits several decision trees on randomly selected data samples from a training set. The idea is that a decision done by relatively uncorrelated models should outperform any individual model. Random forest classifier predicts a value at

Predictors			Target
Season	Humidity	Windy	Take an umbrella
Autumn	High	False	Yes
Autumn	High	True	Yes
Summer	High	False	No
Summer	Normal	False	No
Summer	Normal	True	Yes

(a) Data set.



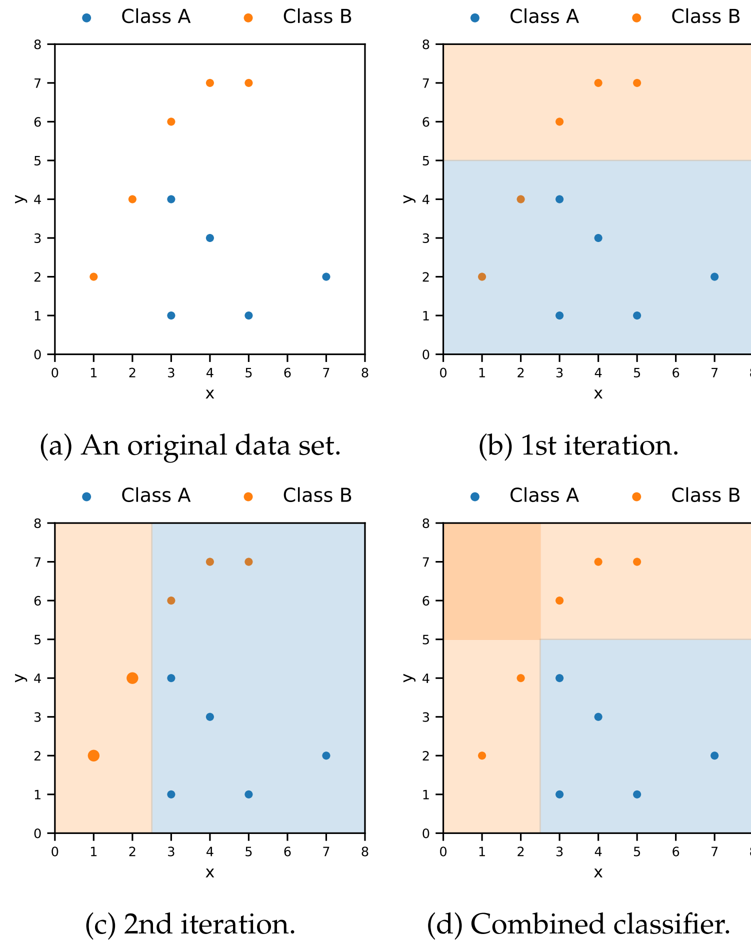
(b) Decision tree.

Figure 7.7: Example of decision tree.

a new sample as follows: each tree predicts a value at the new sample, and the class selected by major voting is assigned to the new sample.

AdaBoost classifier. Freund and Schapire introduced Adaptive Boosting (AdaBoost) classifier [FS97] in 1997. The classifier aims to increase classification accuracy by combining multiple classifiers (so-called base learners), which show poor performance, e.g. small decision trees. The classifier is fitted on various weighted training samples. For the first iteration, all weights are equal to $1/d$, where d is the number of input samples. In each next iteration, the classifier focuses more on difficult instances: it increases the weights of incorrectly classified samples and decreases the weights of correctly classified samples. The learning algorithm is retrained with new weights and proceeds to the next iteration.

Figure 7.8 shows an example of AdaBoost classifier with decision trees. Figure 7.8(a) presents a data set with 10 points from two classes A and B with the original weight of each point equal to 1/10. Figure 7.8(b) illustrates the first iteration of the algorithm: a decision tree classifier is trained and divides all points into two classes: the orange and blue rectangles. Two points from Class



(a) An original data set. (b) 1st iteration.

(c) 2nd iteration. (d) Combined classifier.

Figure 7.8: Example of AdaBoost classifier with decision trees.

B, namely, $(1,2)$ and $(2,4)$ are misclassified, so their weights are increased at the next iteration of the algorithm as shown in Figure 7.8(c), and a new decision tree is trained. Since the weights of the misclassified points were increased, the second decision tree focuses on correct classification of these points. Figure 7.8(d) shows a final classifier that combines both decision trees from the previous iterations and correctly classifies all points.

Naive Bayes classifier. This classifier makes a naive assumption – described by Webb et al. [Web+11] – in Bayes' theorem such that every pair of features are independent given an output variable y.

Recall *Bayes' theorem*:

$$p(y \mid \mathbf{x}) = \frac{p(y)p(\mathbf{x} \mid y)}{p(\mathbf{x})}, \tag{7.20}$$

where

- $p(\mathbf{x})$ is a probability to observe $\mathbf{x}$;
- $p(y)$ is a probability to observe y;
- $p(\mathbf{x} \mid y)$ is a likelihood to observe $\mathbf{x}$ given y;
- $p(y \mid \mathbf{x})$ is a likelihood to observe y given $\mathbf{x}$.

The naive assumption

$$p(x_i \mid y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_d) = p(x_i \mid y) \tag{7.21}$$

simplifies Equation (7.20) to

$$p(y \mid \mathbf{x}) = \frac{p(y) \prod_{i=1}^{d} p(x_i \mid y)}{p(\mathbf{x})} \ . \tag{7.22}$$

The strong assumption (7.21) is often violated in real data, but the classifier shows surprisingly good results even for such data.

Naive Bayes classifiers differ mainly by the probability of input data given output values. E.g. for binary-valued input vectors, it is assumed that input data is distributed according to a *multivariate Bernoulli distribution*, i.e.

$$p(x_i \mid y) = p(x_i = 1 \mid y) x_i + p(x_i = 0 \mid y)(1 - x_i), \tag{7.23}$$

where $x_i \in \{0, 1\}$ and $p(x_i = 0 \mid y) = 1 - p(x_i = 1 \mid y)$.

In the next section, we present an algorithm that chooses the most suitable classifier among the classifiers described above for a concrete problem, i.e. a concrete compiler-based optimization and a concrete benchmark.

7.2.2 Selection of Best Classifiers

This section describes an algorithm to choose the best classifier to predict an objective (WCET or energy consumption) while performing a compiler-based optimization. As described in Section 4.3, any machine learning method depends on a training set, which can be of any size. A smaller training set speeds up the training process of a prediction model, so when choosing the best classifiers, our goal is twofold:

1. choose a classifier with a high prediction quality;
2. choose a classifier with the smallest training set required.

Algorithm 2 Test classifiers.

1: **Input:** classifiers to be tested,
2: objective function $f : X \to \mathbb{R}$ to generate a training set,
3: minimum size M_{min} of the training set,
4: maximum size M_{max} of the training set,
5: step M_{step} to increase the size of the training set,
6: randomly generated test set $T = \{(\mathbf{x}, y) : \mathbf{x} \in X, y = f(\mathbf{x})\}$.
7:
8: **Output:** a set of classifiers' scores.
9:
10: Generate a training set P of size M_{min}.
11: **while** $|P| < M_{max}$ **do**
12: **for all** classifiers **do**
13: Fit the classifier by using the training set P.
14: Compute MAE from Equation (7.24) by using the test set T.
15: $S_{classifier,|P|} = MAE$. ▷ Store the classifier's score.
16: Add M_{step} randomly generated training points to the set P.
17: **return** a set S consisting of all $S_{classifier,|P|}$.

To evaluate the quality of a classifier, we use the Mean Absolute Error (MAE), which is to be minimized.

Definition 7.1
$\{y_i\}_{i=1}^{I}$ *and* $\{y_i'\}_{i=1}^{I}$ *are objective values computed (or estimated) and predicted for a set of search vectors, respectively. The Mean Absolute Error (MAE) is defined as follows:*

$$MAE = \frac{1}{I} \sum_{i=1}^{I} |y_i - y_i'| \ . \tag{7.24}$$

MAE is usually used for regression problems, but since WCET and energy consumption are, in general, discretized continuous objectives, MAE allows us to take into account the case when the objective is misclassified but the predicted objective is still close to the estimated one.

Any training and test point is a pair $(\mathbf{x}, y)$, where $\mathbf{x}$ is a vector of features and y is the corresponding objective value. E.g. for the function inlining problem formulated in Section 6.3, $\mathbf{x}$ is a search vector that specifies which function calls to be inlined and y is the corresponding WCET or energy consumption estimated by the static analysers aiT or EnergyAnalyser, respectively.

Algorithm 2 presents a procedure to test classifiers (presented in Section 7.2.1) when providing them with training sets of different sizes. We use this algorithm

in our evaluation just to identify the best classifier which is to be used to predict the objectives at compile time, but the algorithm itself is not a part of the considered compiler-based optimization. We integrate only the selected classifier into optimization.

Algorithm 2 takes the following inputs:

- classifiers to be tested;
- an objective function to generate training points;
- minimum and maximum sizes of training sets as well as a step size to increase the size of a training set at each iteration of the algorithm;
- a randomly generated test set T to identify the quality of predictions by using MAE defined in Equation (7.24).

The algorithm outputs a set of scores for each classifier and for each size of a training set.

The algorithm starts with generation of a random training set with M_{min} training points and enters a main loop. The loop condition checks whether the size of the training set is smaller than the largest allowed set size M_{max}. Each classifier is trained based on the training set P, and MAE is computed by using the test set T (Lines 12–15). M_{step} random points are added to the training set and the algorithm proceeds to the next iteration (Line 16).

In Section 7.4.2, we compare classifiers listed in Section 7.2.1 for the function inlining problem formulated in Section 6.3: we run Algorithm 2 for each classifier 10 times, compare the prediction qualities of the classifiers, and choose the best classifiers for the problem.

In the next section, we present a procedure to build a final prediction model with the smallest training set for a selected classifier at compile time.

7.2.3 Building Prediction Models

The previous section describes how to compare classifiers in terms of their prediction quality based on a score metric. After selecting a classifier with the best prediction quality[4], the next step is to build a prediction model for a given program at compile time by using the selected classifier. To avoid a drastic increase in compile time, the aim is to build the model with high prediction accuracy by using the smallest possible training set.

Algorithm 3 presents a procedure to identify a final prediction model with the smallest training set that a classifier requires to achieve a prescribed prediction

[4] Section 7.4.2 presents selection of the best classifier for the exemplary function inlining problem.

Algorithm 3 Find a final prediction model.

```
1:  Input: classifier,
2:         objective function f : X → ℝ to generate a training set,
3:         minimum size M_min of the training set,
4:         maximum size M_max of the training set,
5:         step M_step to increase the size of the training set,
6:         randomly generated test set T = {(x, y) : x ∈ X, y = f(x)},
7:         upper bound S_limit of MAE to terminate the algorithm.
8:
9:  Output: prediction model.
10:
11: S_best := MAX_VALUE.    ▷ Set S_best to the largest possible value of MAE.
12: Generate a training set P of size M_min.
13: while |P| < M_max do
14:     Fit the classifier by using the training set P.
15:     if the classifier is fitted then
16:         Compute MAE from Equation (7.24) by using the test set T.
17:
18:         if MAE ⩽ S_limit then
19:             return the fitted prediction model.    ▷ The model is found.
20:         else
21:             if MAE < S_best then
22:                 S_best = MAE.
23:                 Remember the current prediction model as Model_best.
24:     Add M_step randomly generated training points to the set P.
25: return the prediction model Model_best with the smallest achieved MAE.
```

quality. The algorithm is integrated into the considered multiobjective function inlining to enable WCET and energy consumption predictions during optimization.

Algorithm 3 takes the following inputs:

- a classifier to be fitted;
- an objective function to generate training points;
- minimum and maximum sizes of a training set together with a step size to enlarge the training set;
- a randomly generated test set to compute the accuracy of a model by using MAE defined in Equation (7.24);

- an upper bound S_{limit} of MAE that indicates an acceptable prediction quality and means that the algorithm can be terminated.

The algorithm outputs a prediction model with the smallest possible training set and the highest accuracy.

The algorithm randomly generates a training set of size M_{min} and enters a main loop. The loop condition checks whether the size of the training set is within the acceptable range (Line 13). Inside the loop, the classifier is trained based on the training set P and tested on the test set T by computing the score MAE (Lines 14 and 16). If the score is less than the prescribed upper bound S_{limit}, a prediction model with the smallest possible training set and an acceptable score is found, and the algorithm is terminated (Lines 18–19). If the score is larger than the upper bound S_{limit} and if the current model has the highest accuracy achieved so far, the current model is stored (Lines 20–23). Otherwise, M_{step} random points are added to the training set, and the algorithm proceeds to the next iteration (Line 24). If the training set is larger than its maximum size M_{max} and the classifier has been not able to achieve an acceptable score that is larger than or equal to S_{limit}, the algorithm returns the prediction model with the highest score found so far (Line 25).

The next section describes integration of classifiers into the WCET-Aware C Compiler (WCC) presented in Chapter 3 and used as a base for our evaluations.

7.2.4 WCC and Prediction Models

To build and use a prediction model within WCC, we extended it with a C++ library which we call LIBFUNAPPROX. The library is a bridge between WCC and the scientific toolbox `scikit-learn` developed by Pedregosa et al. [Ped+12]. `scikit-learn` is a widely-used data analysis tool written in Python, it contains implementation of all classifiers considered in this thesis.

Figure 7.9 presents connection of new components to WCC with dashed lines: LIBFUNAPPROX is a new part of WCC, whereas the scikit wrapper is a standalone program written in Python that utilizes `scikit-learn`. LIBFUNAPPROX can be used within optimizations dealing with the high- and low-level intermediate representations of WCC because it operates on a set of samples which represent the search and objective space of an optimization problem and are unaware of any high- or low-level features of a code to be compiled.

LIBFUNAPPROX contains a class called ScikitLib which stores a predictor (or classifier, in our case), which is used to build the prediction model, and the hyperparameters of the predictor if needed. The class supports two methods: train and predict. The "train" method takes a training set as an input parameter and returns true if the predictor is successfully fitted, i.e. `scikit-learn` finishes without errors, and false, otherwise. The "predict" method takes a set of search vectors and returns predicted objective values.

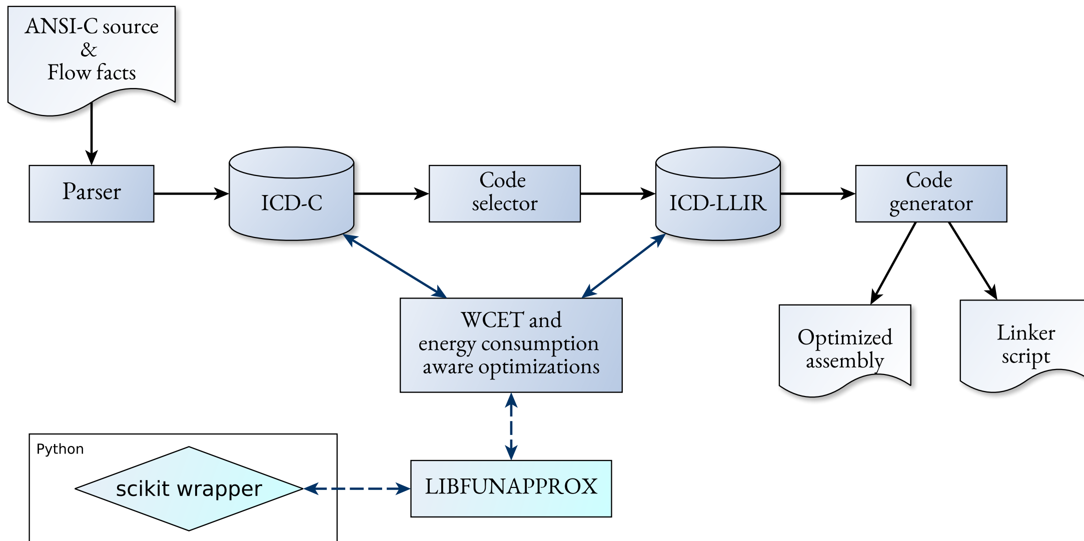


Figure 7.9: New components of WCET-Aware C Compiler (WCC). The solid lines indicate the main flow of WCC, whereas the dashed lines indicate integration of predictors within WCC.

The LIBFUNAPPROX library enables communication between WCC and `scikit-learn`. To fit a prediction model during performing a compiler-based optimization,

1. WCC creates an object of the ScikitLib class, sets a predictor and its hyper-parameters if needed;

2. WCC generates a training set consisting of search vectors of the optimization problem and the corresponding objective values – estimated by aiT and EnergyAnalyser – and passes it to the "train" method of the ScikitLib's object;

3. within the "train" method, the training set is written into a CSV file;

4. WCC calls the scikit wrapper which reads the training set from the CSV file, fits the model by using `scikit-learn` and the training set, and stores the model to a Joblib file. (Joblib files are used to store Python objects.)

At this stage, the prediction model is stored in the Joblib file and can be restored at any time when the model is needed.

To predict objective values during the optimization,

1. WCC passes a set of search vectors to the "predict" method of the ScikitLib's object;

2. within the "predict" method, the search vectors are written to a new CSV file;
3. WCC calls the scikit wrapper that reads the points from the CSV file and restores the prediction model from the Joblib file, predicts objective values, and writes the predicted values to an output CSV file;
4. the "predict" method reads the predicted values from the CSV file and returns them to the optimization.

After extending WCC with the mechanism to build and use prediction models, we utilize them during execution of an evolutionary algorithm at compile time as described in the next section.

7.3 Evolutionary Algorithm and Prediction Model

In Chapter 6, we try to solve a multiobjective problem with WCET, energy consumption, and code size as objectives at compile time by using evolutionary algorithms. Since any evolutionary algorithm extensively evaluates objectives during search space exploration, the solution process becomes very time-consuming due to time-consuming WCET and energy consumption analyses. The previous section describes a prediction model based on classifiers that should substitute time-consuming estimations of the objectives with faster predictions during execution of an evolutionary algorithm at compile time.

Integration of predictions into an evolutionary algorithm execution at compile time is the following:

1. choose a prediction method. In general, the prediction method for every compiler-based optimization should be chosen individually, but, as motivated in Section 7.2, for many compiler-based optimizations, the prediction method, most probably, should be chosen among classification approaches. A general approach to identify a classifier with the best prediction accuracy is described in Section 7.2.2 and is applied to the exemplary function inlining problem in Section 7.4.2;
2. for each time-consuming objective, build a prediction model by using the chosen prediction method and Algorithm 3. As motivated on Page 99, we build independent prediction models for each objective to minimize the size of a training set;
3. during execution of the evolutionary algorithm, when the algorithm requires the objective values (WCET and energy consumption, in our case) at a new point of the search space, predict the objectives by using the prediction models;

4. when the evolutionary algorithm is finished, compute exactly the objectives (in our case, estimate WCET and energy consumption by the static analysers aiT and EnergyAnalyser) of final solutions returned by the algorithm.

Step 4 guarantees that the WCET of the final solutions are safely estimated which is crucial for hard real-time systems.

Section 7.4 presents evaluation results of an evolutionary algorithm that relies on predictions for the multiobjective function inlining problem. We evaluate algorithm's runtime as well as solutions' quality.

7.4 Evaluation

In this section, we present evaluations for the exemplary compiler-based optimization function inlining described in Section 6.3. Section 7.4.1 describes experimental setup, Section 7.4.2 compares classifiers listed in Section 7.2.1 for function inlining and chooses the best classifiers as described in Section 7.2.2, Section 7.4.3 presents the results of solving the multiobjective function inlining problem by an evolutionary algorithm that utilizes WCET and energy consumption predictions which substitute time-consuming WCET and energy consumption analyses.

7.4.1 Experimental Setup

We demonstrate the proposed prediction technique in the context of function inlining formulated in Section 6.3: the input space of a prediction model coincides with the search space of function inlining and the objective space of the model represents WCET or energy consumption. To speed up the compilation process, we build two independent prediction models for WCET and energy consumption with the smallest possible training sets since multiobjective prediction models require many training points as explained by Xu et al. [Xu+19].

We use WCC for an ARM Cortex-M0 processor architecture with optimization level O2, similar to Chapter 6. AbsInt's static analysers aiT and EnergyAnalyser version 20.10i estimate WCET and energy consumption, whereas WCC computes code size. Similar to Chapter 6, we use the benchmark suites MediaBench, MRTC, PolyBench, JetBench, and StreamIt; Appendix B presents a list of used benchmarks.

We use `scikit-learn` version 1.0 [Ped+12]. To tune the parameters of a classifier, we utilize `scikit-learn`'s grid search. For a given training set, it searches the best parameters over specified parameter values by using a k-fold cross-validation score. To compute the cross-validation score, one splits a training set into k smaller sets and repeats the following steps for each newly created set S:

1. train the model by using the remaining $k-1$ sets as training data;

2. measure model accuracy by using the set S as a test set.

The k-fold cross-validation score is the average of the accuracy values computed for k sets. According to James, Witten, and Hastie [Jam+21], $k = 5$ and $k = 10$ are used in practice to compute the k-fold cross-validation score. We preserve `scikit-learn`'s default value $k = 5$.

We use the following values (which are often considered in practice) to tune the parameters of the considered classifiers:

- logistic regression:
  - penalty: none, l_1, l_2, elastic-net (see Equations (7.6) to (7.9));
  - the inverse of regularization strength L: $0.001, 0.01, 0.1, 1, 10$;
  - ratio ρ in elastic-net penalty: $0.2, 0.4, 0.6, 0.8$.

  To solve the problems formulated in Equations (7.6) to (7.9), we use the "saga" solver introduced by Defazio, Bach, and Lacoste-Julien [DBLJ14] since it is the only `scikit-learn`'s solver that supports all three penalties: l_1, l_2, and elastic-net penalties. We increase the maximum number of iterations to 1,000 to guarantee the convergence of the solver.

- nearest neighbours classifier:
  - the number of neighbours to use: $1, 3, 5, \ldots, 21$;
  - the weights of neighbours used in prediction:
    * uniform – all neighbours are weighted equally;
    * distance – neighbours are weighted by the inverse of their distance, i.e. closer neighbours have greater weights;
  - algorithm to find the nearest neighbours: Brute Force, K-D Tree, and Ball Tree.

- support vector classifier with linear and RBF kernels:
  - the inverse of regularization strength L: $0.001, 0.01, 0.1, 1, 10$;
  - $2\sigma^2$ in the RBF kernel defined in Equation (7.12): 0.001, 0.01, 0.1, $1, 10, 100$.

  `scikit-learn` handles multiclass support by following one-to-one approach.

- Gaussian process classifier: no parameter tuning is required since `scikit-learn` assumes the mean vector μ is equal to $\mathbf{0}$ and optimizes automatically the hyperparameter σ of the RBF kernel during training of a prediction model. For multiclass classification, `scikit-learn` follows one-to-rest approach.

- decision tree classifier:
  - maximum tree depth: $2, 5, 10, 20$;
  - the minimum number of samples required to be at a leaf node: $1, 5, 10, 20, 50, 100$;
  - function to measure the quality of a split: the Gini Index function and the Information Gain function.
- random forest classifier:
  - maximum tree depth: $2, 5, 10, 20$;
  - the minimum number of samples required to be at a leaf node: $1, 5, 10, 20, 50, 100$;
  - function to measure the quality of a split: the Gini Index function and the Information Gain function;
  - the number of trees in a forest: $10, 20, 50$.
- AdaBoost classifier:
  - the maximum depth of decision trees used as base learners: $1, 3, 5$;
  - the maximum number of base learners: $10, 50, 100, 500$;
  - weights applied to each classifier at each boosting iteration: $0.0001, 0.001, 0.01, 0.1, 1.0$. This parameter controls contribution of base learners in final predictions.
- naive Bayes: since in the function inlining problem, we consider binary-valued input vectors, we assume that input data is distributed according to a *multivariate Bernoulli distribution*. No parameters to be tuned are present in the classifier.

7.4.2 Selection of Best Classifiers for Function Inlining

This section demonstrates selection of the best classifiers for the function inlining problem based on prediction accuracy and presents results regarding the size of a training set required to build a final prediction model by using the selected classifier.

To choose the most suitable classifiers for WCET and energy consumption when performing function inlining, for each benchmark, we follow the approach described in Section 7.2.2. We run 10 times Algorithm 2 since training sets are randomly generated. We pass the following input parameters to the algorithm:

- *classifiers*: the classifiers listed in Section 7.2.1;

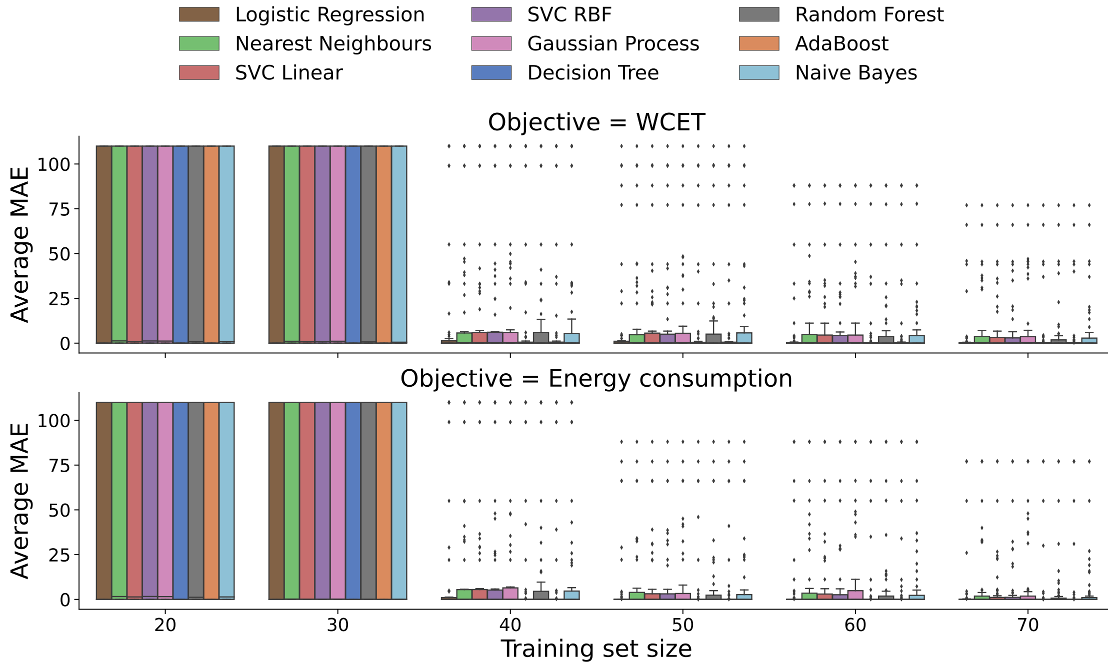


Figure 7.10: Average MAE of classifiers for all considered benchmarks. The best MAE value is 0. Each experiment was repeated 10 times.

- *objective function* f: WCET or energy consumption;
- *minimum size of a training set*: $M_{min} = 20$;
- *maximum size of the training set*: $M_{max} = 70$;
- *step to enlarge the training set*: $M_{step} = 10$;
- *test set* T: a randomly generated set consisting of 10 samples.

We omit training sets of size 10 since 10 training points is not enough to perform k-fold cross-validation with $k = 5$. We limit the maximum size of a training set to 70 and consider a test set of size 10 to minimize the number of expensive evaluations of the objectives. We use the Mean Absolute Error (MAE) defined in Equation (7.24) to measure the accuracy of the classifiers. A smaller value of MAE means a higher accuracy of the classifier. In addition, for each benchmark, we consider relative WCET and relative energy consumption with 100 % corresponding to the objective value of an original program. The relative objectives ensure that values used in MAE computation are of the same magnitude for all benchmarks. If a classifier fails to fit a prediction model for a benchmark, we set MAE to 110 since the worst observed MAE among all successful benchmarks is 107.

Figure 7.10 presents the average MAE values produced in 10 runs of Algorithm 2 for each size of training sets, classifier, and benchmark. The x-axis represents the size of a training set from 20 to 70, the y-axis shows the average MAE with the best value of 0 indicating that all predictions on a given test set are correct. Each colour on the plots corresponds to a specific classifier, e.g. brown corresponds to logistic regression. For each fixed size of a training set and classifier, the box plot shows distribution of the average MAE over all benchmarks for two predicted objectives: WCET (the top plot) and energy consumption (the bottom plot).

For both objectives, the box plot shows that for the training set size equal to 20 and 30, the 3rd quartile is equal to the maximum possible value of 110, which means that all classifiers failed to fit prediction models for around 16 benchmarks (25 % of 62 considered benchmarks) in all 10 runs. For these sizes of a training set, the median of the average MAE is close to 0 for all classifiers, which means that for at least 31 benchmarks (50 % of 62 benchmarks), the classifiers produced a prediction model with a high accuracy in all 10 runs.

Starting from the training set size equal to 40, all classifiers significantly improved their accuracy and the average MAE fell below 25. The box plot shows only some outliers, which means that the classifiers failed only for some benchmarks. An increasing training set size decreases the average MAE of outliers.

For seven benchmarks, all classifiers failed to fit prediction models. The failed benchmarks are `adpcm (MRTC)`, `codecs_codrle1 (misc)`, `codecs_dcodhuff (misc)`, `g721_encode (misc)`, `g723_encode (misc)`, `md5 (NetBench)`, `pm (misc)`. Appendix D shows that the dimension of the search space of these benchmarks, except `pm`, is less than 25 and that for all seven benchmarks, function inlining results in many different values of the objectives, so classification models consist of many classes and require many training points to be fitted. For such benchmarks, a regression model should be probably considered but since it is another large research topic, it is left for future work. From now on, we present results only for 55 successful benchmarks.

To identify the best classifiers for our problem, Figure 7.11 shows the results from Figure 7.10 with the range of the y-axis changed from $[0, 110]$ to $[0, 15]$. For both objectives, logistic regression, decision tree classifier, and AdaBoost classifier show the smallest average MAE, which is below 5 for most benchmarks. Since we use relative objectives to compute MAE, the value 5 can be interpreted as follows: predicted values differ from the corresponding estimated values by 5 %.

The classifiers based on decision trees show high-quality results for our problem, since, as shown in Appendix D, most benchmarks have a large search space but only a limited number of unique WCET and energy consumption values are observed, so many dimensions of the search space do not influence the objectives and decision trees can identify redundant features. It motivated us to develop a search space reduction technique presented in Chapter 8.

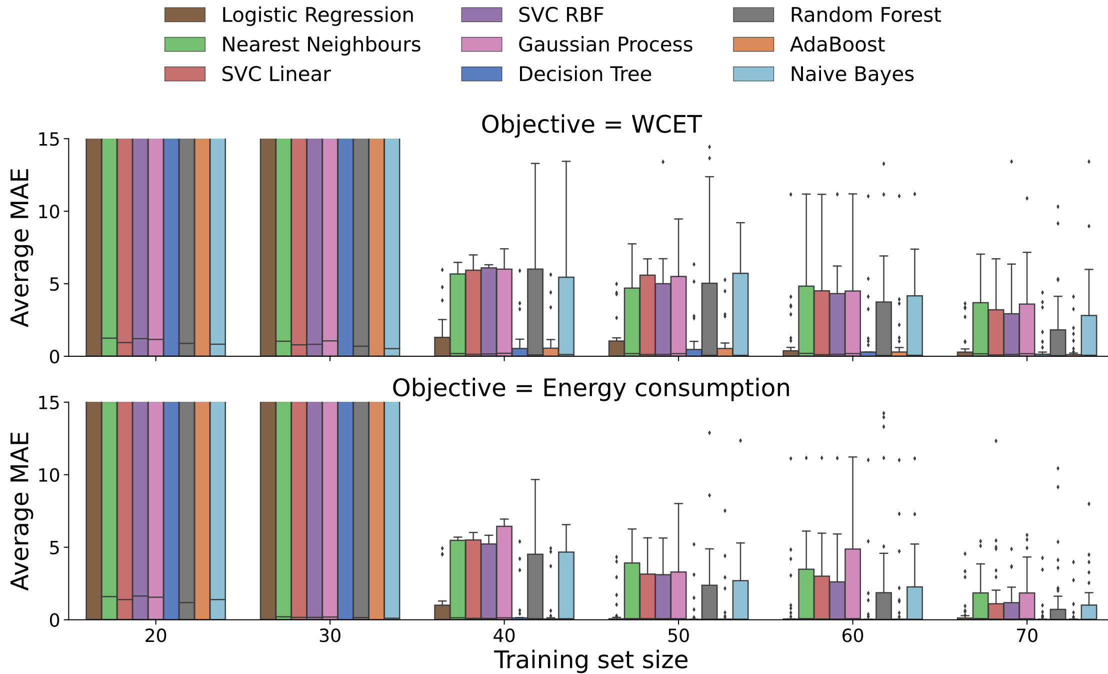


Figure 7.11: Average MAE from Figure 7.10 with the range of the y-axis changed to $[0, 15]$. The best MAE value is 0. Each experiment was repeated 10 times.

In the next section, we evaluate the best found classifiers in terms of guiding an evolutionary algorithm to promising regions of the search space.

7.4.3 Evolutionary Algorithm Based on Predicted Objectives

In Chapter 6, we compared two evolutionary algorithms, BGDE3 and MBPOA. In general, the algorithms showed similar results, but MBPOA resulted in Pareto fronts of a little bit better quality than BGDE3 for many benchmarks. For this reason, we integrate the best found classifiers (logistic regression, decision tree classifier, and AdaBoost classifier) into MBPOA as described in Section 7.3. We set the MBPOA crossover parameter to 0.8 as Section 6.4.2 suggests. We repeat each experiment 10 times due to the randomness of evolutionary algorithms and randomly generated training sets for classifiers.

Before predicting objectives during evolutionary algorithm execution, the compiler fits a prediction model by using Algorithm 3. We consider the following input parameters for Algorithm 3:

- *classifier*: logistic regression, decision tree classifier, and AdaBoost classifier;
- *objective function* f: WCET or energy consumption;

Table 7.1: Maximum training set sizes and the corresponding number of benchmarks for which the sizes were used by Algorithm 3.

Classifier	Objective	Training set size					
		20	30	40	50	60	70
AdaBoost	Energy cons.	34	4	11	2	2	2
	WCET	31	5	13	0	3	3
Decision tree	Energy cons.	32	7	11	0	2	3
	WCET	32	3	11	1	5	3
Log. regression	Energy cons.	32	5	13	1	1	3
	WCET	31	3	13	2	2	4

- *minimum size of a training set*: $M_{min} = 20$;
- *maximum size of the training set*: $M_{max} = 70$;
- *step* $M_{step} = 10$;
- *test set* T: a randomly generated set consisting of 10 samples;
- *upper bound of MAE*: $S_{limit} = 5$.

We set the upper bound of MAE to 5, since, in the previous section, we showed that the average MAE for the selected classifiers is below 5 for most benchmarks (see Figure 7.11).

The goal of Algorithm 3 is to build a model with as few training points as possible and achieve a low MAE on a given test set. For both objectives, all three classifiers achieved the average MAE less than 5 for all considered benchmarks except `codecs_codhuff` and `codecs_dcodrle1`. Logistic regression also resulted in the average MAE greater than 5 for the benchmark `cjpeg_wrbmp`.

For each classifier and objective, Table 7.1 presents maximum training set sizes and the corresponding number of benchmarks for which the sizes were used by Algorithm 3 to build a final prediction model. The third column of the table shows that for half of the benchmarks, the algorithm required at most 20 training points to fit the model by using any of the three classifiers. The fourth and fifth columns show that for some benchmarks, the algorithm required 30 or 40 training points. The sixth, seventh, and eighth columns show that only for a few benchmarks, the algorithm required more than 40 training points.

Appendix E presents the maximum sizes of training sets used to build prediction models in Algorithm 3 and the average score for all considered benchmarks.

Next, we demonstrate how the evolutionary algorithm MBPOA speeds up due to utilization of the prediction models. We also compare Pareto fronts returned

by MBPOA, which explores the search space of a problem by using predictions produced by logistic regression, decision tree classifier, or AdaBoost classifier.

When executing MBPOA, WCC computes code size exactly, whereas WCET and energy consumption we get either from aiT and EnergyAnalyser, or the prediction models. So in this section, we compare Pareto fronts produced by four methods:

1. $\mathsf{MBPOA}_{\mathsf{LogReg}}$ – MBPOA with WCET and energy consumption predicted by logistic regression;
2. $\mathsf{MBPOA}_{\mathsf{DTree}}$ – MBPOA with WCET and energy consumption predicted by decision tree classifier;
3. $\mathsf{MBPOA}_{\mathsf{ABoost}}$ – MBPOA with WCET and energy consumption predicted by AdaBoost classifier based on decision trees;
4. $\mathsf{MBPOA}_{\mathsf{AbsInt}}$ – MBPOA with WCET and energy consumption estimated by aiT and EnergyAnalyser.

$\mathsf{MBPOA}_{\mathsf{LogReg}}$, $\mathsf{MBPOA}_{\mathsf{DTree}}$, and $\mathsf{MBPOA}_{\mathsf{ABoost}}$ return Pareto fronts with predicted values of the objectives which might differ from safely estimated objectives, so aiT and EnergyAnalyser estimate the objectives of Pareto points returned by $\mathsf{MBPOA}_{\mathsf{LogReg}}$, $\mathsf{MBPOA}_{\mathsf{DTree}}$, and $\mathsf{MBPOA}_{\mathsf{ABoost}}$ and we report the results for the estimated objectives.

To fairly evaluate the runtime of MBPOA and the quality of approximated Pareto fronts returned by the methods, we set the number of generations to 30 and the population size to 50 for all MBPOA runs and for all benchmarks, similar to Chapter 6. We randomly generate the initial population of the algorithm and repeated the experiments with $\mathsf{MBPOA}_{\mathsf{LogReg}}$, $\mathsf{MBPOA}_{\mathsf{DTree}}$, $\mathsf{MBPOA}_{\mathsf{ABoost}}$, and $\mathsf{MBPOA}_{\mathsf{orig}}$ 10 times for each benchmark.

Figure 7.12 presents in hours the average runtime of the considered methods: $\mathsf{MBPOA}_{\mathsf{LogReg}}$ (brown), $\mathsf{MBPOA}_{\mathsf{DTree}}$ (blue), $\mathsf{MBPOA}_{\mathsf{ABoost}}$ (orange), $\mathsf{MBPOA}_{\mathsf{AbsInt}}$ (green). The x-axis shows the benchmarks; the y-axis presents average runtime. The runtime of $\mathsf{MBPOA}_{\mathsf{AbsInt}}$ summarizes

1. MBPOA runtime;
2. aiT and EnergyAnalyser runtimes required to estimate the objectives during MBPOA execution.

$\mathsf{MBPOA}_{\mathsf{LogReg}}$, $\mathsf{MBPOA}_{\mathsf{DTree}}$, and $\mathsf{MBPOA}_{\mathsf{ABoost}}$ runtimes summarize

1. MBPOA runtime;
2. runtime to find prediction models for WCET and energy consumption by using Algorithm 3;

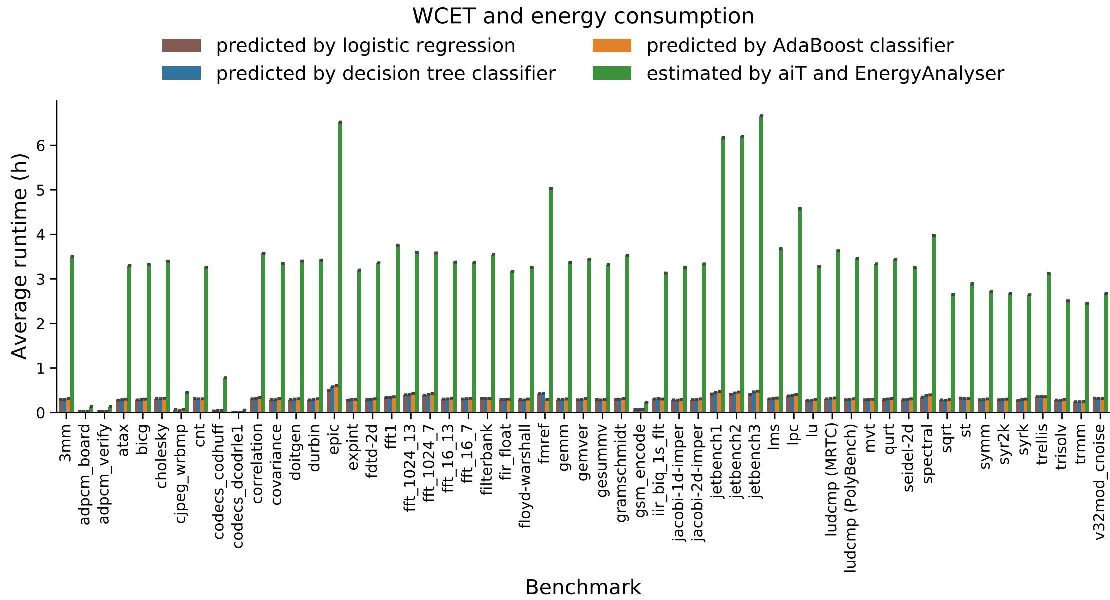


Figure 7.12: Runtimes of the evolutionary algorithm MBPOA. During MBPOA execution, WCET and energy consumption are either estimated by aiT and EnergyAnalyser or predicted by logistic regression, decision tree, or AdaBoost classifiers.

3. runtime needed to predict the objectives during MBPOA execution;
4. aiT and EnergyAnalyser runtimes to estimate the objectives of final MBPOA solutions.

The figure shows a large speed-up if we use predictions. The methods based predictions show almost the same saving in runtime. They are, on average, 3 h faster than the original MBPOA over all tested benchmarks. We observe larger savings for more time-consuming benchmarks, e.g. we notice

- the largest saving for the benchmark `jetbench3`, which is one of the most time-consuming benchmarks:
  - the average $\text{MBPOA}_{\text{AbsInt}}$ runtime is 6.7 h;
  - the average $\text{MBPOA}_{\text{LogReg}}$ runtime is 24 min which is 6.3 h less than the average $\text{MBPOA}_{\text{AbsInt}}$ runtime;
  - the average $\text{MBPOA}_{\text{DTree}}$ and $\text{MBPOA}_{\text{ABoost}}$ runtime is 28 min which is 6.2 h less than the average $\text{MBPOA}_{\text{AbsInt}}$ runtime;
- the smallest saving for the benchmark `codecs_dcodrle1`, which is the least time-consuming benchmark:

– the average $\mathsf{MBPOA}_{\mathtt{AbsInt}}$ runtime is 3.3 min;
– the average $\mathsf{MBPOA}_{\mathtt{LogReg}}$ runtime is 15 s;
– the average $\mathsf{MBPOA}_{\mathtt{DTree}}$ runtime is 17 s;
– the average $\mathsf{MBPOA}_{\mathtt{ABoost}}$ runtime is 18 s.

When utilizing predictions during MBPOA execution, we still observe a large difference between its runtime for different benchmarks, e.g. the average runtime is 26 min for `jetbench3` and 17 s for `codecs_dcodrle1`. This difference is explained by the necessity to generate training sets and to estimate the objectives of final solutions by aiT and EnergyAnalyser: the aiT and EnergyAnalyser runtimes as well as the number of the final solutions differ among the benchmarks.

Figure 7.13 presents stack diagrams for three classifiers: logistic regression (the top plot), decision tree (the middle plot) and AdaBoost (the bottom plot). The stacks consist of two parts:

1. runtime to fit two prediction models for WCET and energy consumption (blue);
2. MBPOA runtime (orange) which uses predictions; it includes aiT and EnergyAnalyser runtimes to estimate the objectives of final solutions.

For most of the benchmarks, the runtime to fit the classifiers is less than 10 % of the overall runtime, e.g.

- for the benchmark `jetbench3`, all classifiers required 20 training points to fit models in all 10 runs (see Table E.1), the average runtime of the classifiers to fit the models is around 2 min, and the average MBPOA runtime is around 24 min;
- for the benchmark `codecs_dcodrle1`, the classifiers required at most 70 training points to fit models in all 10 runs (see Table E.1), the average runtime of the classifiers to fit the models is around 11 s, and the average MBPOA runtime is around 6 s.

All classifiers show almost the same saving in the overall runtime, so next, we compare the quality of Pareto fronts found by $\mathsf{MBPOA}_{\mathtt{LogReg}}$, $\mathsf{MBPOA}_{\mathtt{DTree}}$, $\mathsf{MBPOA}_{\mathtt{ABoost}}$, and $\mathsf{MBPOA}_{\mathtt{AbsInt}}$.

Figure 7.14 shows the values of two considered quality indicators[5]: nondominance ratio NR defined in Equation (4.8) (the top plot) and coverage C defined in Equation (4.9) (the bottom plot) over 10 runs for the considered methods: $\mathsf{MBPOA}_{\mathtt{LogReg}}$ (brown), $\mathsf{MBPOA}_{\mathtt{DTree}}$ (blue), $\mathsf{MBPOA}_{\mathtt{ABoost}}$ (orange), and $\mathsf{MBPOA}_{\mathtt{AbsInt}}$ (green). The bars represent the average values of the quality

[5]The quality indicators were computed as described on Page 84.

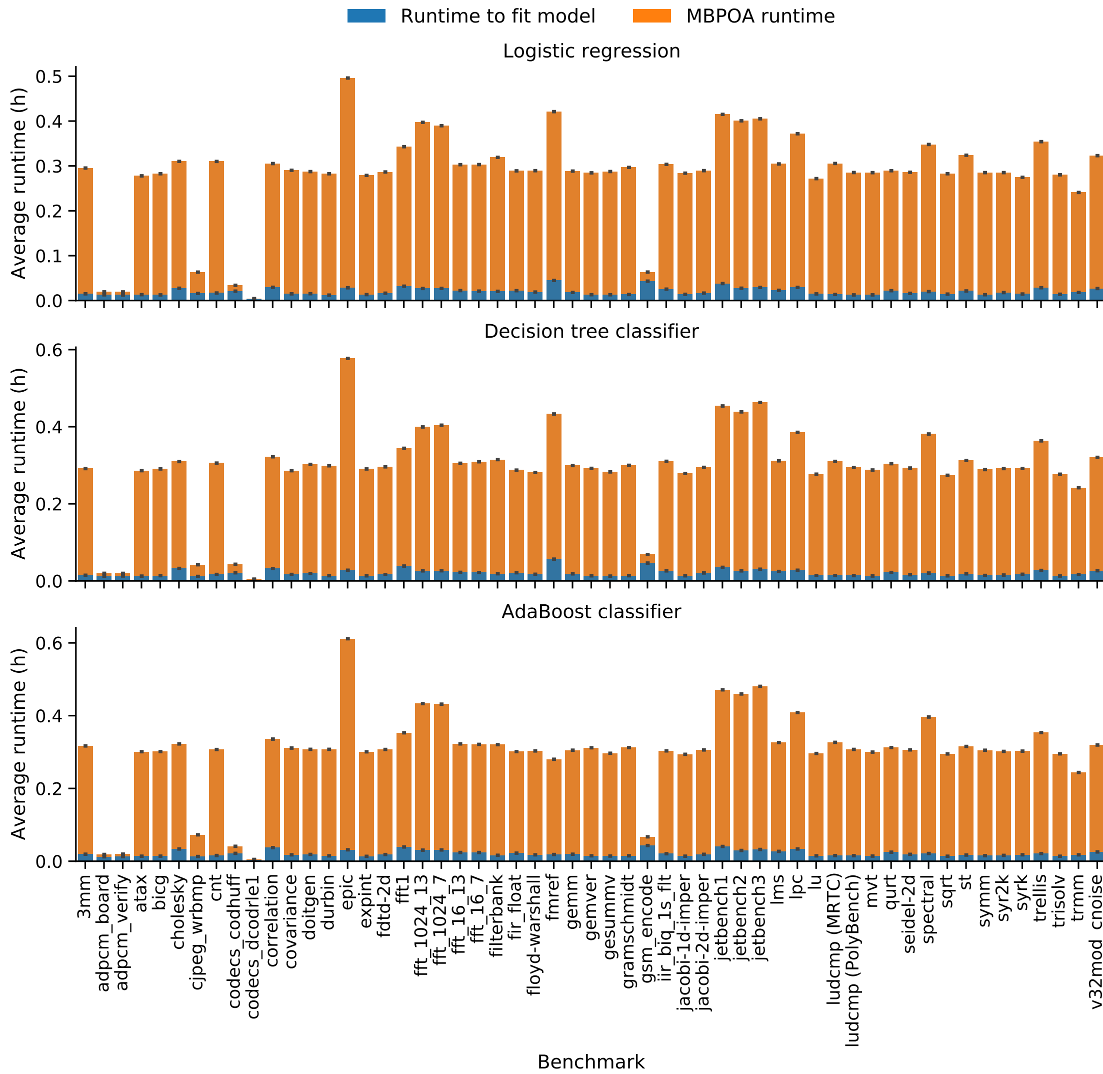


Figure 7.13: Runtimes of $\mathrm{MBPOA}_{\mathrm{LogReg}}$, $\mathrm{MBPOA}_{\mathrm{DTree}}$, and $\mathrm{MBPOA}_{\mathrm{ABoost}}$ consisting of runtime to fit logistic regression, decision tree, and AdaBoost classifiers for WCET and energy consumption and MBPOA runtime which utilizes the prediction models.

indicators and the error bars show the 95 % confidence interval. The best value of nondominance ratio is 1 and the best value of coverage is 0. The x-axis lists the benchmarks and the dimensions of their search spaces in parentheses.

Since randomness is present in evolutionary algorithms and training and test sets for predictors, we compare the average values of the quality indicators by using comparison tolerance equal to 0.05, i.e. two values are equal if their

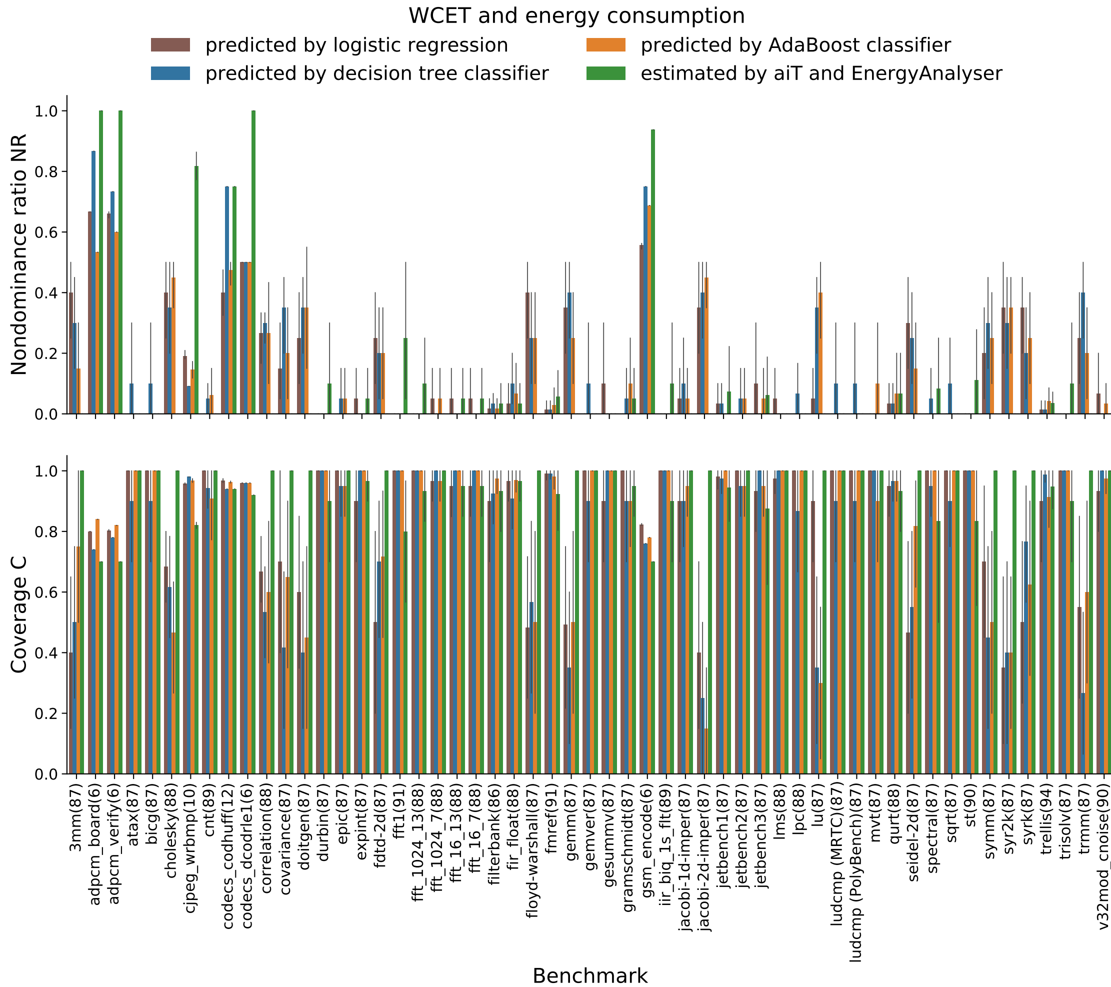


Figure 7.14: Quality indicators of Pareto fronts returned by the evolutionary algorithm MBPOA with estimated and predicted objectives. Nondominance ratio is to be maximized and coverage is to be minimized. The dimensions of the search spaces are specified in parentheses next to the benchmarks' names. Each experiment was repeated 10 times.

absolute difference is less than 0.05. The figure presents the following results for 55 successful benchmarks:

- for 9 benchmarks like, e.g. `epic`, `fft_1024_7`, or `fft_16_13`, all four methods show the same results;

- for 12 benchmarks like, e.g. `adpcm_board`, `adpcm_verify`, or `cjpeg_wrbmp`, $\text{MBPOA}_{\text{AbsInt}}$ outperformed the three other methods based on predictions;

- for 34 benchmarks like, e.g. `3mm`, `atax`, or `cholesky`, at least one of the methods based on predictions outperformed $\mathrm{MBPOA}_{\mathrm{AbsInt}}$.

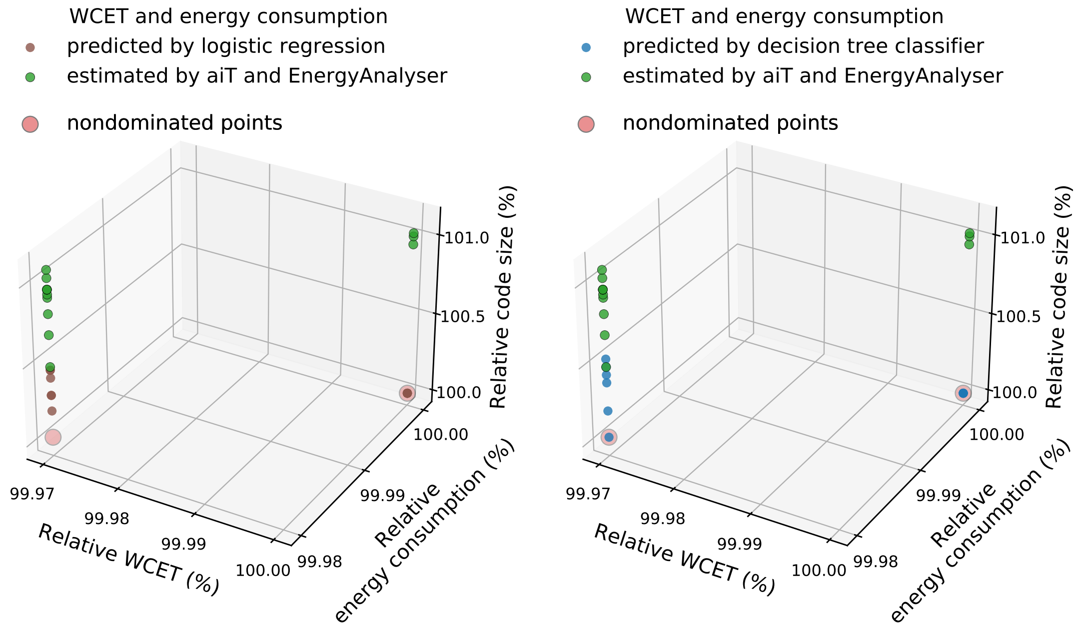
We notice that for 6 benchmarks, `adpcm_board`, `adpcm_verify`, `cjpeg_wrbmp`, `codecs_codhuff`, `codecs_dcodrle1`, `gsm_encode`, which have small search spaces compared to the other benchmarks (in our evaluation, these six benchmarks have the dimension of the search space less than or equal to 12), $\mathrm{MBPOA}_{\mathrm{AbsInt}}$ either show the same quality of solutions as the three other methods based on predictions or outperformed them. Recall that the seven benchmarks, for which all predictors failed to fit the model, also have quite small search spaces compared to the other benchmarks. This means that the evolutionary algorithm can gain from using prediction models if the dimension of the search space is large enough. (In our evaluation, the dimension is greater than 85.)

when comparing the methods based on predictions ($\mathrm{MBPOA}_{\mathrm{LogReg}}$, $\mathrm{MBPOA}_{\mathrm{DTree}}$, and $\mathrm{MBPOA}_{\mathrm{ABoost}}$), we observe that

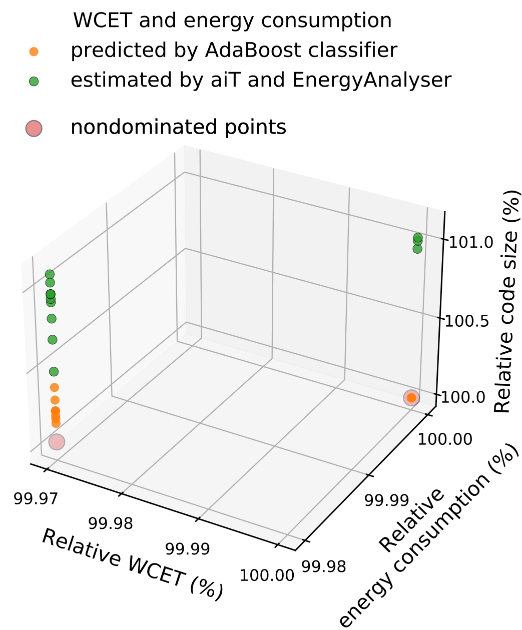
- for 20 out of 55 benchmarks like, e.g. `codecs_dcodrle1`, `durbin`, or `epic`, all three methods showed the same quality of Pareto fronts;
- for 16 benchmarks like, e.g. `adpcm_board`, `adpcm_verify`, or `atax`, $\mathrm{MBPOA}_{\mathrm{DTree}}$ outperformed $\mathrm{MBPOA}_{\mathrm{LogReg}}$ and $\mathrm{MBPOA}_{\mathrm{ABoost}}$;
- for 5 benchmarks, `cholesky`, `gramschmidt`, `jacobi-2d-imper`, `lu`, and `mvt`, $\mathrm{MBPOA}_{\mathrm{ABoost}}$ outperformed $\mathrm{MBPOA}_{\mathrm{LogReg}}$ and $\mathrm{MBPOA}_{\mathrm{DTree}}$;
- for 8 benchmarks like, e.g. `3mm`, `expint`, or `fdtd-2d`, $\mathrm{MBPOA}_{\mathrm{LogReg}}$ outperformed $\mathrm{MBPOA}_{\mathrm{DTree}}$ and $\mathrm{MBPOA}_{\mathrm{ABoost}}$;
- for 3 benchmarks, `cnt`, `doitgen`, and `symm`, $\mathrm{MBPOA}_{\mathrm{DTree}}$ and $\mathrm{MBPOA}_{\mathrm{ABoost}}$ showed the same results (with comparison tolerance 0.05) and outperformed $\mathrm{MBPOA}_{\mathrm{LogReg}}$;
- for 3 benchmarks, `cjpeg_wrbmp`, `trellis`, and `v32mod_cnoise`, $\mathrm{MBPOA}_{\mathrm{LogReg}}$ and $\mathrm{MBPOA}_{\mathrm{ABoost}}$ showed the same results and outperformed $\mathrm{MBPOA}_{\mathrm{DTree}}$.

From these results, we can state that, in general, decision tree classifier is the best one for our problem, since, for most of the benchmarks, the classifier led to almost the same or better results than the two other classifiers. $\mathrm{MBPOA}_{\mathrm{DTree}}$ was outperformed by $\mathrm{MBPOA}_{\mathrm{LogReg}}$ or $\mathrm{MBPOA}_{\mathrm{ABoost}}$ only in the case of 16 out of 55 benchmarks.

Similar to Section 6.4.3, Figures 7.15 and 7.16 present Pareto fronts found for two exemplary benchmarks, `3mm` and `cjpeg_wrbmp`, when executing each method 10 times.



(a) MBPOA$_{\text{LogReg}}$ and MBPOA$_{\text{AbsInt}}$ (b) MBPOA$_{\text{DTree}}$ and MBPOA$_{\text{AbsInt}}$



(c) MBPOA$_{\text{ABoost}}$ and MBPOA$_{\text{AbsInt}}$

Figure 7.15: Pareto fronts returned by MBPOA with estimated and predicted objectives for the benchmark `3mm` in 10 runs. 100% corresponds to the original WCET, energy consumption, and code size.

Solutions returned by $\mathsf{MBPOA}_{\mathsf{LogReg}}$ are shown in brown, by $\mathsf{MBPOA}_{\mathsf{DTree}}$ in blue, by $\mathsf{MBPOA}_{\mathsf{ABoost}}$ in orange, and by $\mathsf{MBPOA}_{\mathsf{AbsInt}}$ in green. Nondominated points are shown in red. The x-axis shows relative WCET, the y-axis – relative energy consumption, and the z-axis – relative code size. 100% corresponds to the benchmarks' original WCET, energy consumption, and code size.

For the benchmark `3mm`, Figure 7.15 presents Pareto fronts returned by $\mathsf{MBPOA}_{\mathsf{AbsInt}}$, nondominated points, and Pareto fronts returned by $\mathsf{MBPOA}_{\mathsf{LogReg}}$, $\mathsf{MBPOA}_{\mathsf{DTree}}$, or $\mathsf{MBPOA}_{\mathsf{ABoost}}$.

The set of nondominated points of this benchmark consists of two points, but $\mathsf{MBPOA}_{\mathsf{AbsInt}}$ failed to find any nondominated points in 10 runs. All solutions returned by $\mathsf{MBPOA}_{\mathsf{AbsInt}}$ have a larger code size than the solutions returned by the three other methods.

We observe the following results for the methods based on predictions:

- according to Figure 7.15(a), $\mathsf{MBPOA}_{\mathsf{LogReg}}$ found one nondominated point; it was found in 8 out of 10 runs;

- according to Figure 7.15(b), $\mathsf{MBPOA}_{\mathsf{DTree}}$ found two nondominated points; it found both nondominated points in 1 out of 10 runs and only the right point in 5 runs, and it failed to find any nondominated point in 4 runs;

- according to Figure 7.15(c), $\mathsf{MBPOA}_{\mathsf{ABoost}}$ found one nondominated point; it was found in 3 out of 10 runs.

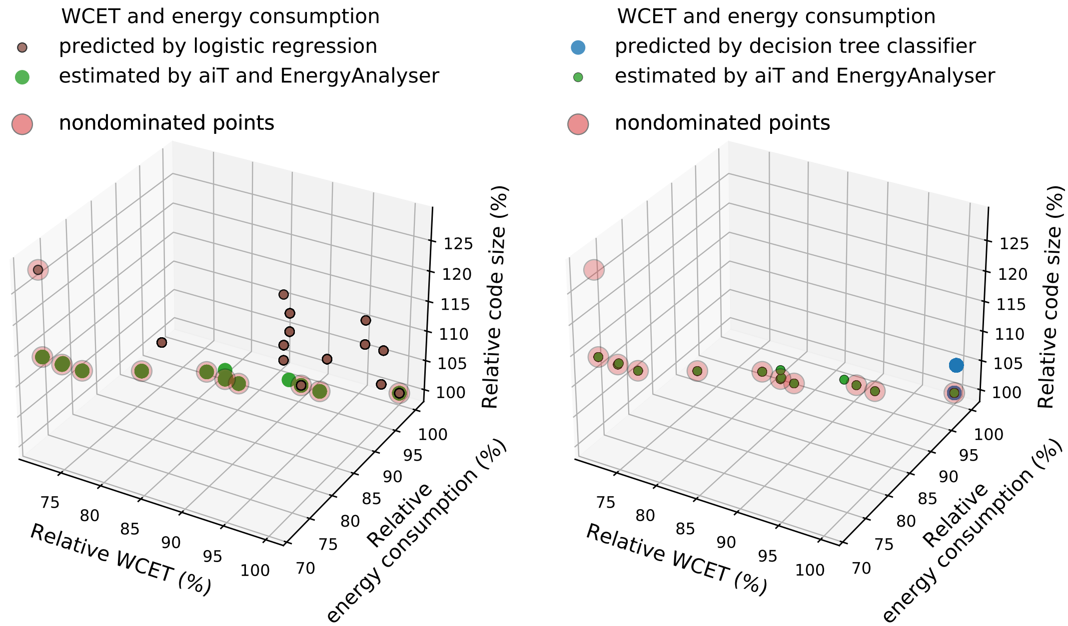
This explains the quality indicators from Figure 7.14 for this benchmark:

- the three methods based on predictions outperformed the original method based on aiT and EnergyAnalyser estimations, since $\mathsf{MBPOA}_{\mathsf{AbsInt}}$ failed to find any nondominated point;

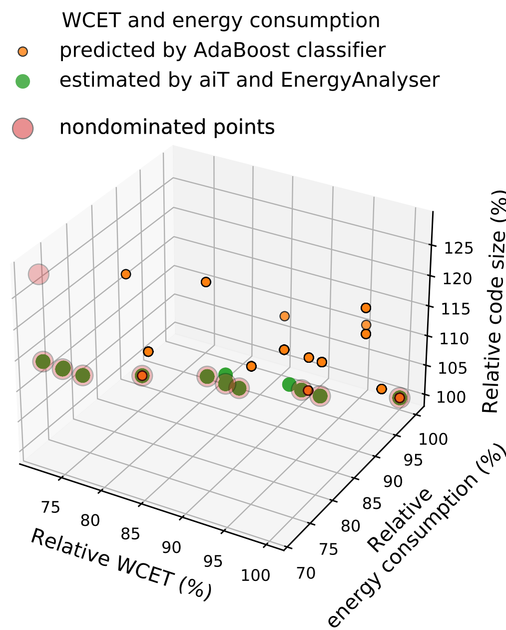
- among the three methods based on predictions, $\mathsf{MBPOA}_{\mathsf{LogReg}}$ showed the lowest coverage and the highest nondominance ratio, since it found a nondominated point in 8 runs, whereas $\mathsf{MBPOA}_{\mathsf{DTree}}$ and $\mathsf{MBPOA}_{\mathsf{ABoost}}$ found it in 6 and 3 runs, respectively. Although $\mathsf{MBPOA}_{\mathsf{DTree}}$ found one nondominated point which was missed by all other methods, it found the point only in one run, so the quality indicators for $\mathsf{MBPOA}_{\mathsf{LogReg}}$ are still better than the quality indicators for $\mathsf{MBPOA}_{\mathsf{DTree}}$.

Figure 7.16 presents Pareto fronts returned by $\mathsf{MBPOA}_{\mathsf{AbsInt}}$, nondominated points, and Pareto fronts returned by $\mathsf{MBPOA}_{\mathsf{LogReg}}$, $\mathsf{MBPOA}_{\mathsf{DTree}}$, or $\mathsf{MBPOA}_{\mathsf{ABoost}}$ for the benchmark `cjpeg_wrbmp`.

For this benchmark, the set of nondominated points consists of 11 points and $\mathsf{MBPOA}_{\mathsf{AbsInt}}$ found most of the nondominated points, it failed to find only one point (the left upper corner of Figure 7.16(a)), which $\mathsf{MBPOA}_{\mathsf{LogReg}}$ found.



(a) $\mathrm{MBPOA}_{\mathrm{LogReg}}$ and $\mathrm{MBPOA}_{\mathrm{AbsInt}}$ (b) $\mathrm{MBPOA}_{\mathrm{DTree}}$ and $\mathrm{MBPOA}_{\mathrm{AbsInt}}$



(c) $\mathrm{MBPOA}_{\mathrm{ABoost}}$ and $\mathrm{MBPOA}_{\mathrm{AbsInt}}$

Figure 7.16: Pareto fronts returned by MBPOA with estimated and predicted objectives for the benchmark `cjpeg_wrbmp` in 10 runs. 100% corresponds to the original WCET, energy consumption, and code size.

When comparing the methods based on predictions, we notice:

- according to Figure 7.16(a), $\mathsf{MBPOA}_{\mathsf{LogReg}}$ found three nondominated points. In each run, the algorithm found two nondominated points; it found the nondominated point from the left upper corner only in one run;
- according to Figure 7.16(b), $\mathsf{MBPOA}_{\mathsf{DTree}}$ returned only two solutions and one of them is nondominated (the right bottom corner). The algorithm found the nondominated solution in all 10 runs;
- according to Figure 7.16(c), $\mathsf{MBPOA}_{\mathsf{ABoost}}$ found two nondominated points. The algorithm found the point from the right bottom corner in all 10 runs, and the other nondominated point in 5 runs.

These results coincide with the results from Figure 7.14, which shows that $\mathsf{MBPOA}_{\mathsf{AbsInt}}$ significantly outperforms the methods based on predictions.

7.5 Conclusion

In this chapter, we presented a method to predict WCET and energy consumption at compile time. Proposed prediction models utilize machine learning, and one can integrate them into any compiler-based optimization since they are independent of any optimization-specific features and rely only on the search and objective spaces of a problem. The WCC compiler fits two independent prediction models for WCET and energy consumption for every input program since our goal was to minimize the number of training points which, in its turn, minimizes the number of expensive evaluations of WCET and energy consumption, whereas multi-output approaches require a large training set to build a prediction model.

We demonstrated the advantage of the prediction models by utilizing them within function inlining that we formulated as a multiobjective problem with three objectives: WCET, energy consumption, and code size. To choose a machine learning algorithm, we showed that function inlining can be considered as a classification problem because we observed a limited number of unique objective values.

We tuned the parameters of nine well-known classifiers for each benchmark and showed that logistic regression, decision tree classifier, and AdaBoost classifier based on decision trees resulted in high prediction accuracy. Since the quality of a prediction model depends on the size of a training set, we presented an algorithm that finds a prediction model with a high accuracy and as few training points as possible.

Solving the function inlining problem, we used an evolutionary algorithm and substituted time-consuming WCET and energy consumption estimations with quicker predictions. We compared Pareto fronts produced by four methods:

1. the evolutionary algorithm with estimated WCET and energy consumption;
2. the evolutionary algorithm with WCET and energy consumption predicted by logistic regression;
3. the evolutionary algorithm with WCET and energy consumption predicted by decision tree classifier;
4. the evolutionary algorithm with the WCET and energy consumption predicted by AdaBoost classifier based on decision trees.

The prediction models significantly speed up the evolutionary algorithm while solving the multiobjective function inlining problem. For all evaluated benchmarks, the three selected classifiers showed the average saving of 3 h in optimization runtime.

For many benchmarks, predictions were accurate enough to find Pareto fronts that are not worse than Pareto fronts found by the evolutionary algorithm with estimated objectives. Since for most of the benchmarks, the evolutionary algorithm combined with decision tree classifier showed the same or a better quality of Pareto fronts than the algorithm combined with logistic regression or AdaBoost classifier, we suggest to try decision tree classifier first for a given program, but, in general, there is no obvious winner between these three classifiers.

For 7 out of 62 benchmarks, all nine considered classifiers failed to fit prediction models. For other six benchmarks with the dimension of the search space less than 25, the evolutionary algorithm with estimated objectives outperformed the methods based on predictions in terms of solution quality. For all these benchmarks, function inlining results in many unique values of WCET and energy consumption. In this case, regression problems should be considered instead of classification problems. We leave consideration of such benchmarks for future work since it is another large scientific topic.

In this chapter, we presented that the search space of most benchmarks is large but a set of observed unique WCET and energy consumption values is small, i.e. changing a search vector along certain dimensions keeps WCET and energy consumption unchanged. Decision tree classifier is among the classifiers that showed the best prediction accuracy since, most probably, it can identify the redundant dimensions of the search space due to the way that it constructs decision trees. This motivates a search space reduction technique described in the next Chapter 8. It aims to identify the redundant dimensions of the search space and to provide an evolutionary algorithm with a reduced search space to speed it up. Moreover, in Chapter 9, we combine the reduction technique with the prediction models presented in this chapter and show that their combination can improve the quality of Pareto front and speed up the evolutionary algorithm even further.

8 Search Space Reduction

In Chapter 6, we discussed a multiobjective compiler-based optimization for hard real-time systems. We considered three objectives: code size, WCET, and energy consumption. A compiler computed code size, and static analysers estimated WCET and energy consumption. We showed that an evolutionary algorithm can find trade-offs between the objectives but the optimization process is time-consuming for many benchmarks. The algorithm extensively evaluates the objectives while solving a problem, whereas WCET and energy consumption estimations are time-consuming. Algorithm's runtime might be up to 6 h, and tuning of the algorithm's parameters might take up to some weeks for a large benchmark.

In Chapter 7, we presented prediction models based on machine learning to substitute expensive estimations of WCET and energy consumption with cheaper predictions while executing an evolutionary algorithm at compile time. For many benchmarks, we observed only a few unique values of the objectives, although the dimensions of the search spaces are large. This means that some dimensions of a search space might be redundant and that changing the values of a search vector along redundant dimensions keeps WCET and energy consumption unchanged. This motivates a search space reduction technique presented in this chapter. The idea is to reduce the search space of a problem at compile time and provide an evolutionary algorithm with a smaller search space. This should lead to fewer evaluations of the objectives and speed up the exploration process of the evolutionary algorithm. Similar to the previous Chapter 7, our goal is to reduce the search space of a problem without relying on any specific features of optimization but only by using the search and objective spaces of a problem. In this chapter, we reduce a search space regarding time-consuming objectives, particularly WCET and energy consumption. We ignore code size in our reduction technique since its computation is simple and cheap at compile time.

We organize the chapter as follows: Section 8.1 presents related work; Section 8.2 motivates our approach and presents a method to reduce a search space; Section 8.3 contains evaluation results; Section 8.4 concludes the chapter.

8.1 Related Work

To the best of our knowledge, no studies have been presented to reduce the search space of a specific compiler-based optimization. But few approaches were

proposed to reduce the search space while identifying an optimal sequence of compiler-based optimizations.

One way to find an optimal sequence of optimizations is through *iterative compilation*: many versions of a program are generated by changing the sequence of compiler-based optimizations, the quality of every code version is measured, and the best version is chosen. Iterative compilation makes decisions on a generated code rather than estimations of its characteristics. Iterative compilation approaches differ in their iterative strategies as described by, e.g. Kisuki et al. [Kis+00] or Knijnenburg, Kisuki, and O'Boyle [KKO02]. All approaches are time-consuming due to the necessity to compile a program many times, so some techniques to reduce its search space have been proposed.

Thomson et al. [Tho+10] presented a technique to reduce the search space of iterative compilation. The search space of the problem comprises program's features, e.g. the number of certain instructions within a program. Average-case execution time is an objective of the problem. To reduce the search space, the authors represented the vectors of the reduced space as a linear combination of the original vectors by applying Principal Component Analysis (PCA) proposed by Pearson [Pea01] in 1901.

PCA aims to find a lower dimensional subspace that minimizes the sum of the squared distances from given data points $\mathbf{x}_i \in \mathbb{R}^n, i = \overline{1, d}$ to their projections θ_i in the subspace. Collins, Dasgupta, and Schapire [CDS01] mentioned probabilistic interpretation of PCA, which says that a given data point $\mathbf{x}_i$ is a random draw from an unknown unit Gaussian distribution with mean $\boldsymbol{\theta}_i \in \mathbb{R}^n$. PCA searches for the parameters $\boldsymbol{\theta}_i$ that maximize the likelihood of data and lie in a low-dimensional space, i.e. PCA says that the given points $\mathbf{x}_i$ are the true low-dimensional points $\boldsymbol{\theta}_i$ corrupted with some noise. PCA assumes that the noise is Gaussian which may be inappropriate for search spaces represented by bit vectors, which is the case for many compiler-based optimizations.

Triantafyllis et al. [Tri+03] proposed Optimization-Space Exploration (OSE), which is another iterative compilation technique. The authors utilized correlations between different optimizations to reduce the search space: a compiler tests a small set of optimizations, gets feedback in terms of average-case execution time, and chooses optimizations to be added to the set.

Purini and Jain [PJ13] considered the problem of finding an optimal sequence of compiler-based optimizations to improve average-case execution time. The authors presented a downsampling technique to reduce a search space represented by all possible optimization sequences. By using a training set, they classified programs and identified a subspace of optimization sequences for each class. Every sequence from the subspace was reduced further by iterating the sequence and eliminating optimizations that degrade program performance.

Triantafyllis et al. [Tri+03] as well as Purini and Jain [PJ13] proposed approaches that tackle the problem of identifying an optimal sequence of opti-

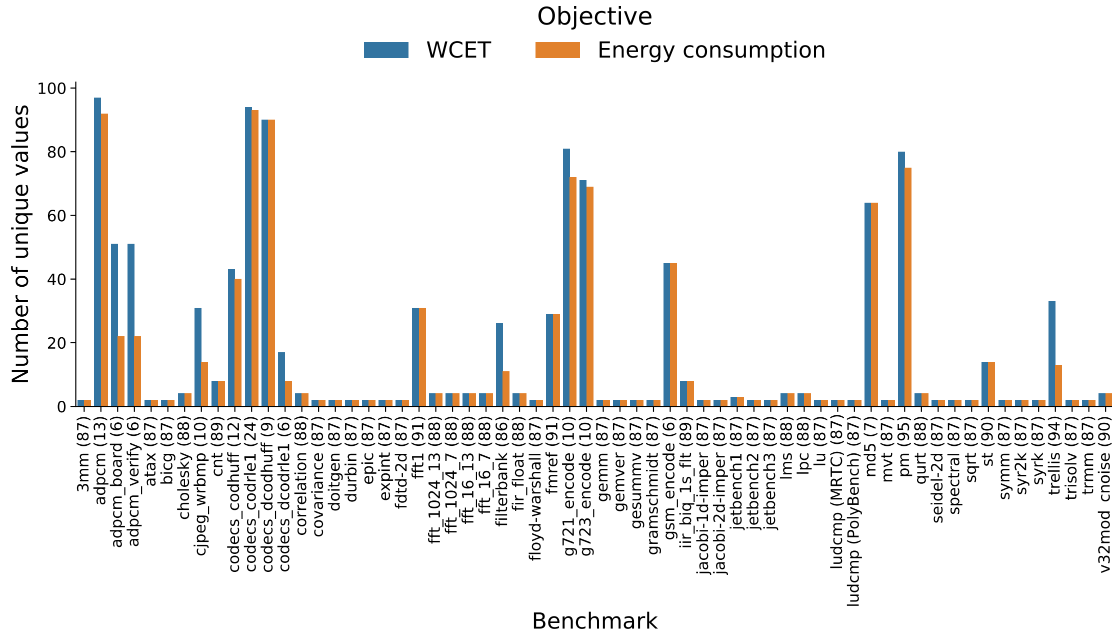


Figure 8.1: The number of unique WCET and energy consumption values among 100 random search vectors while performing function inlining. The dimensions of the search spaces are specified in parentheses next to the benchmarks' names.

mizations, but our reduction technique – described in the next section – is general enough to be applied to any compiler-based optimization. Our approach relies on the search and objective spaces of a problem and ignores any specific optimization features. Moreover, in contrast to Thomson et al. [Tho+10], our method can be applied to binary-encoded search spaces, which is the case for many compiler-based optimizations.

8.2 Reduction Method

Similar to the previous Chapter 7, this chapter considers n-dimensional search vectors $\mathbf{x} \in \mathbb{R}^n$, where the coordinates of a vector $\mathbf{x}$ are called *features*. We aim to identify those features that influence WCET and energy consumption, i.e. where changing the value of a feature leads to the change of at least one objective; we call such features *important*.

To motivate our reduction technique described in the next sections, Figure 8.1 presents the number of unique WCET and energy consumption values when performing function inlining introduced in Section 6.3: we generated 100 random search vectors for the function inlining problem, estimated the WCET and

energy consumption of the search vectors by using the static analysers aiT and EnergyAnalyser (for each search vector, we inlined each function call that correspond to the value 1 in the search vector, and the analysers estimated the objectives for the resulting program), and counted the number of unique objective values. In the figure, the x-axis presents benchmarks and the dimensions of their search spaces in parentheses next to the benchmarks' names; the y-axis shows the number of unique values of WCET (blue) and energy consumption (orange).

From the figure, we see that for many benchmarks, the number of unique values for both objectives is much smaller than the dimension of the search space, e.g. for the benchmark `3mm`, the dimension of the search space is 87, whereas we observed only three unique values for each objective. This means that only some features from the search space influence the objectives. We notice similar results for many considered benchmarks, so to reduce a search space, we identify features that influence the objectives (important features) and remove the remaining redundant features from the search space. We explain the limited number of observed objective values by the fact that the static analysers aiT and EnergyAnalyser identify a Worst-Case Execution Path (WCEP) to estimate WCET and energy consumption, respectively. Function inlining may change the WCEP, but if inlined functions lie outside the WCEP, WCET and energy consumption might be kept unchanged.

We aim to reduce a search space by taking into account two objectives: WCET and energy consumption. But since we want to keep all features that influence at least one objective, the general workflow to reduce the search space is the following:

1. find a set of important features for WCET;
2. find a set of important features for energy consumption;
3. merge both sets of important features;
4. keep the features from the merged set and remove the others.

So we identify important features for each objective independently.

This section is organized as follows: Section 8.2.1 describes strategies to identify important features, Section 8.2.2 presents the search space reduction method.

8.2.1 Selection Strategy

If a feature is important for at least one objective, the feature should remain in a reduced search space. For this reason, we select important features for each objective and then collect the important features of all objectives into one set.

Algorithm 4 presents a procedure to find important features for a problem with a binary-valued search space and a single objective. Many compiler-based

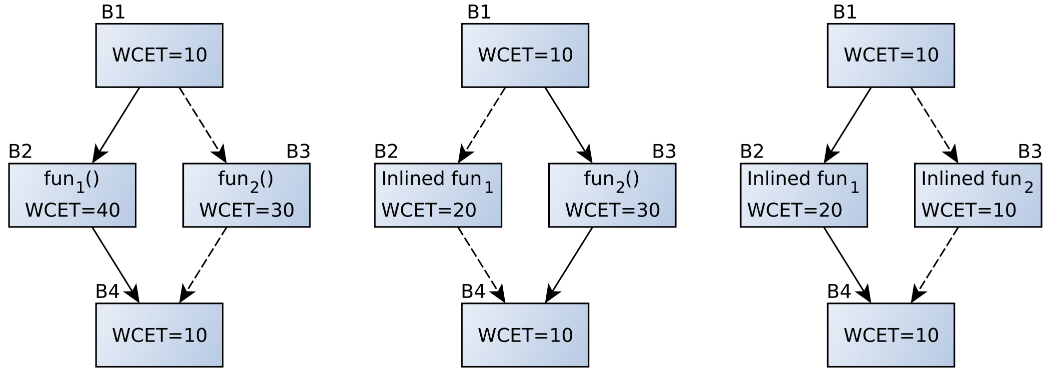
Algorithm 4 Selection of important features.

```
1:  Input: search space dimension n,
2:         randomly generated training set P = {(x, y) : x ∈ X, y = f(x)}, where
3:         f : X → ℝ is an objective function.
4:
5:  Output: indices of important features.
6:
7:  Normalize y to lie in the interval of length 1.
8:  for j = 1,n (overlined) do                                  ▷ For each feature.
9:      P_0 = {(x, y) ∈ P : x_j = 0}                            ▷ See Remark 8.1.
10:     P_1 = {(x, y) ∈ P : x_j = 1}
11:
12:     if |P_0| > 0 and |P_1| > 0 then                         ▷ Sets are not empty.
13:         mean_0 = 1/|P_0| Σ_{(x,y)∈P_0} y
14:         mean_1 = 1/|P_1| Σ_{(x,y)∈P_1} y
15:         DIFF_j = |mean_0 − mean_1|
16:     else
17:         DIFF_j = 1
18:
19: q = GetLowerBound({DIFF_j}_1^n)                             ▷ Get a lower bound for DIFF_j by
20:                                                             ▷ using a strategy from Table 8.2.
21: return {j ∈ {1, 2, ..., n} : DIFF_j > q}.
```

optimizations can be formulated in such a way that the search space is restricted to binary values, but in Remark 8.1, we explain how to extend the proposed method to a more general case, where a search space is restricted to take values from a finite set of possible values.

The algorithm takes the dimension of the search space n and a randomly generated training set consisting of the following pairs: a search vector $\mathbf{x} \in X$ and the corresponding value of a scalar objective function f. The algorithm returns the indices of important features.

First, the objective values of the training set are normalized to lie in the interval of length 1. Then, for each feature j, sets P_0 and P_1 contain all those training points with $x_j = 0$ and $x_j = 1$, respectively (Lines 9 and 10). The algorithm calculates the mean of the objective for the sets P_0 and P_1 and the absolute difference DIFF_j between the mean values (Lines 13–15). If the difference is large enough, the algorithm assumes that the feature influences the objective (Lines 20 and 21). If the training set P contains all points with the same value for the feature j, the algorithm cannot state or decline that the feature influences the objective, so at



(a) Original WCETs. (b) WCETs after inlining $\mathtt{fun}_1$. (c) WCETs after inlining $\mathtt{fun}_1$ and $\mathtt{fun}_2$.

Figure 8.2: Example of WCEP switch due to function inlining. The CFG consists of four basic blocks and their WCETs. The solid edges represent the WCEP. The basic block B_2 contains a call of $\mathtt{fun}_1$ and the basic block B_3 contains a call of $\mathtt{fun}_2$. Inlining of $\mathtt{fun}_1$ and $\mathtt{fun}_2$ decreases the WCET.

Line 17 the algorithm assigns the maximum possible value to $DIFF_j$ to guarantee that the feature j is marked as important.

Remark 8.1
In Algorithm 4, we assume that a decision space X *is represented by bit vectors. If* X *consists of vectors that take values from a finite set of values* $\{l_i\}_{i=1}^{L}$, *we define the sets* P_i *and mean values* mean_i *at Lines 9–14 as follows:*

$$\begin{aligned} P_i &= \{(\mathbf{x}, y) \in P : x_j = l_i\} \quad \forall i = \overline{1, L}, \\ \mathrm{mean}_i &= \frac{1}{|P_i|} \sum_{(\mathbf{x},y) \in P_i} y \quad \forall i = \overline{1, L} \ . \end{aligned} \tag{8.1}$$

In this case, at Line 15

$$DIFF_j = \max_{\substack{q=\overline{1,L} \\ r=\overline{1,L} \\ q \neq r}} |\mathrm{mean}_q - \mathrm{mean}_r| \ . \tag{8.2}$$

In Algorithm 4, we consider each feature inside the loop independent of other features, but it does not mean that we assume that the features are independent of each other. Their dependence is implicitly accounted for in mean_0 and mean_1,

Table 8.1: WCET changes due to function inlining applied to the program from Figure 8.2.

Description	Feature x_1	Feature x_2	WCET (cycles)
Neither fun_1 nor fun_2 is inlined.	0	0	60
Only fun_1 is inlined.	1	0	50
Only fun_2 is inlined.	0	1	60
Both fun_1 and fun_2 are inlined.	1	1	40

and if the training set is large enough, then a dependent feature will be also marked as important.

Figure 8.2 presents an example: the function inlining problem from Section 6.3 with only two function calls fun_1 and fun_2, and objective function WCET (see Figure 8.2(a)). The WCET of the original program is 60 cycles. In the original program, the function call fun_1 lies on the WCEP but the call fun_2 does not. Figure 8.2(b) shows that after inlining the function fun_1, the WCET decreases to 50 cycles, and the WCEP changes such that the call fun_2 is on the WCEP. If in addition to the inlined fun_1, we also inline fun_2, the WCET decreases further to 40 cycles as shown in Figure 8.2(c). This means that fun_2 becomes an important feature, if fun_1 is inlined.

Table 8.1 summarizes all possible combinations of fun_1 and fun_2 and the corresponding WCETs for our example. We denote by x_1 a feature corresponding to fun_1 and by x_2 a feature corresponding to fun_2.

In the example, a training set passed to Algorithm 4 consists of four samples presented in Table 8.1. When considering the first feature x_1 at Lines 8–17, we get

- $P_0 = \{(0,0),(0,1)\}$ and $P_1 = \{(1,0),(1,1)\}$;
- $mean_0 = \frac{1}{2}(60+60) = 60$ and $mean_1 = \frac{1}{2}(50+40) = 45$;
- $DIFF_1 = 15$.

Since the difference between the values of $mean_0$ and $mean_1$ is greater than 0, the algorithm can assume that the feature x_1 is important.

When considering the second feature x_2, we get

- $P_0 = \{(0,0),(1,0)\}$ and $P_1 = \{(0,1),(1,1)\}$;
- $mean_0 = \frac{1}{2}(60+50) = 55$ and $mean_1 = \frac{1}{2}(60+40) = 50$;
- $DIFF_2 = 5$.

In this case, $DIFF_2$ is smaller than $DIFF_1$, since when inlining fun_2, we decrease the WCET only if fun_1 is inlined.

Table 8.2: Strategies for the procedure GetLowerBound from Algorithm 4 to select important features.*

Name	Description	Lower bound
Outliers	Keep outliers.	$Q3 + 1.5 \cdot (Q3 - Q1)$
25%	Keep 25 % of features with the largest DIFF.	Q3
50%	Keep 50 % of features with the largest DIFF.	Q2
75%	Keep 75 % of features with the largest DIFF.	Q1
GTH0	Keep features with DIFF > 0.	0

*We denote by Q1, Q2, and Q3 the first, second, and third quartiles of a data set.

This example shows that if the training set is large enough, i.e. it contains search vectors with many different combinations of features' values, the difference $DIFF_j$ is larger than 0 even for dependent features.

At Line 20, Algorithm 4 relies on a function *GetLowerBound* which returns a lower bound for the absolute difference $DIFF_j$ to select important features. We consider five strategies listed in Table 8.2 to define the lower bound for $DIFF_j$.

Figure 8.3 presents an example of the strategies when selecting important features for the benchmark `md5` and objective WCET. The search space of the benchmark consists of seven features shown on the x-axis, i.e. the dimension of the search space is 7. The y-axis presents $DIFF_j$ computed by Algorithm 4 at Line 15 for each feature j. In the figure, each row shows results for one of the strategies from Table 8.2. Important features are shown in blue and redundant features in orange. On the right side of the figure, the blue box plots show the quartiles and outliers for a set of all $DIFF_j$ values.

When following the strategy Outliers, the algorithm returns only one important feature which is an outlier as shown in the box plot. When following the strategy 25%, 2 out of 7 features are important since $DIFF_j$ for these features is larger than the third quartile. Similarly, when following Strategies 50% and 75%, the chosen important features have $DIFF_j$ larger than the second and first quartiles, respectively. When following the strategy GTH0, all features are marked as important since for all features $DIFF_j > 0$.

The strategy Outliers produces the smallest set of important features, whereas the strategy GTH0 produces the largest set; less important features lead to a larger reduction of the search space.

The next step is to choose an optimal strategy for a problem. Our goal is to choose the strategy such that

1. the size of a training set in Algorithm 4 is as small as possible to minimize the number of expensive evaluations of the objective;

2. the search space is reduced as much as possible;

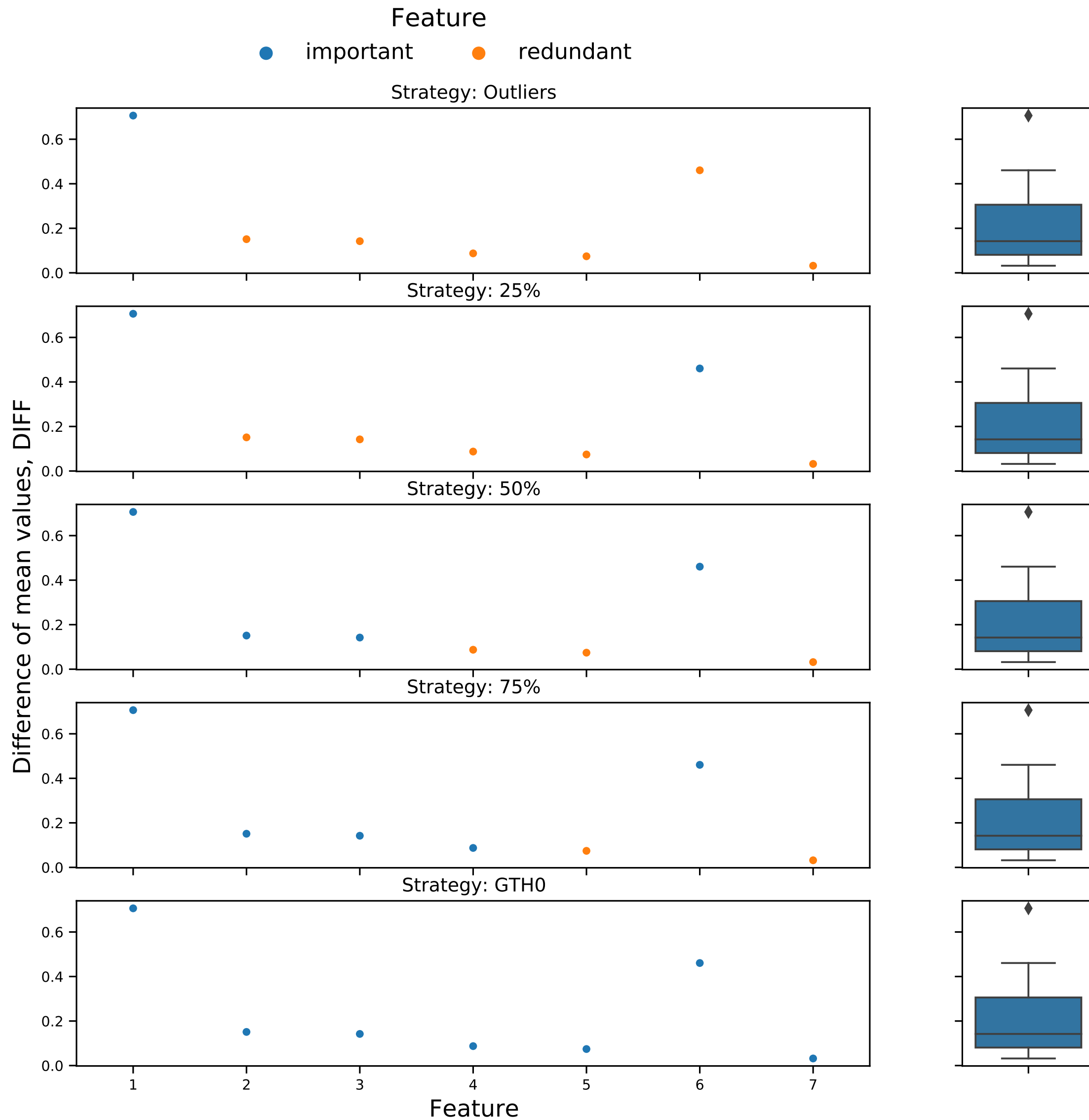


Figure 8.3: Example of the strategies from Table 8.2 for the benchmark `md5` and objective WCET.

3. the objective values computed on the reduced search space are as close as possible to the values computed on the original search space.

The last requirement from the list above serves as a measure to define the quality of a strategy. Similar to Section 7.4.2, we use MAE score metric.

Algorithm 5 Selection of important features by using the strategies from Table 8.2.

1: **Input:** search space X,
2: objective function $f : X \to \mathbb{R}$ to generate a training set,
3: minimum size M_{min} of the training set,
4: maximum size M_{max} of the training set,
5: step M_{step} to increase the size of the training set,
6: randomly generated test set $T = \{(\mathbf{x}, y) : \mathbf{x} \in X, y = f(\mathbf{x})\}$,
7: upper bound S_{limit} of MAE to terminate the algorithm.
8:
9: **Output:** indices of important features.
10:
11: **for** $\mathtt{Strategy} \in \{\mathtt{Outliers}, 25\%, 50\%, 75\%, \mathtt{GTH0}\}$ **do**
12: Generate a training set P of size M_{min} by using the objective function f.
13: **while** $|P| < M_{max}$ **do**
14: Get indices of important features by using $\mathtt{Strategy}$ and the training set P in Algorithm 4.
15: Reduce the search space X by keeping only important features.
16: For each $(\mathbf{x}, y) \in T$, define $\mathbf{x}'$ to be a vector from the reduced space corresponding to $\mathbf{x}$ and $y' = f(\mathbf{x}')$. ▷ See Remark 8.2.
17: Compute MAE defined in Equation (8.3).
18: **if** $S \leqslant S_{limit}$ **then**
19: **return** the indices of the important features.
20: **else**
21: Add M_{step} randomly generated training points to the set P.
22: **return** the indices of all features.

Definition 8.1
For $\{y_i\}_{i=1}^{I}$ and $\{y_i'\}_{i=1}^{I}$ being objective values computed on original and reduced search spaces, MAE is defined as follows:

$$\mathrm{MAE} = \frac{1}{I} \sum_{i=1}^{I} |y_i - y_i'| \quad . \tag{8.3}$$

MAE takes values from the interval $[0, 1]$ with the best value 0, meaning that the corresponding objective values coincide.

Algorithm 5 presents a procedure to choose the strategy from Table 8.2 that achieves the smallest MAE and requires the smallest training set. The input of the algorithm is the following:

- an objective function to generate a training set;
- an original search space X to be reduced;

- minimum and maximum sizes of training sets and a step size to increase a training set;
- a randomly generated test set T to compute the accuracy of reduction by using MAE defined in Equation (8.3). The test set consists of samples represented by pairs of search vectors $\mathbf{x} \in X$ and the corresponding objective value y;
- an upper bound S_{limit} of MAE to terminate the algorithm.

The algorithm outputs the indices of important features that should be kept in a reduced search space.

The algorithm iterates over the strategies from Table 8.2 (Line 11) by starting with the strategy Outliers that reduces the search space most aggressively. To speed up the algorithm, training and test sets must be as small as possible since their generation requires expensive evaluations of the objective. Thus, the algorithm utilizes training sets of different sizes – starting from the smallest one (Line 12) – to determine important features. By using a training set and a strategy, the algorithm gets important features from Algorithm 4 and computes MAE on the test set T (Lines 14–17). If the score is less than or equal to the upper bound S_{limit}, the algorithm returns currently found important features, otherwise, it increases the size of the training set and proceeds to the next iteration (Lines 18–21). If the algorithm fails to find a reduction strategy with a small MAE, it returns all features, meaning that all features are important.

Remark 8.2

At Line 16 of Algorithm 5, the objective function f must be well-defined on the reduced search space. E.g. when considering function inlining introduced in Section 6.3, for each test point $(\mathbf{x}, y) \in T$, we define $\mathbf{x}'$ by keeping the values of important features and setting all others to 0:

$$\mathbf{x}' = (x'_1, x'_2, \ldots, x'_n)$$
$$x'_k = \begin{cases} x_k, & k \in K, \\ 0, & \textit{otherwise}, \end{cases} \qquad \forall k = \overline{1, n}, \tag{8.4}$$

where K is a set of important features' indices and $x'_k = 0$ means that the corresponding function call remains unchanged. The decision vector $\mathbf{x}' \in X$, so $f(\mathbf{x}')$ is well-defined.

8.2.2 Search Space Reduction

The previous section describes how to select important features concerning a scalar function, but we aim to reduce a search space concerning both expensive objectives considered in the thesis: WCET and energy consumption.

In order to reduce the search space regarding both objectives, we

1. generate a test set for Algorithm 5;
2. get a set of important indices I_{WCET} for WCET from Algorithm 5;
3. get a set of important indices I_{Energy} for energy consumption from Algorithm 5;
4. merge the sets of the indices: $I = I_{WCET} \bigcup I_{Energy}$;
5. reduce the search space by keeping only features from the set I and ignoring all others. The dimension of the reduced search space is equal to the size of the set I.

This workflow allows to keep all those features that are important for at least one objective.

Remark 8.3
In this chapter, our goal is to reduce the search space of a problem concerning WCET and energy consumption estimated by the static analysers aiT and EnergyAnalyser As described in Section 2.1.3, both analysers rely on a path analysis which determines program paths with the highest WCET and energy consumption. The path might coincide for both objectives, i.e. important features for the objectives might also coincide.

In general, energy analyses rely on average-case simulations, i.e. important features for energy consumption most probably will differ from important features for WCET. Since we identify important features for each objective independently and merge all important features before reducing the search space, all important features are kept in the reduced search space. So one can also use the proposed reduction technique in this case, but we would expect to achieve poorer search space reduction.

After reducing a search space, we pass the reduced search space into an evolutionary algorithm. This should speed up the algorithm since it has to explore a much smaller search space, which means it requires fewer evaluations of the objectives.

The next section presents evaluation of the proposed reduction technique: we reduce the search space of the exemplary multiobjective function inlining problem described in Section 6.3, run an evolutionary algorithm on the reduced search space, and compare the resulting solution sets with those produced by running the algorithm on the original search space.

8.3 Evaluation

This section presents evaluation of the search space reduction described in the previous section for the multiobjective function inlining problem. Since the

proposed reduction method relies only on the search and objective spaces of a problem and does not use any specific characteristics of the function inlining optimization, it can be utilized within any compiler-based optimization.

The section is structured as follows: Section 8.3.1 describes experimental setups, Section 8.3.2 presents evaluation of Algorithm 4, which identifies important features, Section 8.3.3 presents evaluation of the search space reduction technique, Section 8.3.4 compares solution sets obtained by an evolutionary algorithm on reduced and original search spaces when solving the multiobjective function inlining problem.

8.3.1 Experimental Setup

We evaluate the proposed search space reduction technique by applying it to the function inlining problem presented in Section 6.3. We integrate the reduction Algorithms 4 and 5 into WCC described in Chapter 3. Similar to the previous chapters, we compile programs for an ARM Cortex-M0 processor architecture and enable optimization level O2. During evaluation, AbsInt's static analysers aiT and EnergyAnalyser version 20.10i estimate WCET and energy consumption, and WCC computes code size. We run all evaluations on a server with characteristics listed in Table 5.1. Similar to Chapter 6, we consider benchmarks from the benchmark suites PolyBench, MediaBench, MRTC, DSPstone, and UTDSP with search space dimensions greater than 5. All considered benchmarks are listed in Appendix B.

8.3.2 Selection of Important Features

This section evaluates Algorithm 4, which searches important features for WCET and energy consumption when considering the function inlining problem. In this section, we consider these two objectives independently since, in the final reduction technique, the algorithm is run independently for each objective.

For each objective, the number of important features selected by the algorithm should be as small as possible since fewer important features lead to larger reduction of the search space. In order to evaluate whether all important features are found, for each objective, we reduce the search space by keeping the important features selected for the objective. Then, by using a test set, we compare the objective computed on the reduced and original search spaces by using MAE defined in Definition 8.1. Similar to Section 7.4.2, we use a test set of 10 samples.

Since training sets are randomly generated during algorithm execution, we repeat the algorithm for each benchmark 10 times.

We consider the following configurations of the algorithm:

- training sets of the following sizes: $10, 20, 30, 40, 50, 60, 70$;

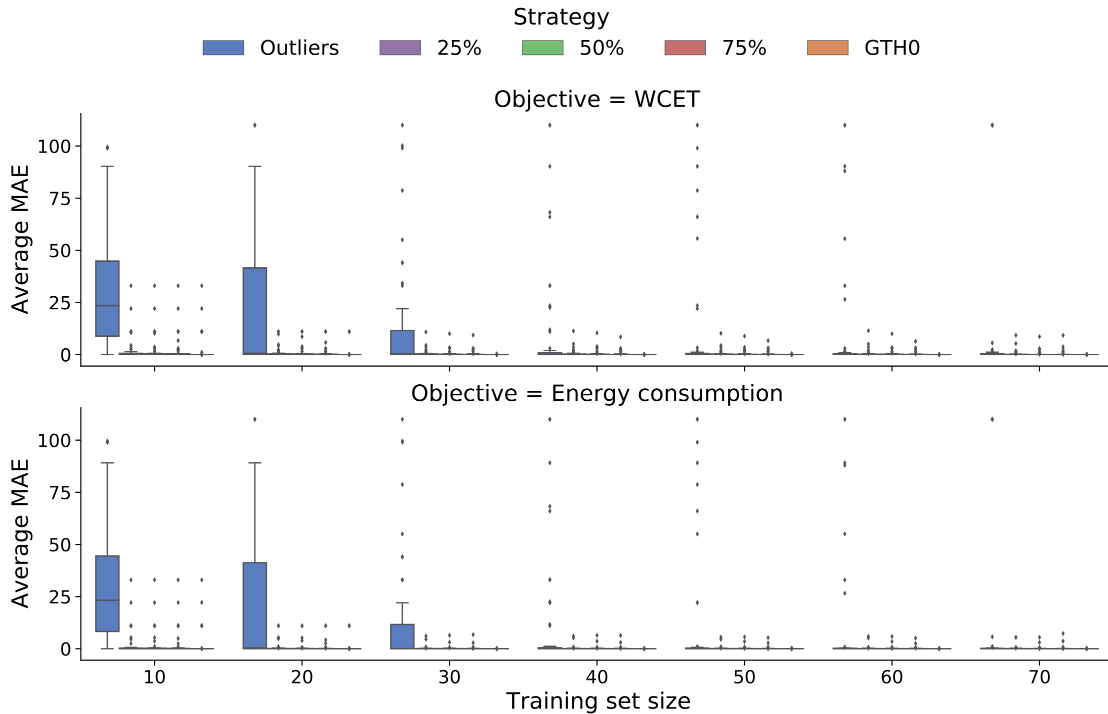


Figure 8.4: Average MAE for all considered benchmarks. MAE is computed for a test set after identifying important features and reducing the search space of a benchmark for WCET and energy consumption independently. Algorithm 4 identifies important features by using the strategies from Table 8.2. The best MAE value is 0. Each experiment was repeated 10 times with different training sets.

- objective: WCET or energy consumption;
- the strategies from Table 8.2 to compute the upper bound at Line 20.

Similar to Section 7.4.2, we consider the maximum size of training sets equal to 70 to reduce the number of expensive WCET and energy consumption evaluations. In this section, we set the minimum size of training sets to 10, since there are no restrictions in the algorithm regarding this[1].

Similar to Section 7.4.2, we consider relative WCET and energy consumption, where 100% corresponds to the objective of an original program without function inlining. If the algorithm marks all features of a search space as important for an objective, i.e. no search space reduction regarding this objective is possible, we set MAE to 110 in order to check for how many benchmarks the search space

[1]In Section 7.4.2, we set the minimum size of training sets to 20 due to k-fold cross-validation with $k = 5$.

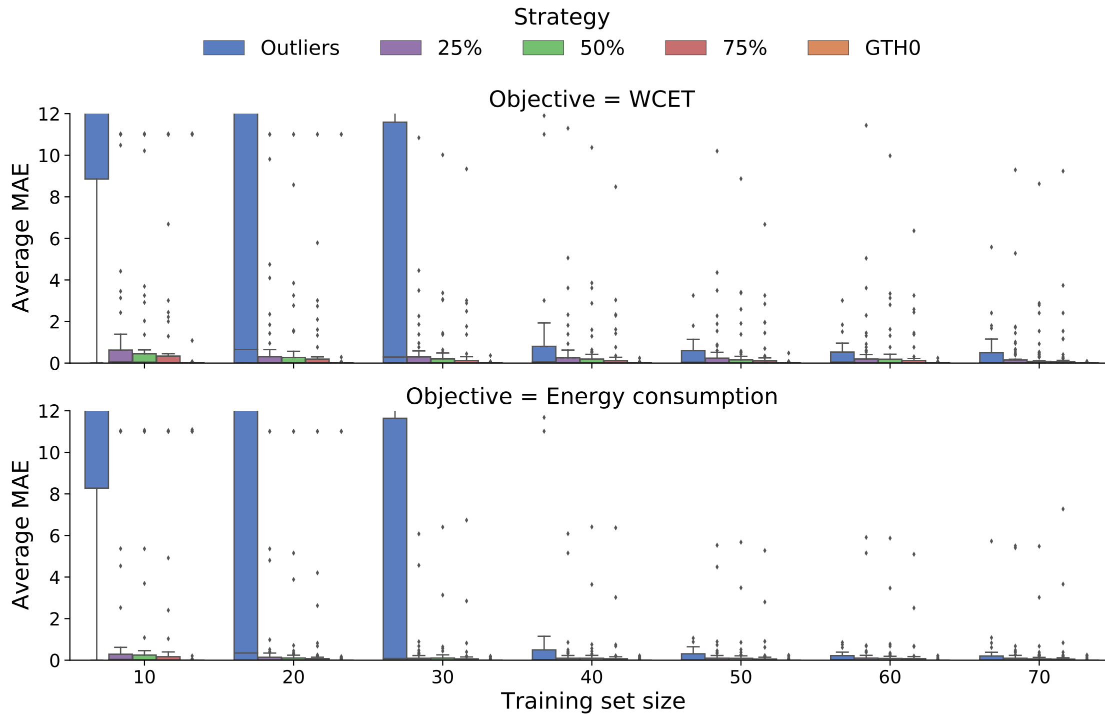


Figure 8.5: Average MAE from Figure 8.4 with the range of the y-axis changed to $[0, 12]$. The experiment was repeated 10 times with different training sets.

reduction technique might fail and whether this remains unchanged for all sizes of training sets.

Figure 8.4 presents the average MAE produced by running 10 times Algorithm 4 for each training set size, strategy, and benchmark. The x-axis shows the training set sizes from 10 to 70, the y-axis presents the average MAE with the best value 0 meaning that for all test samples, the objective computed on the reduced and original search spaces coincide. Each colour corresponds to a specific strategy from Table 8.2, e.g. blue corresponds to the strategy Outliers. For each fixed training set size and strategy, the box plot shows distribution of the average MAE over all benchmarks for two objectives: WCET (the top plot) and energy consumption (the bottom plot).

For both objectives, the results are very similar. The plots show that when the training set size equal to 10, the strategy Outliers results in the average MAE much larger than the other strategies for most benchmarks. The strategy Outliers requires at least 40 training points to produce a small average MAE for most benchmarks. For the training set size equal to 10 and the strategy Outliers, the average MAE is less than 110 for all benchmarks, i.e. for each benchmark, Algorithm 4 identified redundant features in at least one run.

All strategies different from Outliers result in the average MAE which is smaller than 22 for all benchmarks, except the benchmark `codecs_codhuff`. The general tendency is that the average MAE decreases as training set size increases.

To set the upper bound of MAE in Algorithm 5, Figure 8.5 shows the results from Figure 8.4 with the range of the y-axis changed from [0, 110] to [0, 12]. Recall that in Section 7.4.2, the results suggest the upper bound of MAE equal to 5. Figure 8.5 shows that for both objectives and for all training set sizes, the average MAE is less than 5 for most benchmarks, if the lower bound selection strategy differs from the strategy Outliers. Starting from 20 training samples, the algorithm with the strategy Outliers results in the average MAE less than 1 for at least 50 % of the considered benchmarks (the median of the box plots is below 1). We conclude that for the search space reduction technique, setting the upper bound of MAE to 5 is also an appropriate choice, since then, we expect to reduce the search spaces of (almost) all benchmarks.

8.3.3 Search Space Reduction

Section 8.2.2 describes a procedure to reduce a search space regarding WCET and energy consumption. In this section, we evaluate the proposed search space reduction technique. The technique relies on Algorithm 5 which identifies important features for each objective by testing different strategies from Table 8.2 and training sets of different sizes.

For each benchmark, we set the input parameters of Algorithm 5 to the following values:

- an original search space X: the search space of the function inlining problem for the benchmark;
- an objective function: WCET or energy consumption computed by aiT and EnergyAnalyser;
- minimum and maximum sizes of a training set and a step size to increase the training set are set – similar to Section 7.4.2 – to limit the number of expensive evaluations of the objectives: $M_{min} = 10, M_{max} = 70, M_{step} = 10$;
- a test set: a randomly generated test set of size 10 similar to Section 7.4.2;
- an upper bound: $S_{limit} = 5$ as the previous section suggests.

Since training sets are randomly generated during algorithm execution, we repeat 10 times Algorithm 5 for each benchmark.

Figure 8.6 presents strategies selected by Algorithm 5 repeated 10 times for WCET: the stack diagram (the top plot) shows the number of experiments in

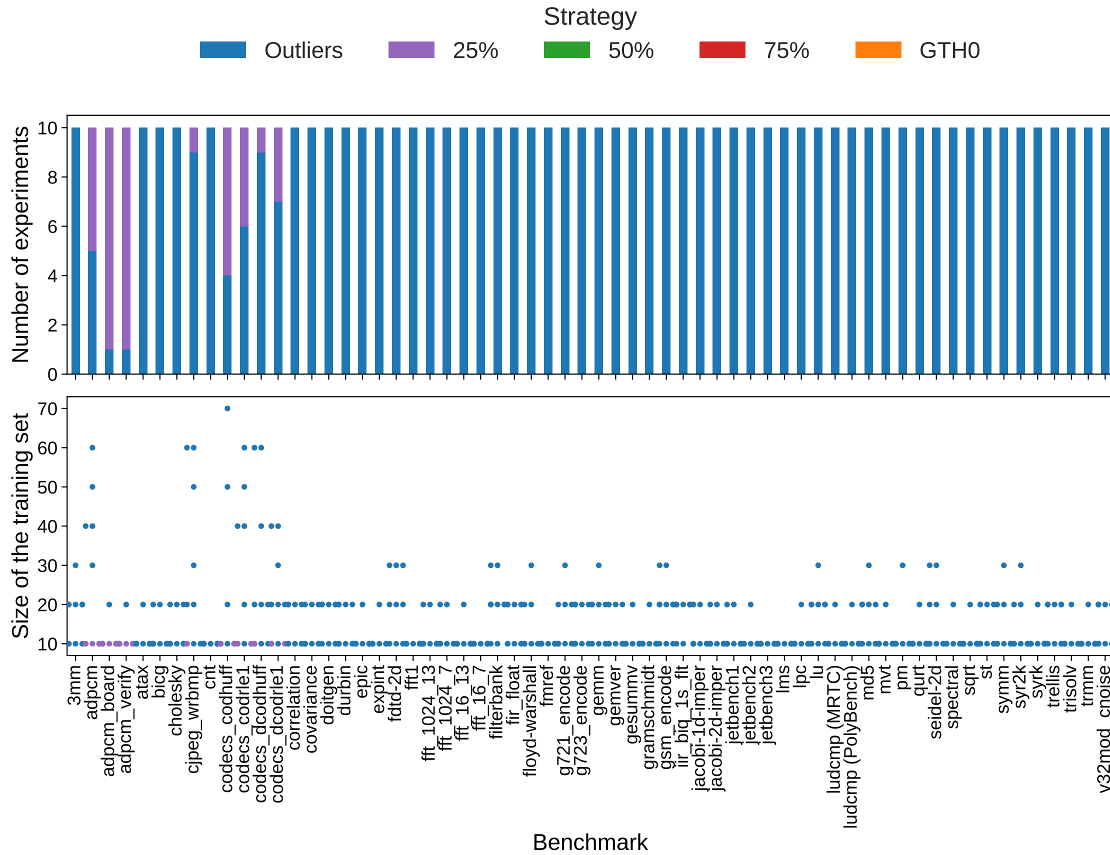


Figure 8.6: Statistics of resulting strategies when running Algorithm 5 for WCET. The experiment was repeated 10 times with different training sets.

which a certain strategy was chosen as the best strategy; the scatter plot (the bottom plot) shows the corresponding training set sizes ($|P|$) used to identify important features by the best strategy. The x-axis lists the benchmarks and the y-axes – the number of experiments (the top plot) and the training set sizes (the bottom plot). Each colour corresponds to a strategy specified in Table 8.2, e.g. the strategy `Outliers` – blue, the strategy 25% – purple, etc.

For 54 out of 62 benchmarks, the algorithm terminated at the strategy `Outliers` in all 10 experiments. For the remaining 8 benchmarks, the algorithm terminated at the strategy 25% in some experiments to achieve MAE less than 5. E.g. in the case of the benchmark `codecs_codhuff`, the algorithm kept the outliers in four experiments and 25 % of the features in six experiments. The figure shows that for all 62 tested benchmarks, the algorithm found important features in all 10 experiments (on the top plot, the height of all bars is 10).

The bottom plot of Figure 8.6 shows that final training set size is less than 40 in most cases. The strategy 25% required only 10 training points to achieve a small

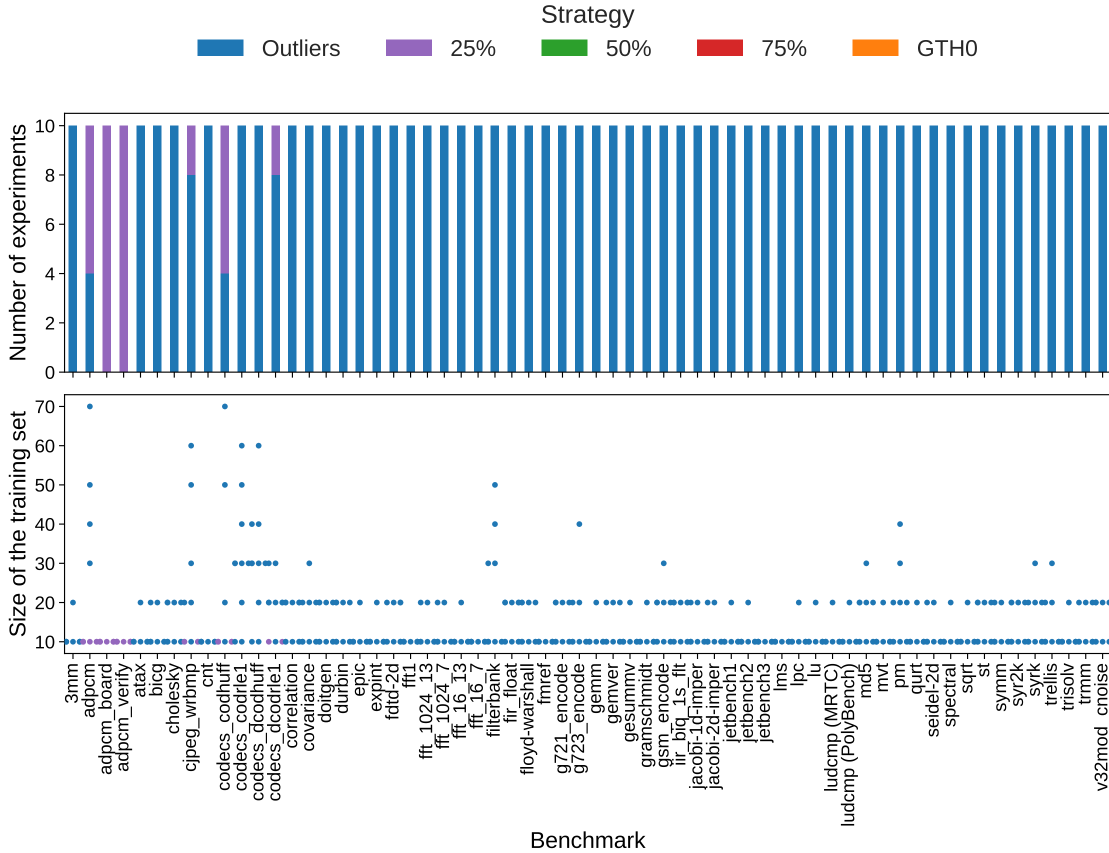


Figure 8.7: Statistics of resulting strategies when running Algorithm 5 for energy consumption. The experiment was repeated 10 times with different training sets.

MAE on a test set, whereas the strategy Outliers might require more training points for the same benchmark. E.g. for the benchmark `adpcm`, the strategy that keeps outliers required at least 30 training points, whereas the strategy 25% required only 10 training points. Algorithm 5 is developed in such a way that it uses as few as possible training points to identify important features. It tests all strategies with 10 training points, if the desirable MAE is not achieved, it adds 10 training points to the training set and tests the strategies again. It means, e.g. for the benchmark `adpcm`, in 5 experiments, 10 training points was enough for the algorithm to terminate after achieving a small MAE with the strategy Outliers. In the remaining 5 experiments, the algorithm with the strategy 25% required at least 30 training points to terminate.

Figure 8.7 presents the same statistics as Figure 8.6 but for energy consumption. The results are very similar for both objectives. For six benchmarks, the algorithm terminated at the strategy 25% in some experiments while identifying important

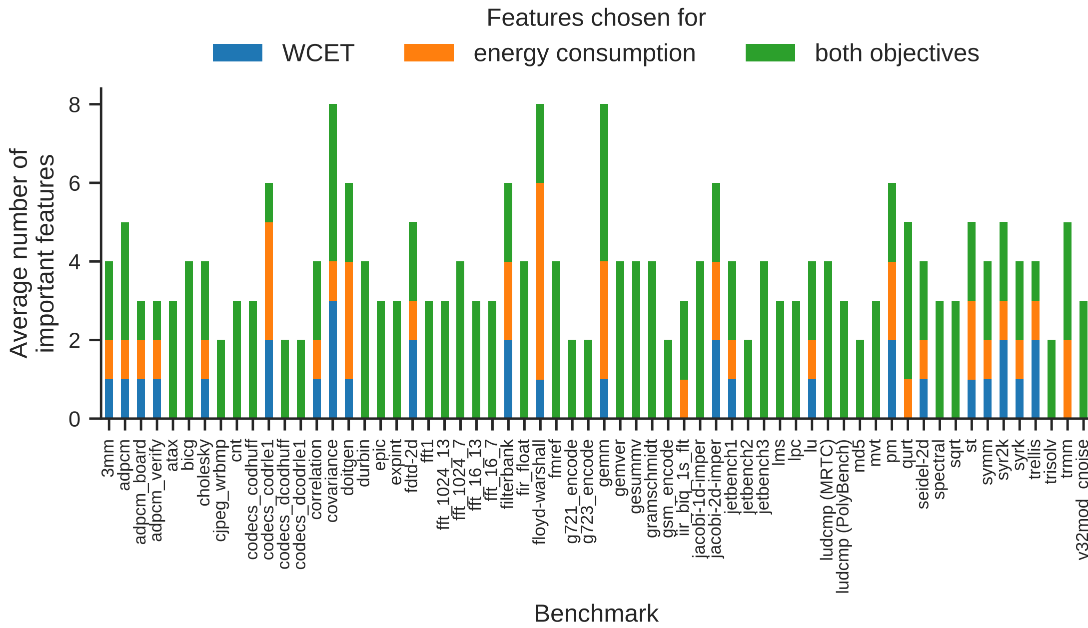


Figure 8.8: The number of important features selected by Algorithm 5 repeated 10 times with different randomly generated training sets.

features for energy consumption. For these six benchmarks, it also terminated at the strategy 25% in some experiments while identifying important features for WCET. In the case of the benchmarks `codecs_codrle1` and `codecs_dcodhuff`, for energy consumption, the algorithm terminated at the strategy `Outliers` in all experiments, whereas for WCET, it terminated at the strategy 25% in some experiments. For the benchmarks `adpcm_board` and `adpcm_verify`, for energy consumption, the algorithm terminated at the strategy 25% in all experiments, and for WCET, it terminated at the strategy 25% in 9 out of 10 experiments.

Figure 8.8 presents the average number of important features that were chosen only for WCET (blue), only for energy consumption (orange), and for both objectives (green) when running Algorithm 5 for each objective 10 times. The x-axis lists the benchmarks and the y-axis shows the number of important features. The figure shows that it is necessary to merge the sets of important features for the objectives as we describe in Section 8.2.2 to get a full set of important features, since they may differ for the objectives. E.g. in the case of the benchmark `3mm`, on average, the algorithm selected four features in total, two of them coincide for both objectives, and for each objective, one extra feature was selected.

Figure 8.9 shows in orange the average dimensions of reduced search spaces after reducing 10 times the search spaces of the benchmarks regarding both objectives as described in Section 8.2.2. It also shows in green the dimensions

Figure 8.9: Average dimensions of the search spaces reduced as described in Section 8.2.2. The experiment was repeated 10 times with different randomly generated training sets. The average dimension of the original search space is equal to the dimension of the original search space.

of the original search spaces. The x-axis presents the benchmarks, the y-axis the average dimensions of the search spaces[2].

We observe the following results:

- we reduce the search space dimensions of all evaluated benchmarks by 93 % on average;
- we achieve the poorest reduction of 72 % for the benchmark `adpcm`, where the dimension of the original search space is 13 and the average dimension of the reduced search space is 4;
- we achieve the largest reduction of 98 % for the benchmark `jetbench2`, where the dimension of the original search space is 87 and the average dimension of the reduced search space is 2.

Figure 8.10 presents the average MAE after reducing the search spaces independently 10 times as described in Section 8.2.2. MAE is presented for WCET (the top plot) and for energy consumption (the bottom plot). The x-axis presents the tested benchmarks, the y-axis shows the average MAE. The best MAE value

[2]The average dimension of a original search space is equal to the dimension of the original search space.

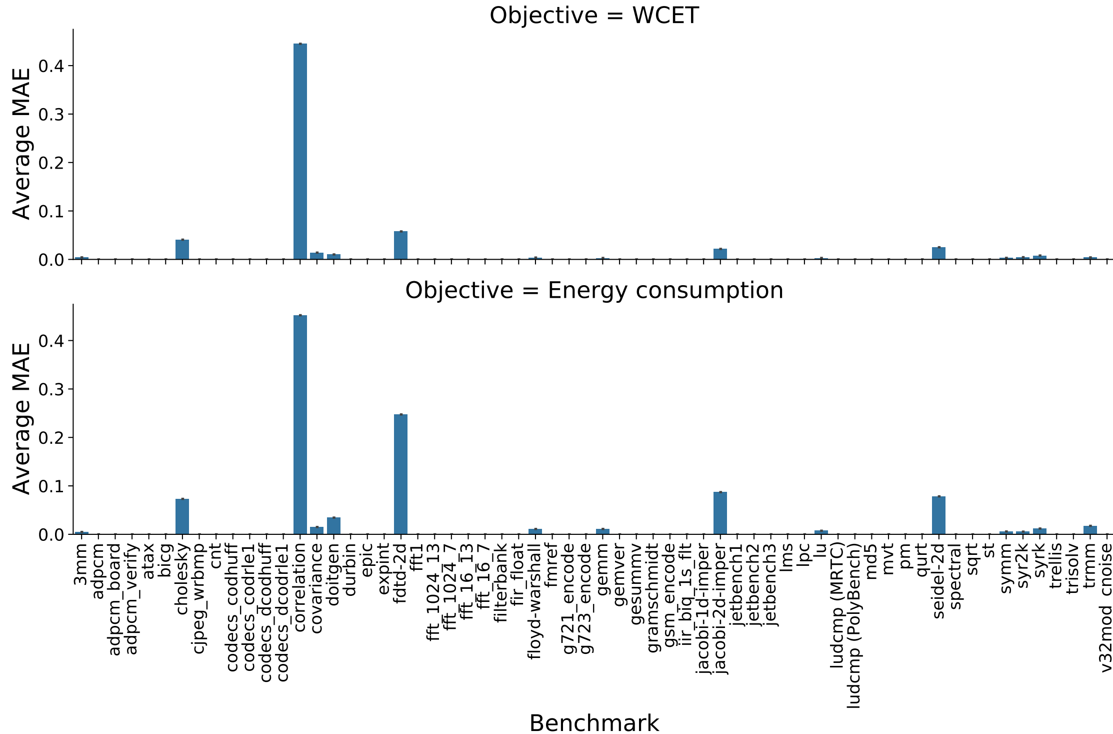


Figure 8.10: MAE for WCET and energy consumption after reducing the benchmarks' search spaces as described in Section 8.2.2 10 times. The best MAE value is 0.

is 0. We observe that the average MAE for both objectives is less than 0.5 for all benchmarks. The average MAE over all benchmarks is 0.01 and 0.02 for WCET and energy consumption, respectively. For the benchmark `correlation`, the figure shows the largest observed MAE, which is equal to 0.44 and 0.45 for WCET and energy consumption, respectively. For 47 out of 62 benchmarks, the average MAE is 0 for both objectives. There are no benchmarks such that the average MAE for one objective is 0 and for the other is greater than 0.

In the next section, we demonstrate the efficiency of the search space reduction technique by supplying an evolutionary algorithm with a reduced search space in order to provide it with a smaller search space to be explored.

8.3.4 Evolutionary Algorithm and Search Space Reduction

In the previous section, we showed that Algorithm 5 significantly reduces the search space of the function inlining problem for many benchmarks. In this section, we present the results of solving the multiobjective function inlining problem with the evolutionary algorithm MBPOA (see Section 6.2) that we supply

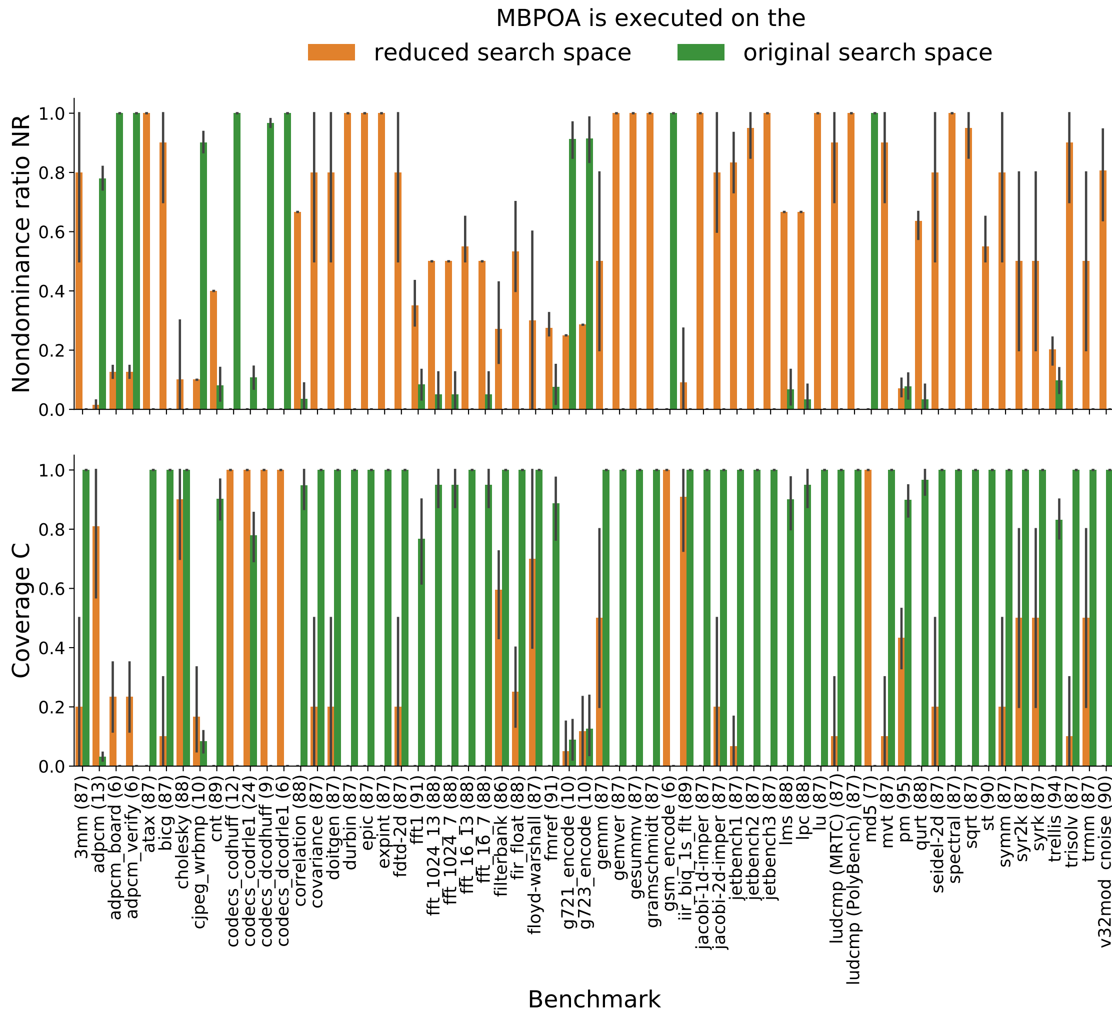


Figure 8.11: Quality indicators of the evolutionary algorithm MBPOA executed on original and reduced search spaces. The dimensions of the original search spaces are specified in parentheses next to the benchmarks' names. The experiment was repeated 10 times with different randomly generated initial populations. Nondominance ratio is to be maximized and coverage is to be minimized.

with a reduced search space. We follow the approach from Section 8.2.2 to reduce search spaces concerning WCET and energy consumption and pass the reduced search spaces to the evolutionary algorithm. The solution process of the algorithm should be faster with a reduced search space since the algorithm explores a smaller search space.

Similar to Section 7.4.3 and as Section 6.4.2 suggests, we set the MBPOA crossover parameter to 0.8. To speed up the algorithm, we use an archive that

stores explored search vectors and the corresponding objectives. We preserve the population size and the number of generations used in Chapters 6 and 7 for evaluation – the population size equal to 50 and the number of generations equal to 30 – for two main reasons:

- the preserved values ensure fair comparison after running MBPOA on the original and reduced search spaces of a benchmark;
- a smaller search space requires a smaller population and fewer generations, so the preserved values should be large enough to find a better approximation of Pareto fronts on the reduced search space than on the original one.

We repeat each experiment 10 times due to the randomness of training sets in Algorithm 5 and the randomness integrated into the evolutionary algorithm.

Figure 8.11 depicts the quality indicators of the approximated Pareto fronts computed for original problems (green) and the problem with a reduced search space (orange). The figure shows two quality indicators[3] to estimate the quality of approximated Pareto fronts: nondominance ratio NR defined in Equation (4.8) in the top plot and coverage C defined in Equation (4.9) in the bottom plot. Nondominance ratio is to be maximized and coverage is to be minimized. The bars represent the average values of the quality indicators and the error bars show the 95 % confidence interval. The x-axis lists the benchmarks with the dimensions of the search spaces specified in parentheses next to the benchmarks' names. The y-axis shows the values of the quality indicators.

Due to randomness present in evolutionary algorithms and search space reduction, we compare the average values of the quality indicators by assuming comparison tolerance equal to 0.05, i.e. two values are equal if their absolute difference is less than 0.05. We notice the following statistics:

- for 42 out of 62 benchmarks like, e.g. `durbin`, `epic`, `expint`, etc., where $NR = 0$ and $C = 1$ for MBPOA with the original search spaces, MBPOA on the original search spaces found no nondominated points;
- for 6 benchmarks, `codecs_codhuff`, `codecs_codrle1`, `codecs_dcodhuff`, `codecs_dcodrle1`, `gsm_encode`, and `md5`, where $NR = 0$ and $C = 1$ for MBPOA with reduced search spaces, MBPOA on the reduced search spaces found no nondominated points. All these benchmarks have the original search spaces with the dimension less than or equal to 24, which are quite small search spaces compared to the ones of the other benchmarks;
- for the remaining 14 benchmarks, both approaches found at least one nondominated point:

[3] The quality indicators were computed as described on Page 84.

– for 4 benchmarks, adpcm, adpcm_board, adpcm_verify, cjpeg_wrbmp, MBPOA with the original search spaces outperformed MBPOA with reduced search spaces. These benchmarks have the original search spaces with the dimension less than or equal to 13, which are also quite small search spaces compared to the most of the other benchmarks;
– for 10 benchmarks like, e.g. 3mm, bicg, cholesky, etc., MBPOA with reduced search spaces outperformed MBPOA with the original search spaces.

To conclude, MBPOA with reduced search spaces showed a higher quality of Pareto fronts than MBPOA with original search spaces for 52 out of 62 benchmarks. For the remaining 10 benchmarks, MBPOA showed better results with the original search spaces than with reduced ones. These 10 benchmarks have small search spaces compared to the other benchmarks. In our experiments, their search space dimensions are less than or equal to 24, whereas the dimensions of most of the remaining benchmarks are greater than 85.

In the previous Chapter 7, we evaluated MBPOA based on predictions, where WCET and energy consumption are predicted and code size is computed exactly. When comparing logistic regression, decision tree classifier, and AdaBoost classifier, we observed that for most of the benchmarks, decision tree classifier showed the same or better results than the other two classifiers. Thus, in this section, we compare MBPOA based on decision trees with the original search space and MBPOA executed on a reduced search space without predictions.

Figure 8.12 shows the quality indicators of Pareto fronts returned by MBPOA with reduced search spaces (orange) and by MBPOA based on decision trees with the original search spaces (blue). The figure shows two quality indicators: nondominance ratio NR defined in Equation (4.8) in the top plot and coverage C defined in Equation (4.9) in the bottom plot. The bars represent the average values and the error bars show the 95 % confidence interval. The x-axis presents the benchmarks with the corresponding search space dimensions specified in parentheses next to the benchmarks' names, whereas the y-axis shows the values of the quality indicators.

Recall that for seven benchmarks[4], all predictors failed to build prediction models. For the remaining 55 benchmarks, we observe the following results:

- for 44 benchmarks like, e.g. durbin, epic, expint, etc., where $NR = 0$ and $C = 1$ for MBPOA with decision tree, MBPOA based on predictions found no nondominated points;
- for 6 benchmarks, cjpeg_wrbmp, cnt, correlation, fft_1024_7, lpc, and trellis, MBPOA with reduced search spaces outperformed MBPOA based on predictions;

[4]Benchmarks: adpcm, codecs_codrle1, codecs_dcodhuff, g721_encode, g723_encode, md5, pm.

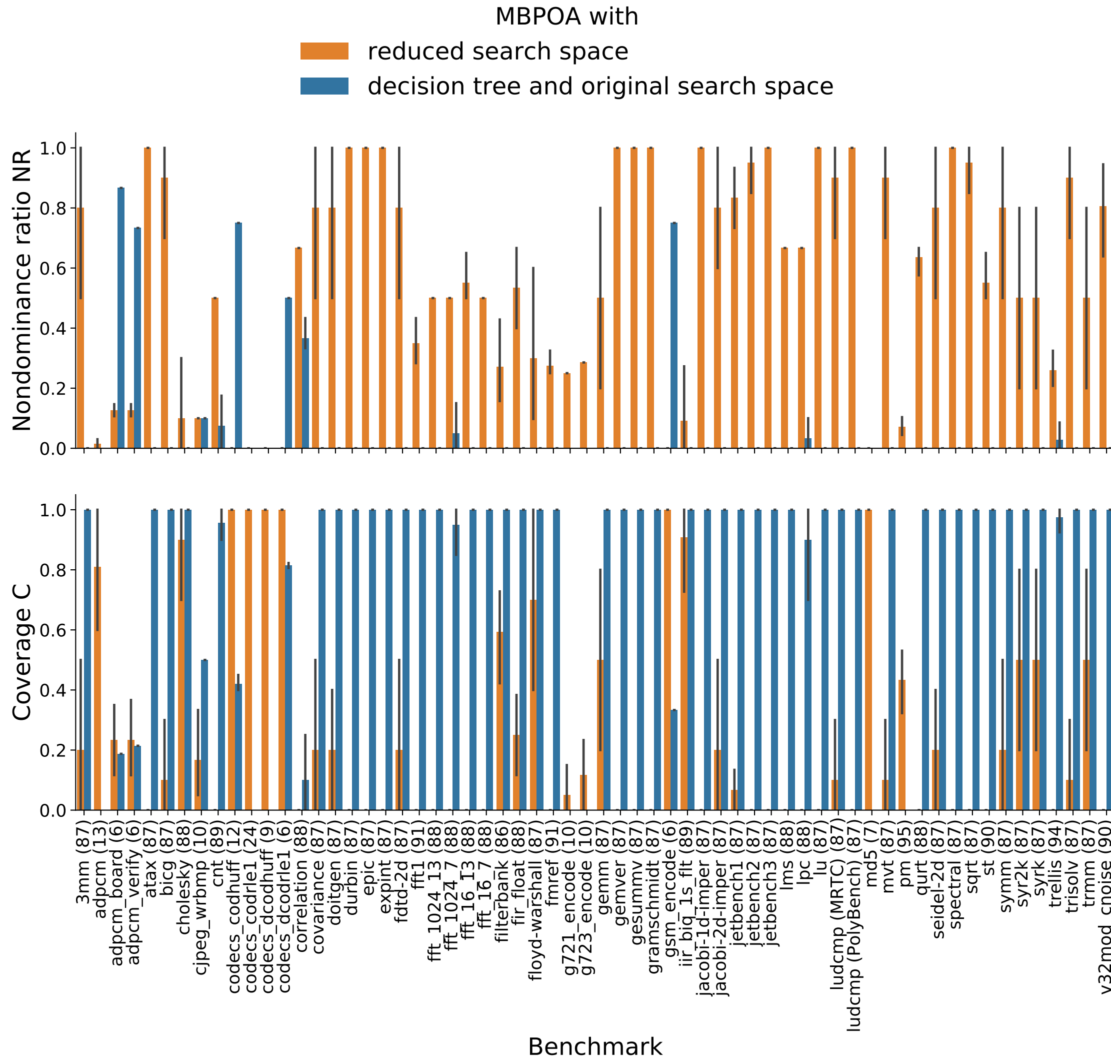


Figure 8.12: Quality indicators of the evolutionary algorithm MBPOA with reduced search spaces and MBPOA based on predictions. The dimensions of the search spaces are specified in parentheses next to the benchmarks' names. The experiment was repeated 10 times with different randomly generated initial populations. Nondominance ratio is to be maximized and coverage is to be minimized.

- for 5 benchmarks, `adpcm_board`, `adpcm_verify`, `codecs_codhuff`, `codecs_dcodrle1`, and `gsm_encode`, MBPOA with predictions outperformed MBPOA with reduced search spaces.

When comparing these two approaches, the results suggest that MBPOA with reduced search spaces significantly outperforms MBPOA based on predictions.

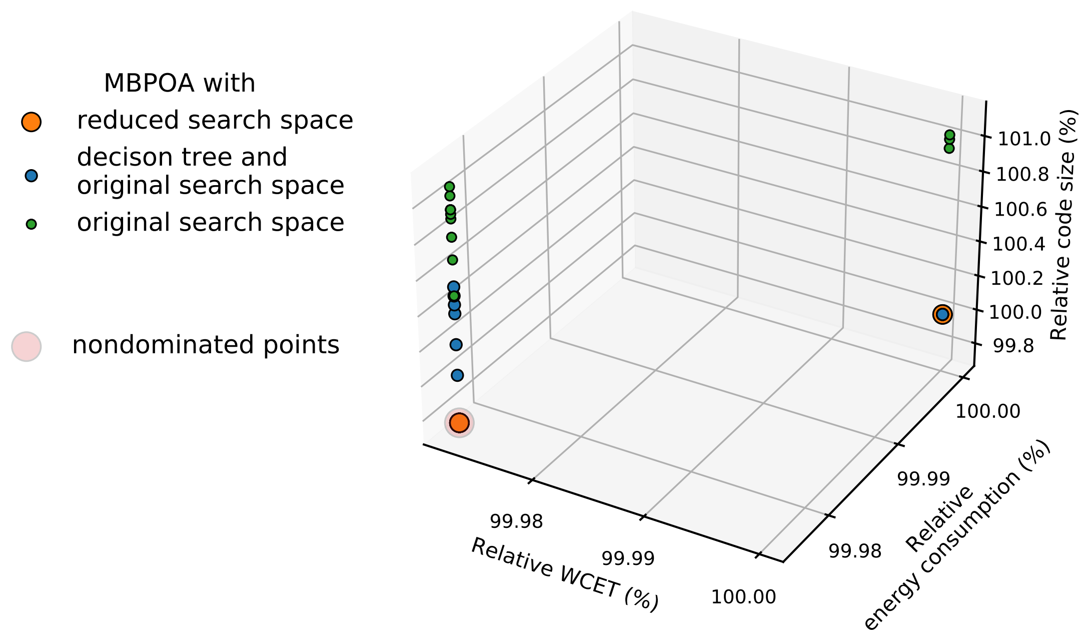


Figure 8.13: Solutions returned by MBPOA when solving the function inlining problem on the original and reduced search spaces for the benchmark `3mm` in 10 runs. 100% corresponds to the original WCET, energy consumption, and code size.

Figures 8.13 and 8.14 present examples of approximated Pareto fronts for two benchmarks: `3mm` and `cjpeg_wrbmp`. The figures present results for MBPOA with reduced search spaces (orange), MBPOA with decision tree classifier and the original search spaces (blue), and MBPOA with the original search spaces (green). In the figures, nondominated points are shown in red. The x-axis presents WCET, the y-axis – energy consumption, and the z-axis – code size.

For the benchmark `3mm`, MBPOA executed on the reduced search space found one solution (the left bottom corner) which dominates all other solutions. The algorithm returned the nondominated solution in 8 out of 10 experiments. For this benchmark, MBPOA with the reduced search space significantly outperforms MBPOA with the original search space and MBPOA with predictions.

For the benchmark `cjpeg_wrbmp`, MBPOA with the reduced search space returned four solutions and only one of them is nondominated (the right bottom corner). This nondominated solution was found in all 10 experiments by MBPOA with the reduced search space and by MBPOA with predictions. Similar to MBPOA with decision tree, MBPOA with the reduced search space also failed to find the nondominated point (the left upper corner) that was found by MBPOA based on logistic regression in the previous Chapter 7 (see Figure 7.16).

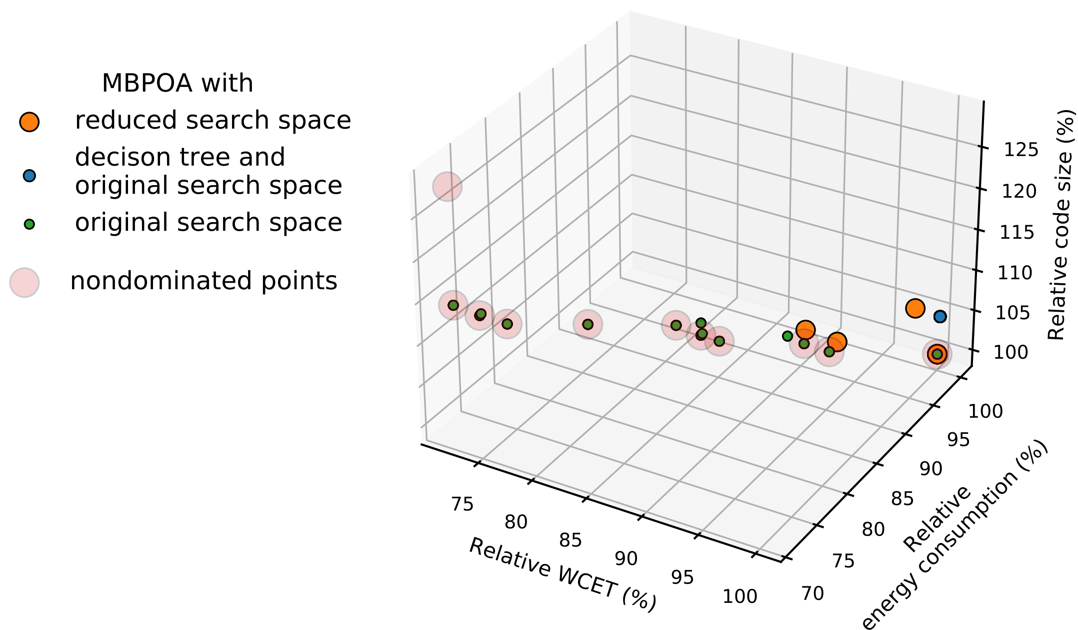


Figure 8.14: Solutions returned by MBPOA when solving the function inlining problem on the reduced and original search spaces for the benchmark `cjpeg_wrbmp` in 10 runs. 100% corresponds to the original WCET, energy consumption, and code size.

Next, we compare the runtime of the considered MBPOA configurations. Figure 8.15 presents MBPOA's average runtimes when solving the function inlining problem on reduced search spaces (orange), on the original search spaces with predictions (blue) and without predictions (green). Each experiment was repeated 10 times. The x-axis shows the tested benchmarks and the y-axis presents the average runtime.

If MBPOA solves the problem on a reduced search space, the runtime consists of the following parts:

- runtime to reduce the search space as described in Section 8.2.2;
- runtime to execute MBPOA.

If MBPOA solves the problem on the original search space and uses decision tree predictions during its execution, the runtime is computed as defined on Page 128. If MBPOA solves the problem on the original search space, the runtime represents MBPOA runtime.

The figure shows that MBPOA with the reduced search spaces is much faster than MBPOA with the original search spaces without predictions. We also

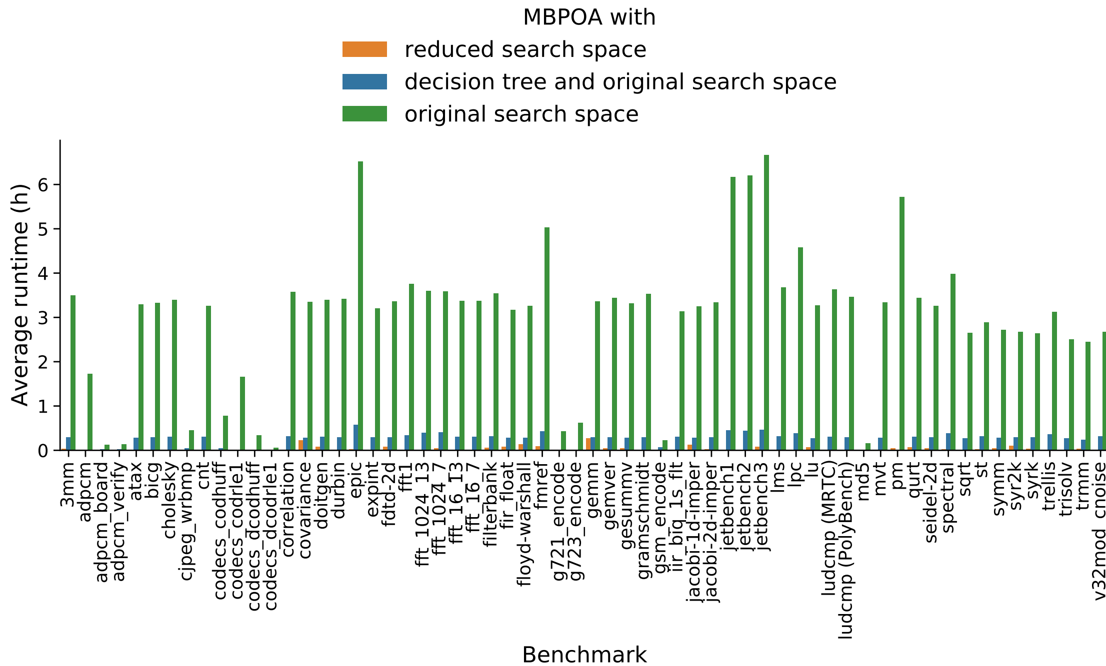


Figure 8.15: Runtimes of the evolutionary algorithm MBPOA executed on reduced and original search spaces with and without predictions made by decision tree. The experiment was repeated 10 times.

observe that the search space reduction technique speeds up MBPOA more than predictions. MBPOA based on reduced search spaces is 3 h on average faster than MBPOA with the original search spaces without predictions, and it is 15 min on average faster than MBPOA with predictions. We observe the largest saving for the benchmark `jetbench3`:

- the average runtime of MBPOA with the original search space without predictions is 6.7 h;
- the average runtime of MBPOA with the original search space and decision tree predictions is 28 min;
- the average runtime of MBPOA with the reduced search space is 5 min.

The smallest saving we observe for the benchmark `codecs_dcodrle1`:

- the average runtime of MBPOA with the original search space without predictions is 3.3 min;
- the average runtime of MBPOA with the original search space and decision tree predictions is 17 s;

- the average runtime of MBPOA with the reduced search space is 3 s.

In this section, we showed that a reduced search space speeds up an evolutionary algorithm and improves the quality of its solutions. The search space reduction technique leads to a greater saving in algorithm's runtime and a higher quality of Pareto fronts than predictions presented in the previous Chapter 7. In the next chapter, we combine the reduction technique with predictions to check whether this can speed up further the solution process of the evolutionary algorithm.

8.4 Conclusion

In this chapter, we presented a method to reduce the search spaces of compiler-based optimizations. Our method identifies those features of a search space that change the values of objectives and removes redundant features from the search space. We compared five strategies to identify the features that influence the objective; they differ in the portion of the features marked as important. We demonstrated our approach on function inlining by reducing its search space concerning WCET and energy consumption. We utilized an evolutionary algorithm to solve the multiobjective function inlining problem with WCET, code size, and energy consumption as objectives and compared Pareto fronts produced by the algorithm that we executed on the reduced search space and on the original search space with and without predictions described in the previous Chapter 7.

For evaluated benchmarks, we reduced the search space by 93 % on average. The proposed technique reduced the search space for all considered benchmarks. The algorithm executed on the original search spaces (with and without predictions) was slower than the one executed on reduced search spaces for all benchmarks. The search space reduction sped up the solution process of the evolutionary algorithm by

- 3 h on average compared to the algorithm executed on the original search space without predictions;
- 15 min on average compared to the algorithm executed on the original search space with predictions.

For most of the considered benchmarks, the evolutionary algorithm returned approximated Pareto fronts of higher quality, when it was executed on the reduced search spaces than on the original ones. Similar to the previous Chapter 7, we observed that if an original search space is not large enough (in our evaluations, these were search spaces with dimensions less than 25), then the algorithm executed on the original search space (without predictions) returns more nondominated solutions than the one executed on a reduced search space.

In the next chapter, we combine the reduction technique with predictions from Chapter 7 in order to substitute time-consuming estimations of the objectives with faster predictions after reducing a search space. We evaluate the approach in terms of runtime and capability of producing high-quality solutions for multiobjective compiler-based optimizations.

9 Search Space Reduction and Prediction Model

In Chapter 6, we illustrated that evolutionary algorithms drastically increase compile times when solving a multiobjective optimization problem with WCET, energy consumption, and code size as objectives. We attribute this to extensive evaluations of objectives carried out by evolutionary algorithms and time-consuming WCET and energy consumption analyses. In previous chapters, we present two methods to tackle this issue:

- Chapter 7 describes a prediction model that substitutes expensive evaluations of the objectives with cheaper predictions;
- Chapter 8 presents a technique that reduces the search space of a problem and supplies an evolutionary algorithm with a smaller search space to be explored.

In this chapter, we combine the reduction technique with the prediction model in order to solve a multiobjective problem. We evaluate the approach in terms of evolutionary algorithm runtime and the quality of Pareto fronts.

The results of this chapter were presented at the workshop on Software and Compilers for Embedded Systems (SCOPES) in 2021 [MF21a].

This chapter is structured as follows: Section 9.1 describes the approach that combines the reduction technique and the prediction model, Section 9.2 presents evaluation results, and Section 9.3 gives a conclusion.

9.1 Method

By combining the reduction technique from Chapter 8 and the prediction model from Chapter 7, we aim to run an evolutionary algorithm on a reduced space and predict both WCET and energy consumption during its execution, so the prediction model must be fitted on the reduced search space.

Reducing a search space and building a prediction model require training and test sets with the expensively evaluated objectives; the generation of these sets is the most time-consuming part of both techniques. To minimize the number of expensive evaluations, two fundamental approaches can be pursued:

1. "Predict→Reduce":
   a) fit the prediction model on the original search space of a problem;
   b) reduce the search space by using the prediction model to generate training and test sets, i.e. the search space reduction from Chapter 8 relies on predicted WCET and energy consumption instead of the objectives estimated by aiT;
   c) refit the prediction model on the reduced search space.

   Following this approach, the WCC compiler expensively estimates the objectives to fit the prediction model on the original search space and uses cheaper predictions to reduce the search space.

   When fitting the model on the original search space, a predictor takes into account all features. If the original search space is large, the prediction model might be too complex, and predictions might be poor. If we reduce the search space, map the reduced search space to the original one, and predict the objectives on the reduced search space (mapped to the original one) by using the prediction model fitted on the original search space, we will not improve the quality of predictions. In order to improve the quality, the compiler should refit the prediction model on the reduced search space. After reducing the search space, the predictor focuses only on important features. This should reduce model complexity and improve the quality of predictions. There are two main drawbacks of this approach:

   - to refit the prediction model, the compiler requires expensive estimations of the objectives on the reduced search space;
   - if the predictor fails to fit the prediction model on the original search space, either the entire method fails or the search space has to be reduced by using expensive estimations of the objectives, which is equivalent to the next approach "Reduce→Predict".

2. "Reduce→Predict":
   a) reduce the search space;
   b) fit the prediction model on the reduced search space.

   The compiler expensively evaluates the objectives when reducing the search space and fitting the prediction model, but, in contrast to the first approach, the compiler fits the prediction model only once.

Both approaches have common characteristics:

- they require initial training and test sets with expensively evaluated objectives in their first steps: the first approach to fit the prediction model in Step 1a and the second one to reduce the search space in Step 2a;

- they fit the prediction model on the reduced search space by utilizing expensive evaluations of the objectives: the first approach refits the model in Step 1c, the second approach fits the model in Step 2b.

We prefer the second approach over the first one since in Section 7.4.2, there are seven benchmarks for which all considered predictors failed to fit their prediction models on the original search spaces, i.e. Step 1a of the first approach fails. In Section 8.3.3, the search space was reduced for all benchmarks, i.e. the second approach might work for all benchmarks including these seven benchmarks.

Moreover, the second approach is faster than the first one for the following reasons:

- the second approach fits the prediction model once, whereas the first one fits it twice;

- if the original search space is large, the reduction technique usually requires fewer training samples than the prediction model as we can see in Figures 8.6 and 8.7 and Table E.1, e.g. for the benchmarks `fft1`, `fmref`, `v32mod_cnoise`.

In general, to solve a multiobjective problem on a reduced search space by utilizing a prediction model, we

1. reduce the original search space of a problem regarding WCET and energy consumption as described in Section 8.2.2;

2. fit prediction models on the reduced search space for WCET and energy consumption by using Algorithm 3;

3. run an evolutionary algorithm on the reduced search space with the objectives evaluated by the prediction model during algorithm execution;

4. estimate the WCET and energy consumption of the final solutions by static analysers.

Since Step 3 of our approach predicts WCET and energy consumption while executing the evolutionary algorithm and since these predictions might differ from safe estimations, static analyser estimate the objectives of the final solutions in Step 4.

In the next section, we evaluate the proposed approach for the multiobjective function inlining problem in terms of evolutionary algorithm runtime and the quality of Pareto fronts.

9.2 Evaluation

The previous section describes an approach that combines predictions from Chapter 7 and the search space reduction from Chapter 8 to speed up an evolutionary algorithm while solving a multiobjective problem at compile time. In this section, we evaluate the approach when solving the multiobjective function inlining problem from Section 6.3.

The section is organized as follows: Section 9.2.1 describes experimental setups and Section 9.2.2 presents evaluation results.

9.2.1 Experimental Setup

Similar to the previous chapters, we use WCC described in Chapter 3 to compile programs for an ARM Cortex-M0 processor architecture with optimization level O2. We estimate WCET and energy consumption by using the static analysers aiT [Abs22] and EnergyAnalyser [Tea] version 20.10i. Table 5.1 gives the characteristics of a server used to run evaluations. We utilize benchmarks from the benchmark suites PolyBench, MediaBench, MRTC, DSPstone, and UTDSP with the dimensions of search spaces greater than 5 similar to Chapters 6–8. Appendix B lists all evaluated benchmarks.

To evaluate the approach presented in the previous section, we solve the function inlining problem by using the evolutionary algorithm MBPOA. We set the MBPOA crossover parameter to 0.8 as Section 6.4.2 suggests and preserve the population size equal to 50 individuals and the number of generations equal to 30 as it is done in the previous chapters to guarantee fair comparison between all approaches in the next section.

To reduce a search space, we follow steps described in Section 8.2.2 and use input parameters for Algorithm 5 presented on Page 154.

To build a prediction model, we follow steps from Section 7.3 and use input parameters for Algorithm 3 listed on Page 126. In this chapter, we consider logistic regression, decision tree, and AdaBoost classifiers which showed the best prediction accuracy in Chapter 7. In the next section, we evaluate the classifiers in terms of prediction quality on reduced search spaces.

9.2.2 Prediction Models on Reduced Search Spaces

To build a prediction model on a reduced search space, Algorithm 3 requires a classifier to fit the model. To choose among logistic regression, decision tree, and AdaBoost classifiers, we compare the quality of solutions produced by MBPOA when solving the function inlining problem with reduced search spaces and using one of the considered classifiers to predict WCET and energy consumption as described in Section 9.1. We repeat each experiment 10 times.

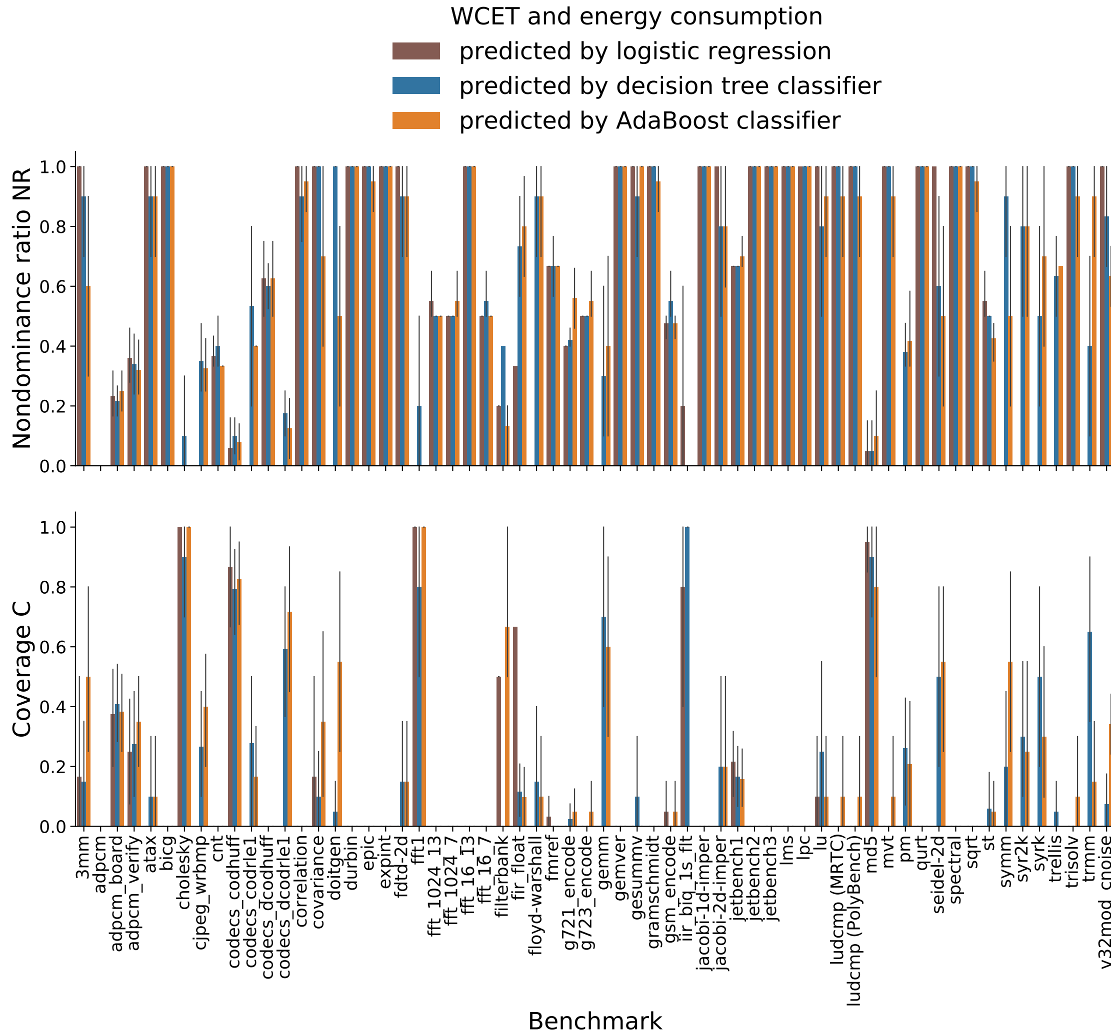


Figure 9.1: Quality indicators of Pareto fronts returned by MBPOA on reduced search spaces and with WCET and energy consumption predicted on the reduced search spaces by logistic regression, decision tree, and AdaBoost classifiers. The dimensions of the search spaces are specified in parentheses next to the benchmarks' names. Nondominance ratio is to be maximized and coverage is to be minimized. Each experiment was repeated 10 times.

Figure 9.1 presents the quality indicators of Pareto fronts returned by MBPOA executed on reduced search spaces and with WCET and energy consumption predicted on the reduced search spaces by three classifiers: logistic regression (brown), decision tree classifier (blue) and AdaBoost classifier (orange). The bars represent the average values and the error bars show the 95 % confidence interval.

The x-axis presents the benchmarks and the y-axis shows two considered quality indicators: nondominance ratio NR (the top plot) defined in Equation (4.8) and coverage C (the bottom plot) defined in Equation (4.9). Nondominance ratio is to be maximized and coverage is to be minimized.

In Section 7.4.2, all tested predictors failed to fit prediction models on the original search spaces for 7 out of 62 benchmarks. The three classifiers considered in this chapter failed to fit their prediction models on the reduced search space for one benchmark `adpcm`. AdaBoost classifier also failed to build the model for the benchmark `iir_biq_1s_flt`, whereas logistic regression failed for 12 benchmarks – additional to `adpcm` – like e.g. `cjpeg_wrbmp`, `codecs_codrle1`, `codecs_dcodrle1`, etc.

Since logistic regression failed for many benchmarks compared to decision tree and AdaBoost classifiers, we compare further only the two more successful classifiers. Due to randomness present in evolutionary algorithms, search space reduction, and predictors, when comparing the average values of the quality indicators, we assume comparison tolerance equal to 0.05, i.e. two values are equal if their absolute difference is less than 0.05. Both classifiers led to the quality indicators of the same quality for 32 out of 61 benchmarks (excluding the failed benchmark `adpcm`), e.g. like `adpcm_board`, `atax`, `bicg`, etc. For the benchmark `codecs_codrle1`, we cannot state which classifier is better, since according to nondominance ratio, decision tree outperformed AdaBoost, and according to coverage, vice versa. For the remaining 28 benchmarks,

- decision tree classifier outperformed AdaBoost classifier in the case of 19 benchmarks like, e.g. `3mm`, `adpcm_verify`, `cholesky`, etc.;
- AdaBoost classifier outperformed decision tree classifier in the case of 9 benchmarks like, e.g. `fir_float`, `g721_encode`, `gemm`, etc.

Similar to Section 7.4.3, decision tree classifier led to the same or a higher quality of Pareto fronts for more benchmarks than AdaBoost classifier, so we evaluate further predictions on reduced search spaces by considering only decision tree classifier.

In order to justify whether predictions on reduced search spaces degrade the quality of Pareto fronts, Figure 9.2 presents the quality indicators when considering four configurations of MBPOA listed in Table 9.1: $\mathrm{MBPOA}_{\mathrm{red+pred}}$ (purple), $\mathrm{MBPOA}_{\mathrm{red}}$ (orange), $\mathrm{MBPOA}_{\mathrm{pred}}$ (blue), and $\mathrm{MBPOA}_{\mathrm{orig}}$ (green). The x-axis shows the evaluated benchmarks, and the y-axis presents nondominance ratio (the top plot) and coverage (the bottom plot). The bars represent the average values and the error bars show the 95 % confidence interval.

We observed the following results for 62 tested benchmarks:

- $\mathrm{MBPOA}_{\mathrm{red+pred}}$ outperformed $\mathrm{MBPOA}_{\mathrm{orig}}$ for 47 benchmarks like, e.g. `3mm`, `atax`, or `bicg`. For the remaining 15 benchmarks, we notice that

Table 9.1: MBPOA configurations.

Name	Reduced search space	Predicted objectives*
$\mathsf{MBPOA}_{\mathtt{red+pred}}$	✓	✓
$\mathsf{MBPOA}_{\mathtt{red}}$	✓	–
$\mathsf{MBPOA}_{\mathtt{pred}}$	–	✓
$\mathsf{MBPOA}_{\mathtt{orig}}$	–	–

*The objectives – WCET and energy consumption – were predicted by decision tree classifier and estimated by the static analysers aiT and Energy-Analyser, otherwise.

- for 2 benchmarks `filterbank` and `iir_biq_1s_flt`, both methods, $\mathsf{MBPOA}_{\mathtt{red+pred}}$ and $\mathsf{MBPOA}_{\mathtt{orig}}$ failed to find any nondominated point in 10 experiments ($\mathsf{NR} = 0, \mathsf{C} = 1$);
- for 13 benchmarks like, e.g. `adpcm`, `adpcm_board`, or `adpcm_verify`, $\mathsf{MBPOA}_{\mathtt{orig}}$ outperformed not only $\mathsf{MBPOA}_{\mathtt{red+pred}}$ but also $\mathsf{MBPOA}_{\mathtt{pred}}$ and $\mathsf{MBPOA}_{\mathtt{red}}$. These results are in line with the previous Chapters 7 and 8. These 13 benchmarks have small original search spaces compared to the other benchmarks. Their dimensions are less than 25, whereas all other benchmarks have the original search spaces with dimensions greater than 85;

- $\mathsf{MBPOA}_{\mathtt{red+pred}}$ outperformed $\mathsf{MBPOA}_{\mathtt{pred}}$ for 54 benchmarks like, e.g. `3mm`, `atax`, or `bicg`. For the remaining 8 benchmarks, we observe that
  - for the benchmark `adpcm`, both approaches failed to fit prediction models;
  - for 2 benchmarks `filterbank` and `iir_biq_1s_flt`, both methods failed to find any nondominated point in 10 experiments ($\mathsf{NR} = 0, \mathsf{C} = 1$);
  - for the remaining 5 benchmarks, `adpcm_board`, `adpcm_verify`, `codecs_codhuff`, `codecs_dcodrle1`, and `gsm_encode`, $\mathsf{MBPOA}_{\mathtt{pred}}$ outperformed $\mathsf{MBPOA}_{\mathtt{red+pred}}$. The search spaces of these benchmarks are small with dimensions less than or equal to 12;
- $\mathsf{MBPOA}_{\mathtt{red+pred}}$ outperformed $\mathsf{MBPOA}_{\mathtt{red}}$ for 17 benchmarks like, e.g. `3mm`, `bicg`, or `cnt`. For the remaining 45 benchmarks, we observe that
  - for 30 benchmarks like, e.g. `cholesky`, `durbin`, or `epic`, both methods led to the same quality of Pareto fronts;
  - for 15 benchmarks like, e.g. `adpcm_board`, `adpcm_verify`, or `atax`, predictions on reduced search spaces degraded the quality of Pareto

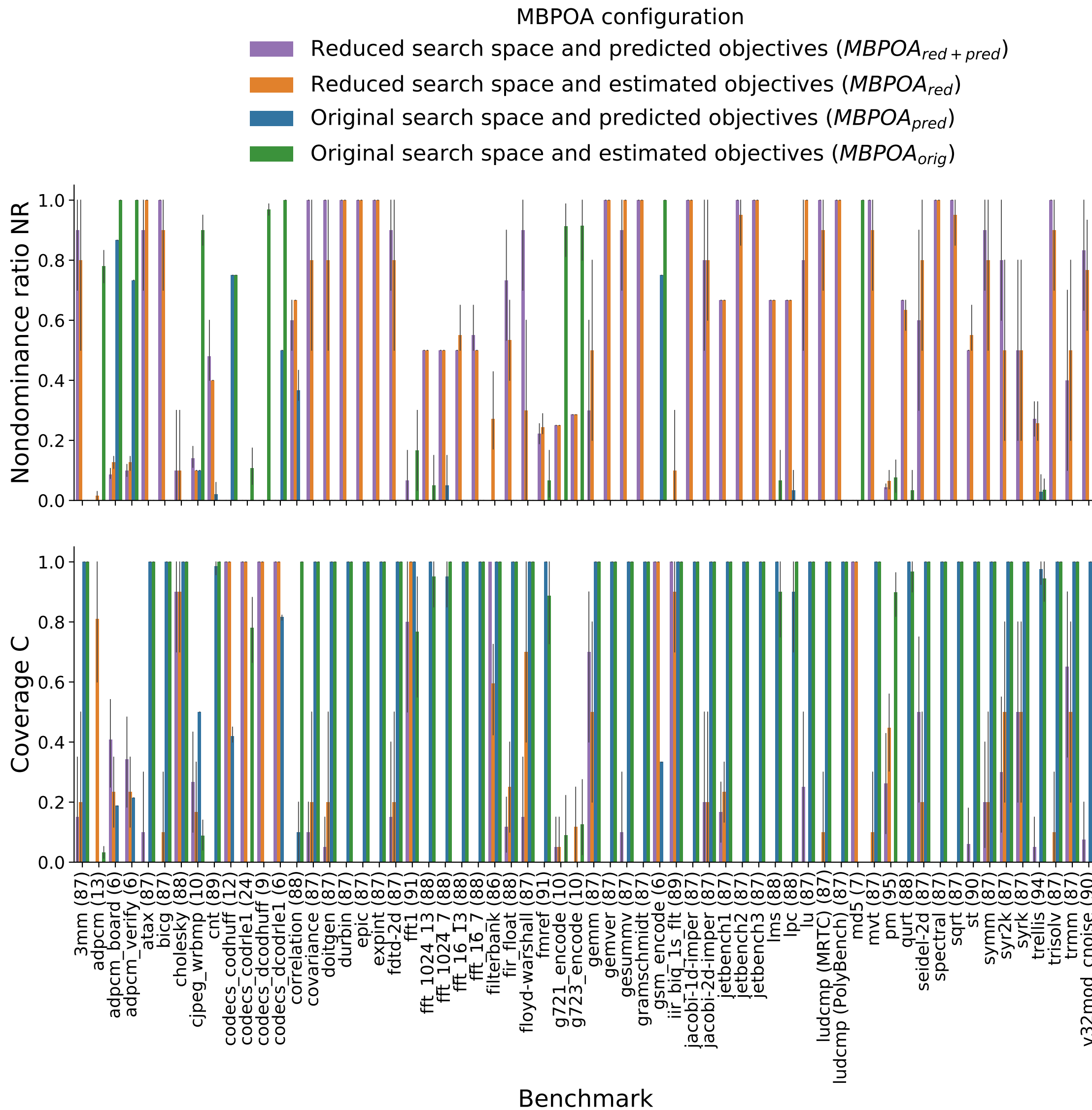


Figure 9.2: Quality indicators of Pareto fronts returned by MBPOA with configurations from Table 9.1. The dimensions of the search spaces are specified in parentheses next to the benchmarks' names. Nondominance ratio is to be maximized and coverage is to be minimized. Each experiment was repeated 10 times.

fronts. For theses benchmarks, there is no dependency between the quality indicators' results and dimensions of the original or reduced search spaces. E.g. the dimension of the search space is 6 for `adpcm_board` and 87 for `atax`. Figure 8.9 does not show any pattern for the dimensions of the reduced search spaces for these 15 benchmarks.

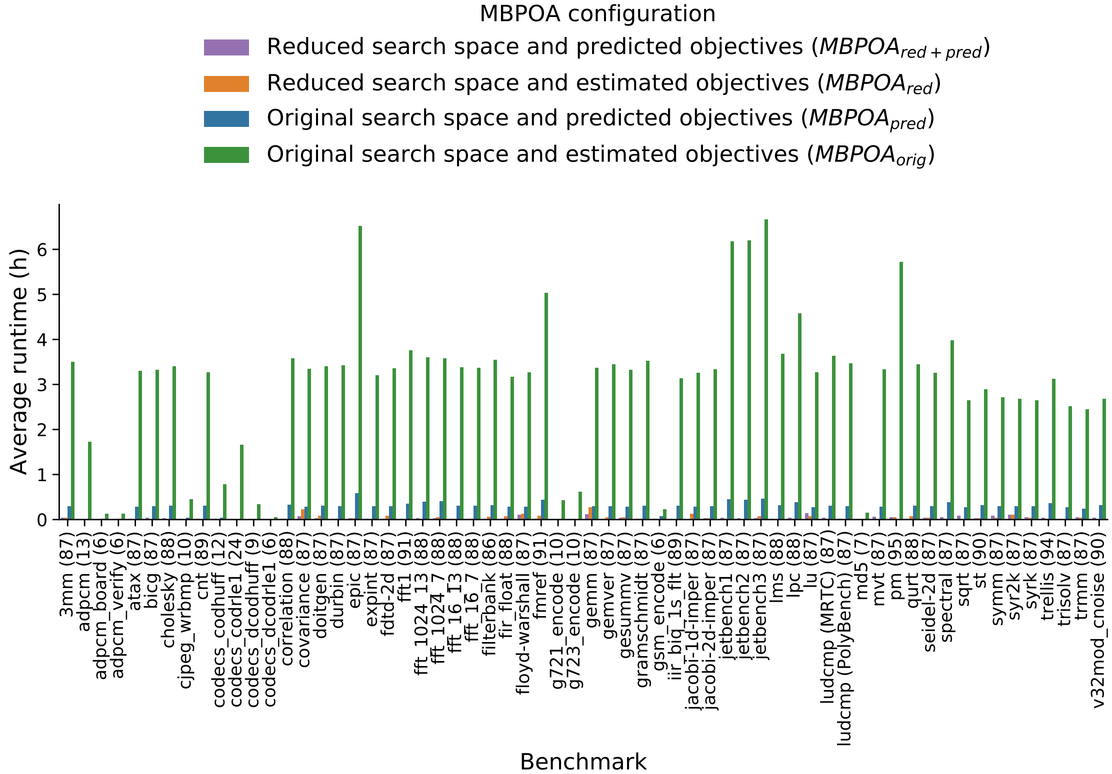


Figure 9.3: Runtimes of the evolutionary algorithm MBPOA with configurations from Table 9.1. The objectives were predicted by decision tree classifier and estimated by aiT and EnergyAnalyser. The dimensions of the search spaces are specified in parentheses next to the benchmarks' names. Each experiment was repeated 10 times.

To sum up, similar to the previous Chapters 7 and 8, $\text{MBPOA}_{\text{red+pred}}$ works for the case when the search space of a problem is large enough, in our evaluations, it works for benchmarks with search space dimensions greater than 85. Moreover, predictions on reduced search spaces ($\text{MBPOA}_{\text{red+pred}}$) improved the quality of Pareto fronts returned by MBPOA executed on the reduced search spaces without predictions ($\text{MBPOA}_{\text{red}}$) for 1/4 of evaluated benchmarks.

Figure 9.3 shows the average runtime of MBPOA over 10 experiments for the approaches listed in Table 9.1: $\text{MBPOA}_{\text{red+pred}}$ (purple), $\text{MBPOA}_{\text{red}}$ (orange), $\text{MBPOA}_{\text{pred}}$ (blue), and $\text{MBPOA}_{\text{orig}}$ (green). The x-axis presents the benchmarks with the dimensions of the search spaces in parentheses and the y-axis shows the average runtime.

The figure shows that $\text{MBPOA}_{\text{red+pred}}$ was faster than $\text{MBPOA}_{\text{pred}}$ and $\text{MBPOA}_{\text{orig}}$ for all benchmarks. Over all benchmarks and all 10 experiments for

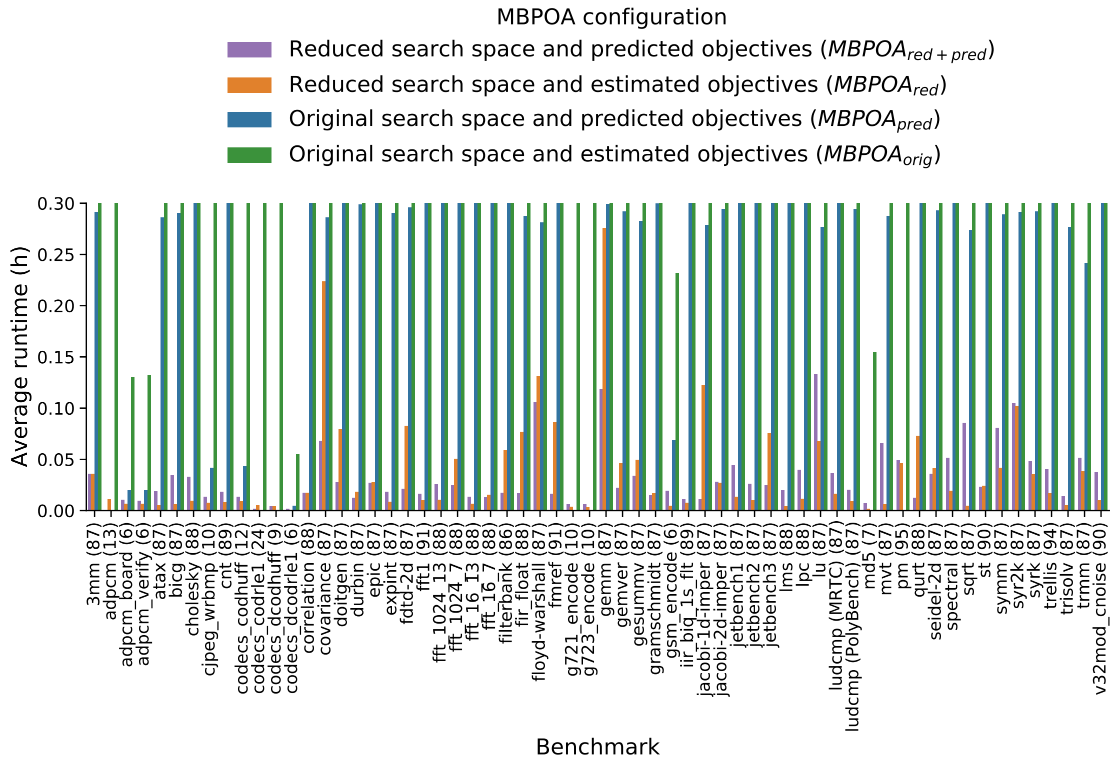


Figure 9.4: MBPOA runtime from Figure 9.3 with the range of the y-axis changed to [0, 0.3].

each benchmark, $\text{MBPOA}_{\text{red+pred}}$ was 3 h on average faster than $\text{MBPOA}_{\text{orig}}$ and 15 min on average faster than $\text{MBPOA}_{\text{pred}}$.

To compare the runtimes of $\text{MBPOA}_{\text{red+pred}}$ and $\text{MBPOA}_{\text{red}}$, Figure 9.4 presents the results from Figure 9.3 with the range of the y-axis changed from [0, 6] to [0, 0.3].

For 39 benchmarks like, e.g. `adpcm_board`, `adpcm_verify`, or `atax`, $\text{MBPOA}_{\text{red+pred}}$ was slower than $\text{MBPOA}_{\text{red}}$, i.e. predictions on reduced search spaces slowed down the evolutionary algorithm. The average runtime increase for these benchmarks is only 1 min. The figure shows the largest runtime increase of 4.9 min (from 17 s to 5.1 min) for the benchmark `sqrt`.

For the remaining 22 benchmarks[1] like e.g. `covariance`, `doitgen`, or `durbin`, $\text{MBPOA}_{\text{red+pred}}$ was faster than $\text{MBPOA}_{\text{red}}$, i.e. predictions on reduced search spaces sped up the evolutionary algorithm. For these benchmarks, the average runtime decrease is only 3 min. The figure shows the largest runtime decrease of 10 min (from 17 min to 7 min) for the benchmark `gemm`.

[1] The benchmark `adpcm` is excluded since decision tree classifier failed to fit the prediction model on the reduced search space.

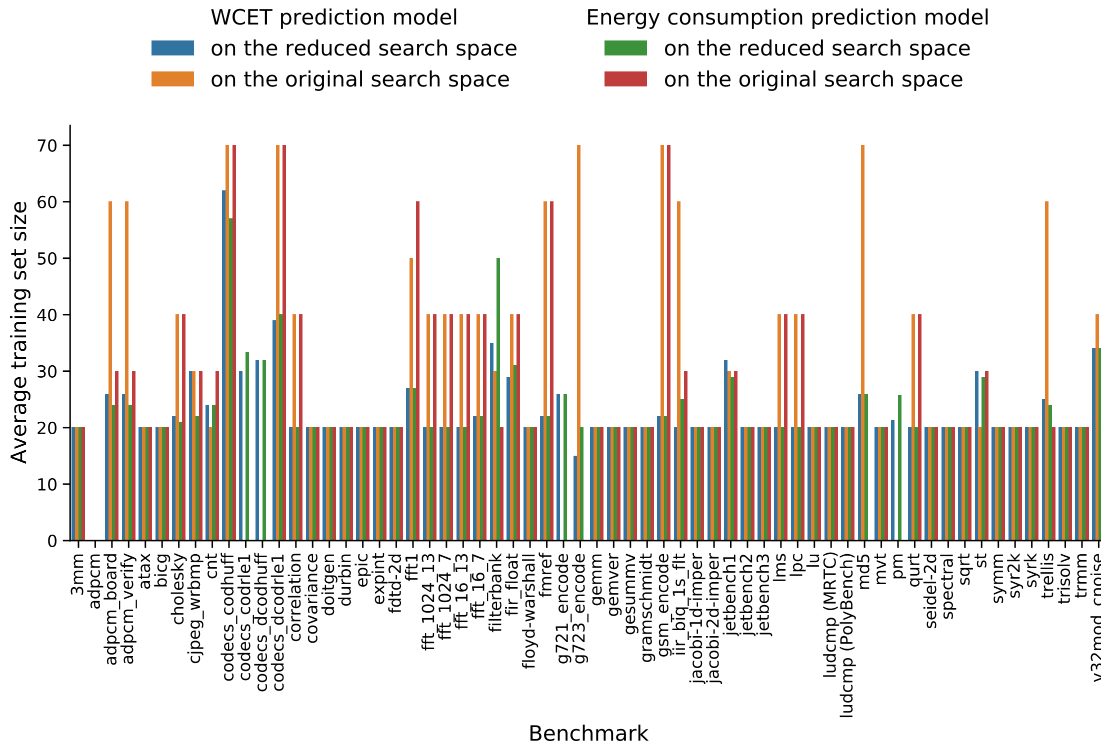


Figure 9.5: Training set sizes when fitting decision tree classifier for WCET and energy consumption on the original and reduced search spaces. Each experiment was repeated 10 times.

When building prediction models for WCET and energy consumption, we would expect that Algorithm 3 requires fewer training samples with reduced search spaces than with the original ones to achieve a high prediction rate. Figure 9.5 presents the average size of training sets when building decision tree classifier on the original and reduced search space for WCET and energy consumption. The x-axis shows the benchmarks, the y-axis presents the average size of final training sets used in Algorithm 3.

The figure shows the following results:

- for the benchmarks `adpcm`, Algorithm 3 failed to fit prediction models on the original and reduced search spaces for both objectives;
- for 6 benchmarks, `codecs_codrle1`, `codecs_dcodhuff`, `g721_encode`, `g723_encode`, `md5`, `pm`, Algorithm 3 failed to fit prediction models on the original search spaces for at least one objective, but it successfully fitted predictions models on the reduced search spaces for both objectives;

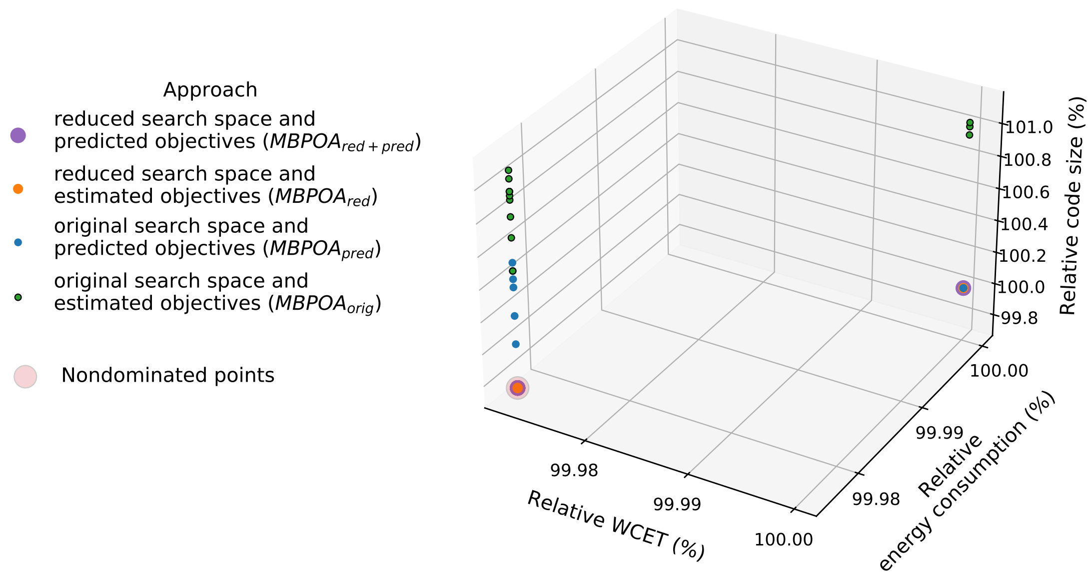


Figure 9.6: Solutions returned by MBPOA when solving the function inlining problem on the original and reduced search spaces with and without predictions for the benchmark `3mm` in 10 runs. 100% corresponds to the original WCET, energy consumption, and code size.

- for 30 benchmarks like, e.g. `3mm`, `atax`, `bicg`, the training set sizes coincide for the original and reduced search spaces and for both objectives. The training set size is equal to 20 for these benchmarks;
- for 20 benchmarks like, e.g. `adpcm_board`, `adpcm_verify`, `cholesky`, the average training set size decreased for both objectives;
- for the benchmark `filterbank`, the average training set size increased from 30 to 35 for WCET and from 20 to 50 for energy consumption;
- for 3 benchmarks, `cnt`, `jetbench1`, and `st`, the training set size increased for WCET and decreased for energy consumption;
- for the benchmark `trellis`, the training set size decreased for WCET and increased for energy consumption.

To sum up, only for 5 benchmarks, decision tree classifier required more training points on the reduced search spaces than on the original ones.

Similar to the previous chapters, Figures 9.6 and 9.7 present examples of approximated Pareto fronts found by MBPOA in 10 runs for the benchmarks

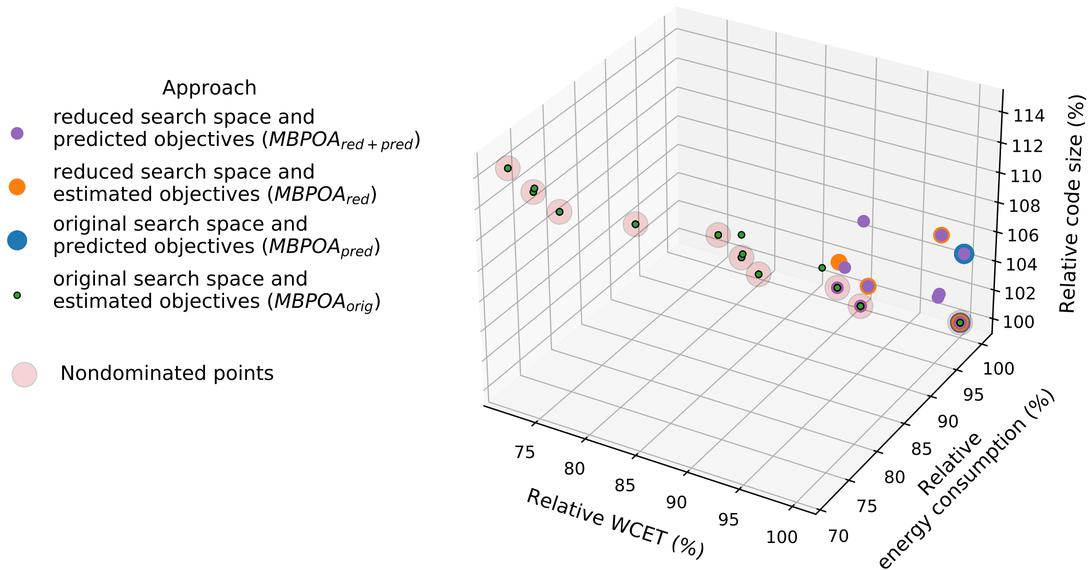


Figure 9.7: Solutions returned by MBPOA when solving the function inlining problem on the original and reduced search spaces with and without predictions for the benchmark `cjpeg_wrbmp` in 10 runs. 100% corresponds to the original WCET, energy consumption, and code size.

`3mm` and `cjpeg_wrbmp`, respectively. The figures show the solutions of four approaches: $\text{MBPOA}_{\text{red+pred}}$ (purple), $\text{MBPOA}_{\text{red}}$ (orange), $\text{MBPOA}_{\text{pred}}$ (blue), and $\text{MBPOA}_{\text{orig}}$ (green). Nondominated points are shown in red.

For the benchmark `3mm`, Figure 9.6 shows only one nondominated solution (the left bottom corner) which was found by $\text{MBPOA}_{\text{red+pred}}$ and $\text{MBPOA}_{\text{red}}$. This solution dominates all solutions found by $\text{MBPOA}_{\text{pred}}$ and $\text{MBPOA}_{\text{orig}}$.

In total, $\text{MBPOA}_{\text{red+pred}}$ returned two solutions which coincide with the solutions found by $\text{MBPOA}_{\text{red}}$: one nondominated solution and one dominated solution. The dominated solution from the right bottom corner was also found by $\text{MBPOA}_{\text{pred}}$.

In general, for this benchmark, predictions on the reduced search space slightly improve the quality of the resulting Pareto fronts, since $\text{MBPOA}_{\text{red+pred}}$ found the nondominated solution in 10 runs and $\text{MBPOA}_{\text{red}}$ found it in 8 runs.

For the benchmark `cjpeg_wrbmp`, Figure 9.7 shows that $\text{MBPOA}_{\text{orig}}$ significantly outperforms not only $\text{MBPOA}_{\text{red}}$ and $\text{MBPOA}_{\text{pred}}$ but also $\text{MBPOA}_{\text{red+pred}}$.

$\text{MBPOA}_{\text{red+pred}}$ returned 10 solutions, three of which are nondominated, whereas $\text{MBPOA}_{\text{red}}$ found 4 solutions, one of which is nondominated (the right bottom corner). This nondominated solution was also found by $\text{MBPOA}_{\text{pred}}$.

Table 9.2: Benchmark `cjpeg_wrbmp`. The number of nondominated solutions $|PF_A \cap PF|$ and the total number of solutions $|PF_A|$ returned by $\mathsf{MBPOA}_{red+pred}$ and MBPOA_{red} in 10 runs and the corresponding coverage defined in Equation (4.9).

Run	$\mathsf{MBPOA}_{red+pred}$			MBPOA_{red}		
	$\lvert PF_A \cap PF\rvert$	$\lvert PF_A\rvert$	Coverage	$\lvert PF_A \cap PF\rvert$	$\lvert PF_A\rvert$	Coverage
1	1	3	0.67	1	1	0.00
2	1	2	0.50	1	2	0.50
3	2	3	0.33	1	1	0.00
4	3	3	0.00	1	3	0.67
5	2	2	0.00	1	1	0.00
6	1	1	0.00	1	2	0.50
7	1	3	0.67	1	1	0.00
8	1	1	0.00	1	1	0.00
9	1	1	0.00	1	1	0.00
10	1	2	0.50	1	1	0.00
Average			0.27			0.17

For this benchmark, since $\mathsf{MBPOA}_{red+pred}$ found more nondominated solutions than MBPOA_{red} and MBPOA_{pred}, Figure 9.2 shows that nondominance ratio is larger for $\mathsf{MBPOA}_{red+pred}$ than for MBPOA_{red} and MBPOA_{pred}.

Figure 9.2 shows that coverage is higher for $\mathsf{MBPOA}_{red+pred}$ than for MBPOA_{red}, although $\mathsf{MBPOA}_{red+pred}$ found more nondominated solutions over 10 runs than MBPOA_{red}. This means predictions on the reduced search space degraded coverage. Table 9.2 presents the number of nondominated solutions $|PF_A \cap PF|$ and the total number of solutions $|PF_A|$ returned by $\mathsf{MBPOA}_{red+pred}$ and MBPOA_{red} in 10 runs. It also presents coverage for each run together with the average coverage over 10 runs. A larger average coverage for $\mathsf{MBPOA}_{red+pred}$ than for MBPOA_{red} is explained by the following facts:

- in 7 out 10 runs, coverage is equal to 0 in the case of MBPOA_{red}, since in these runs, the method returned a solution set consisting of only one solution which is nondominated;

- in 5 out of 10 runs, $\mathsf{MBPOA}_{red+pred}$ returned solution sets consisting of only nondominated solutions, which led to coverage equal to 0.

To sum up, for the benchmark `cjpeg_wrbmp`, predictions on the reduced search space improved nondominance ratio but degraded coverage.

9.3 Conclusion

In this chapter, we presented an approach that

1. reduces the search space of multiobjective compiler-based optimization;
2. builds a prediction model on the reduced search space to quickly predict WCET and energy consumption instead of estimating them by time-consuming static analysers;
3. supplies an evolutionary algorithm – used to solve the multiobjective problem – with a smaller search space to be explored and the prediction model.

For the exemplary multiobjective function inlining problem from Section 6.3, we used decision tree classifier to build a prediction model on a reduced search space since it outperformed logistic regression and AdaBoost classifier – these three classifiers showed the best prediction accuracy in Chapter 7.

We compared solution sets produced by four methods presented in Table 9.1. Comparing the quality of the resulting solutions sets, we observed the following results for 62 tested benchmarks:

- $\mathsf{MBPOA}_{\mathsf{red+pred}}$ outperformed $\mathsf{MBPOA}_{\mathsf{orig}}$ for 47 benchmarks;
- $\mathsf{MBPOA}_{\mathsf{red+pred}}$ outperformed $\mathsf{MBPOA}_{\mathsf{pred}}$ for 54 benchmarks;
- $\mathsf{MBPOA}_{\mathsf{red+pred}}$ outperformed $\mathsf{MBPOA}_{\mathsf{red}}$ for 17 benchmarks and for 30 benchmarks $\mathsf{MBPOA}_{\mathsf{red+pred}}$ and $\mathsf{MBPOA}_{\mathsf{red}}$ led to the same solution quality.

Similar to the previous Chapters 7 and 8, $\mathsf{MBPOA}_{\mathsf{red+pred}}$ works for the case when the original search space of a problem is large enough. In our evaluations, the method was successful for benchmarks with the dimensions of the original search spaces greater than 85.

For 39 benchmarks, $\mathsf{MBPOA}_{\mathsf{red+pred}}$ was slower than $\mathsf{MBPOA}_{\mathsf{red}}$, i.e. predictions on reduced search spaces slowed down the evolutionary algorithm but the average runtime increased only by 1 min for evaluated benchmarks. For 22 benchmarks, $\mathsf{MBPOA}_{\mathsf{red+pred}}$ was faster than $\mathsf{MBPOA}_{\mathsf{red}}$ with the average runtime decrease of 3 min.

10 Conclusion and Future Work

10.1 Summary

Many compiler-based optimizations have been proposed to optimize a single objective ignoring other objectives, which might lead to drastic degradation of the ignored objectives. In this thesis, we presented approaches to solve multiobjective optimization problems at compile time. We considered three contradicting objectives that are critical for hard real-time systems: WCET, code size, and energy consumption.

Chapter 5 presents a novel compiler-based compression technique for hard real-time systems. The proposed compression technique considers WCET, energy consumption, and code size as objectives, but since the main goal of compression is to decrease code size as much as possible, this optimization demonstrates how a multiobjective optimization problem can be reformulated as a single-objective optimization problem that minimizes code size and satisfies, e.g. WCET and/or energy consumption constraints.

The proposed compression technique chooses and compresses chunks of an input code at compile time and decompresses them at runtime. The method compresses as many chunks as possible to decrease program size, which includes code size, and guarantees that a WCET constraint is satisfied despite the runtime decompression. The approach does not require expensive hardware to decompress code since a compiler inserts the code of the decompression routine and its calls into the output assembly code to decompress compressed chunks at runtime. For considered benchmarks, program size decreased by 11 % and WCET increased by 9 % on average.

The advantage of reformulating a multiobjective problem as a single-objective problem is that any single-objective problem results in a single optimal value of the objective. But this reformulation is only reasonable if a single objective function can be created from problem formulation. Moreover, putting objectives into constraints means that the values of constrained objectives might not be fully optimized. So another approach to solve multiobjective problems is to find a set of possible trade-offs between objectives. Software-based approaches proposed in this thesis can save time and effort when designing an embedded system: having a full set of possible trade-offs, a system designer can decide which parts of the system must be optimized.

Chapter 6 moves towards pure multiobjective compiler-based optimizations and evaluates evolutionary algorithms in terms of finding trade-offs between WCET, code size, and energy consumption. We showed that the algorithms produce a solution set only for small benchmarks. The approach becomes infeasible for large benchmarks due to very time-consuming WCET and energy consumption analyses which must be invoked to compute the objectives while executing evolutionary algorithms.

Chapter 7 presents a model based on machine learning that predicts WCET and energy consumption at compile time. We utilized the prediction model when running an evolutionary algorithm to substitute costly computed WCET and energy consumption with cheaper predictions. We showed that logistic regression, decision tree classifier, and AdaBoost classifier based on decision trees achieve a high prediction accuracy and do not degrade the quality of solution sets for many tested benchmarks. Predictions sped up the evolutionary algorithm by 3 h on average over all tested benchmarks.

To speed up further the evolutionary algorithm, Chapter 8 presents a novel search space reduction approach. It identifies those features – or dimensions – of a search space that influence the objective values and removes redundant features from the search space. Supplying the evolutionary algorithm with a reduced search space, the algorithm explores fewer search vectors to find a solution set and requires fewer expensive evaluations of the objectives. In the case of an exemplary multiobjective function inlining problem, for the tested benchmarks, the search space was reduced by 93 % on average. The search space reduction decreased the runtime of the evolutionary algorithm by 3 h on average compared to the original evolutionary algorithm and by 15 min compared to the evolutionary algorithm based on predictions. For many benchmarks, the quality of solution sets remained unchanged or was improved.

Chapter 9 combines predictions and the reduction technique. We reduced a search space, fitted a prediction model on the reduced search space, and executed the evolutionary algorithm on the reduced search space with predicted WCET and energy consumption. For the multiobjective function inlining problem, predictions on reduced search spaces decreased the average runtime of the evolutionary algorithm executed on the reduced search spaces by 3 min on average for 1/3 of tested benchmarks and increased it by 1 min on average for 2/3 of the benchmarks. For most benchmarks, predictions on reduced search spaces kept or improved the quality of solution sets returned by the evolutionary algorithm executed on the reduced search spaces without predictions.

To sum up, we have shown that when solving a multiobjective problem at compile time, we can speed up the evolutionary algorithm and improve the quality of solution sets by reducing a large search space and utilizing a prediction model to substitute time-consuming evaluations of the objectives either on the original or reduced search spaces.

10.2 Future Work

The results of the thesis suggest the following promising topics for future research:

Compression. The proposed compiler-based compression technique lacks a smart buffer management. We assumed that all chunks compressed at compile time are decompressed right at the beginning of a program and remain in a buffer during its execution. This limits the usage of the buffer because all decompressed chunks must simultaneously fit into it. Future work is to identify those places in the program where a chunk is used and must be decompressed; this should lead to a higher compression ratio since several chunks can share a memory slot within the buffer.

In this thesis, we considered only code compression and ignored data objects which can also be compressed. Future work is to develop a compiler-based data compression technique that can be used for hard real-time systems.

Prediction model. When building a prediction model, we showed that in most cases, it was reasonable to use a classification method to predict WCET and energy consumption since we observed only a limited number of unique objective values. But if the objective space of a problem consists of many different values, then a classifier might fail to make tighter predictions. Future work is to explore regression models for such problems.

Search Space Reduction. We applied search space reduction to search spaces represented by bit vectors, but we mentioned that the proposed approach can be also extended and applied to problems with search spaces represented by vectors that take values from a finite set of values. Future work is to evaluate the proposed search space reduction techniques for such search spaces.

Our search space reduction explores objective values to identify redundant dimensions of a search space along which changes in objective values are insignificant. We considered WCET and energy consumption as objectives. They were estimated by using AbsInt's tools. For both objectives, AbsInt's estimations rely on path analysis, which identifies program paths with the highest WCET and energy consumption values. The paths for these objectives might coincide, which leads to (almost) the same redundant dimensions. In general, energy analysis relies on average-case simulations. This means that redundant dimensions for WCET most probably will differ from redundant dimensions for energy consumption. Future work is to evaluate the proposed search space reduction technique by considering an energy model that uses average-case simulations.

General. We evaluated the proposed approaches only on one compiler-based optimization, namely, function inlining. But the approaches are general and can be used to perform any compiler-based optimization. Future work is to investigate generalizability of the approaches by applying them to other optimizations.

In this thesis, we focused on solving multiobjective problems and finding trade-offs between objectives. A natural next step is to design and involve a decision-maker that chooses the best trade-off or, at least, sorts or classifies trade-offs according to some criteria.

Bibliography

[ABS14] Nessrine Azzouz, Slim Bechikh, and Lamjed Ben Said. "Steady state IBEA assisted by MLP neural networks for expensive multi-objective optimization problems". In: *Annual Conference on Genetic and Evolutionary Computation*. ACM, July 2014. DOI: 10.1145/2576768.2598271.

[Abs22] AbsInt Angewandte Informatik, GmbH. *aiT Worst-Case Execution Time Analyzers*. 2022. URL: https://www.absint.com/ait/index.htm.

[AD14] Sebastian Altmeyer and Robert I. Davis. "On the correctness, optimality and precision of Static Probabilistic Timing Analysis". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014. DOI: 10.7873/date.2014.039.

[AE07] Vincent J. Amuso and Jason Enslin. "The Strength Pareto Evolutionary Algorithm 2 (SPEA2) applied to simultaneous multi-mission waveform design". In: *International Waveform Diversity and Design Conference (WDDC)*. IEEE, June 2007. DOI: 10.1109/wddc.2007.4339452.

[Ale92] Samuel O. Aletan. "An overview of RISC architecture". In: *ACM/SIGAPP Symposium on Applied computing technological challenges of the 1990's (SAC)*. ACM, 1992. DOI: 10.1145/143559.143570.

[Alt+16] Peter Altenbernd, Jan Gustafsson, Björn Lisper, and Friedhelm Stappert. "Early execution time-estimation through automatically generated timing models". In: *Real-Time Systems* 52.6 (Feb. 2016), pp. 731–760. DOI: 10.1007/s11241-016-9250-7.

[AMCMM12] Alfredo Arias-Montano, Carlos A. Coello Coello, and Efrén Mezura-Montes. "Multiobjective Evolutionary Algorithms in Aeronautical and Aerospace Engineering". In: *IEEE Transactions on Evolutionary Computation* 16.5 (Oct. 2012), pp. 662–694. DOI: 10.1109/tevc.2011.2169968.

[Ans20] Philip Ansari. "Compilerunterstützung für parametrische Schleifengrenzen auf Assemblersprachen-Ebene während einer WCET- Analyse." Bachelor Thesis. Hamburg University of Technology, School of Electrical Engineering, Computer Science and Mathematics, 2020.

[ARM09a] ARM Ltd. *Cortex-M0 Devices Generic User Guide*. ARM DUI 0497. 2009. URL: `https://developer.arm.com/documentation/dui0497/latest/`.

[ARM09b] ARM Ltd. *Cortex-M0 Technical Reference Manual*. ARM DDI 0432C. 2009. URL: `https://developer.arm.com/documentation/ddi0432/latest/`.

[Bak+14] Wafae Bakkali, Mohamed Tlich, Pascal Pagani, and Thierry Chonavel. "A measurement-based model of energy consumption for PLC modems". In: *18th IEEE International Symposium on Power Line Communications and Its Applications (ISPLC)*. IEEE, Mar. 2014. DOI: `10.1109/isplc.2014.6812326`.

[BCP02] Guillem Bernat, Antoine Colin, and Stefan M. Petters. "WCET analysis of probabilistic hard real-time systems". In: *23rd IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2002. DOI: `10.1109/real.2002.1181582`.

[Ben75] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (Sept. 1975), pp. 509–517. DOI: `10.1145/361002.361007`.

[BH07] Philippe Baufreton and Reinhold Heckmann. "Reliable and Precise WCET and Stack Size Determination for a Real-life Embedded Application". In: *Workshop on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. 2007, pp. 41–48.

[BJ03] B.V. Babu and M. Mathew Leenus Jehan. "Differential evolution for multi-objective optimizatior". In: *Congress on Evolutionary Computation (CEC)*. IEEE, 2003. DOI: `10.1109/cec.2003.1299429`.

[BMN01] Luca Benini, Alberto Macii, and Alberto Nannarelli. "Cached-code compression for energy minimization in embedded processors". In: *International symposium on Low power electronics and design (ISLPED)*. ACM, 2001. DOI: `10.1145/383082.383177`.

[Bon10] Talal Bonny. "Huffman-based Code Compression Techniques for Embedded Systems". PhD thesis. Karlsruhe Institute of Technology, 2010.

[Bon+17] 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)*. Leibniz-Zentrum für Informatik GmbH, 2017, 5:1–5:9. DOI: 10.4230/OASICS.WCET.2017.5.

[Bra02] Jürgen Branke. *Evolutionary Optimization in Dynamic Environments*. Boston, MA: Springer US, 2002. ISBN: 9781461509110.

[Bra91] Mark F. Bramlette. "Initialization, Mutation and Selection Methods in Genetic Algorithms for Function Optimization". In: *ICGA*. 1991.

[Bre01] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32. DOI: 10.1023/a:1010933404324.

[Bre13] Thomas Bress. *Effective LabVIEW programming*. Allendale, N.J: National Technology and Science Press, 2013. ISBN: 9781934891087.

[BS02] Hans-Georg Beyer and Hans-Paul Schwefel. "Evolution strategies – A comprehensive introduction". In: *Natural Computing* 1.1 (2002), pp. 3–52. DOI: 10.1023/a:1015059928466.

[BSL18] Eric Bradford, Artur M. Schweidtmann, and Alexei Lapkin. "Efficient multiobjective optimization employing Gaussian processes, spectral sampling and a genetic algorithm". In: *Journal of Global Optimization* 71.2 (Feb. 2018), pp. 407–438. DOI: 10.1007/s10898-018-0609-2.

[BZ11] Johannes Bader and Eckart Zitzler. "HypE: An Algorithm for Fast Hypervolume-Based Many-Objective Optimization". In: *Evolutionary Computation* 19.1 (Mar. 2011), pp. 45–76. DOI: 10.1162/evco_a_00009.

[Caz+12] Francisco J. Cazorla, Eduardo Quinones, Tullio Vardanega, Liliana Cucu-Grosjean, and Benoit Triquet. *PROARTIS: Probabilistically Analysable Real-Time Systems*. Research rep. INRIA, 2012. URL: https://hal.inria.fr/hal-00663329/document.

[CDS01] Michael Collins, Sanjoy Dasgupta, and Robert E. Schapire. "A Generalization of Principal Component Analysis to the Exponential Family". In: *14th International Conference on Neural Information Processing Systems: Natural and Synthetic (NIPS)*. 2001, pp. 617–624.

[CFM09] Daniel Cordes, Heiko Falk, and Peter Marwedel. “A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, Mar. 2009. DOI: 10.1109/cgo.2009.17.

[CG+12] Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quinones, and Francisco J. Cazorla. “Measurement-Based Probabilistic Timing Analysis for Multi-path Programs”. In: *24th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, July 2012. DOI: 10.1109/ecrts.2012.31.

[Che+12] Tianshi Chen, Ke Tang, Guoliang Chen, and Xin Yao. “A large population size can be unhelpful in evolutionary algorithms”. In: *Theoretical Computer Science* (June 2012), pp. 54–70. DOI: 10.1016/j.tcs.2011.02.016.

[CM05] Gilberto Contreras and Margaret Martonosi. “Power prediction for Intel XScale/spl reg/ processors using performance monitoring unit events”. In: *International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2005. DOI: 10.1109/lpe.2005.195518.

[Coe+19] Carlos A. Coello Coello, Silvia González Brambila, Josué Figueroa Gamboa, Ma Guadalupe Castillo Tapia, and Raquel Hernández Gómez. “Evolutionary multiobjective optimization: open research areas and some challenges lying ahead”. In: *Complex & Intelligent Systems* 6.2 (June 2019), pp. 221–236. DOI: 10.1007/s40747-019-0113-4.

[Cpl17] IBM ILOG Cplex. *V12. 8: User's Manual for CPLEX*. 2017.

[Cul+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. *Predictability Considerations in the Design of Multi-Core Embedded Systems*. May 2010. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.172.4533&rep=rep1&type=pdf.

[CXQ99] C.S. Chang, D.Y. Xu, and H.B. Quek. “Pareto-optimal set based multiobjective tuning of fuzzy automatic train operation for mass transit system”. In: *IEE Proceedings - Electric Power Applications* 146.5 (1999), p. 577. DOI: 10.1049/ip-epa:19990481.

[DBLJ14] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. "SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives". In: *Advances In Neural Information Processing Systems* (July 2014). arXiv: 1407.0202 [cs.LG].

[DCG19] Robert I. Davis and Liliana Cucu-Grosjean. "A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems". In: *Leibniz Transactions on Embedded Systems* (2019). DOI: 10.4230/LITES-V006-I001-A003.

[DD97] Indraneel Das and John E. Dennis. "A closer look at drawbacks of minimizing weighted sums of objectives for Pareto set generation in multicriteria optimization problems". In: *Structural Optimization* 14.1 (Aug. 1997), pp. 63–69. DOI: 10.1007/bf01197559.

[DD98] Indraneel Das and John E. Dennis. "Normal-Boundary Intersection: A New Method for Generating the Pareto Surface in Nonlinear Multicriteria Optimization Problems". In: *SIAM Journal on Optimization* 8.3 (Aug. 1998), pp. 631–657. DOI: 10.1137/s1052623496307510.

[DE02] Saumya Debray and William Evans. "Profile-guided code compression". In: *ACM SIGPLAN Notices* 37.5 (May 2002), pp. 95–105. DOI: 10.1145/543552.512542.

[Deb+02a] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE Transactions on Evolutionary Computation* 6.2 (Apr. 2002), pp. 182–197. DOI: 10.1109/4235.996017.

[Deb+02b] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. "Scalable multi-objective optimization test problems". In: *Congress on Evolutionary Computation (CEC)*. IEEE, 2002. DOI: 10.1109/cec.2002.1007032.

[Dem+20] A. M. Coutinho Demetrios, Daniele De Sensi, Arthur Francisco Lorenzon, Kyriakos Georgiou, Jose Nunez-Yanez, Kerstin Eder, and Samuel Xavier de Souza. "Performance and Energy Trade-Offs for Parallel Applications on Heterogeneous Multi-Processing Systems". In: *Energies* 13.9 (May 2020), p. 2409. DOI: 10.3390/en13092409.

[DM+16] Alan Díaz-Manríquez, Gregorio Toscano, Jose Hugo Barron-Zambrano, and Edgar Tello-Leal. "A Review of Surrogate Assisted Multiobjective Evolutionary Algorithms". In: *Computational Intelligence and Neuroscience* (2016), pp. 1–14. DOI: 10.1155/2016/9420460.

[Dor05] Informatik Centrum Dortmund. *ICD Low-Level Intermediate Representation backend infrastructure (LLIR). Developer Manual*. 2005.

[Dor09] Informatik Centrum Dortmund. *ICD-C Compiler framework. Developer Manual*. 2009.

[Dur+10] Juan J. Durillo, Antonio J. Nebro, Carlos A. Coello Coello, José Garcia-Nieto, Francisco Luna, and Enrique Alba. "A Study of Multiobjective Metaheuristics When Solving Parameter Scalable Problems". In: *IEEE Transactions on Evolutionary Computation* 14.4 (Aug. 2010), pp. 618–635. DOI: 10.1109/tevc.2009.2034647.

[EB01] Stewart Edgar and Alan Burns. "Statistical analysis of WCET for scheduling". In: *22nd IEEE Real-Time Systems Symposium (RTSS)*. IEEE, Dec. 2001. DOI: 10.1109/real.2001.990614.

[EB97] Agoston E. Eiben and Thomas Bäck. "Empirical Investigation of Multiparent Recombination Operators in Evolution Strategies". In: *Evolutionary Computation* 5.3 (Sept. 1997), pp. 347–365. DOI: 10.1162/evco.1997.5.3.347.

[EBN05] Michael Emmerich, Nicola Beume, and Boris Naujoks. "An EMO Algorithm Using the Hypervolume Measure as Selection Criterion". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 62–76. DOI: 10.1007/978-3-540-31880-4_5.

[Ehr06] Matthias Ehrgott. *Multicriteria Optimization*. Springer-Verlag GmbH, Jan. 2006. ISBN: 9783540276593.

[EHW17] Mohammad Hemmat Esfe, Mohammad Hadi Hajmohammad, and Somchai Wongwises. "Pareto Optimal Design of Thermal Conductivity and Viscosity of NDCo3O4 Nanofluids by MOPSO and NSGA II Using Response Surface Methodology". In: *Current Nanoscience* 14.1 (Dec. 2017), pp. 62–70. DOI: 10.2174/1573413713666170914103043.

[ERR94] Agoston E. Eiben, Paul-Erik Raué, and Zsófia Ruttkay. "Genetic algorithms with multi-parent recombination". In: *Parallel Problem Solving from Nature*. Springer Berlin Heidelberg, 1994, pp. 78–87. DOI: 10.1007/3-540-58484-6_252.

[ES11a] Agoston E. Eiben and S. K. Smit. "Evolutionary Algorithm Parameters and Methods to Tune Them". In: *Autonomous Search*. Springer Berlin Heidelberg, 2011, pp. 15–36. DOI: 10.1007/978-3-642-21434-9_2.

[ES11b] Agoston E. Eiben and S. K. Smit. “Parameter tuning for configuring and analyzing evolutionary algorithms”. In: *Swarm and Evolutionary Computation* 1.1 (Mar. 2011), pp. 19–31. DOI: 10.1016/j.swevo.2011.02.001.

[ES15] Agoston E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Springer Berlin Heidelberg, 2015. DOI: 10.1007/978-3-662-44874-8.

[Eve15] Shimon Even. *Graph Algorithms*. Cambridge University Press, Feb. 2015. 202 pp. ISBN: 0521736536. URL: https://www.ebook.de/de/product/16350692/shimon_even_graph_algorithms.html.

[FA04] Marco Farina and Paolo Amato. “A fuzzy definition of "optimality" for many-criteria optimization problems”. In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 34.3 (May 2004), pp. 315–326. DOI: 10.1109/tsmca.2004.824873.

[Fal+16] 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 (WCET)*. Leibniz-Zentrum für Informatik GmbH, 2016, 2:1–2:10. DOI: 10.4230/OASICS.WCET.2016.2.

[Fal+20] Heiko Falk, Shashank Jadhav, Arno Luppold, Kateryna Muts, Dominic Oehlert, Nina Piontek, and Mikko Roth. “Compilation for Real-Time Systems a Decade After Predator”. In: *A Journey of Embedded and Cyber-Physical Systems*. Springer International Publishing, July 2020, pp. 151–169. DOI: 10.1007/978-3-030-47487-4_10.

[FF93] Carlos M. Fonseca and Peter J. Fleming. “Genetic Algorithms for Multiobjective Optimization: FormulationDiscussion and Generalization”. In: *5th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers Inc., 1993, 416–423. DOI: 10.5555/645513.657757.

[FF96] David B. Fogel and Lawrence J. Fogel. “An introduction to evolutionary programming”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1996, pp. 21–33. DOI: 10.1007/3-540-61108-8_28.

[FH10] Ana Ferreira and Laurens De Haan. *Extreme Value Theory*. Springer New York, 2010. 436 pp. ISBN: 144192020X. URL: https://www.ebook.de/de/product/13413833/ana_ferreira_laurens_de_haan_extreme_value_theory.html.

[FK09] Heiko Falk and Jan C. Kleinsorge. "Optimal static WCET-aware scratchpad allocation of program code". In: *46th Annual Design Automation Conference (DAC)*. ACM, 2009. DOI: 10.1145/1629911.1630101.

[FL10] Heiko Falk and Paul Lokuciejewski. "A compiler framework for the reduction of worst-case execution times". In: *Real-Time Systems* 46.2 (July 2010), pp. 251–300. DOI: 10.1007/s11241-010-9101-x.

[FS06] Heiko Falk and Martin Schwarzer. "Loop Nest Splitting for WCET-Optimization and Predictability Improvement". In: *Workshop on Embedded Systems for Real Time Multimedia (ESTIMedia)*. IEEE, Oct. 2006, pp. 115–120. DOI: 10.1109/estmed.2006.321283.

[FS97] Yoav Freund and Robert E Schapire. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". In: *Journal of Computer and System Sciences* 55.1 (Aug. 1997), pp. 119–139. DOI: 10.1006/jcss.1997.1504.

[Gau09] Carl Friedrich Gauß. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*. Sumtibus Frid. Perthes et I. H. Besser, 1809. DOI: 10.3931/E-RARA-522.

[Gin55] Caroline Gini. "Italian: Variabilità e Mutabilità (Variability and Mutability)". In: *Memo rie di Metodologica Statistica* (1955).

[Gol13] David E. Goldberg. *The Design of Innovation*. Springer US, Mar. 14, 2013. 248 pp. ISBN: 9781475736434. URL: https://www.ebook.de/de/product/25190286/david_e_goldberg_the_design_of_innovation.html.

[Gol89] David Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Publishing Company, 1989. ISBN: 9780201157673.

[GP20] Manolis Georgioudakis and Vagelis Plevris. "A Comparative Study of Differential Evolution Variants in Constrained Structural Optimization". In: *Frontiers in Built Environment* (July 2020). DOI: 10.3389/fbuil.2020.00102.

[GS96] Liam Goudge and Simon Segars. "Thumb: reducing the cost of 32-bit RISC performance in portable and consumer applications". In: *Technologies for the Information Superhighway Digest of Papers (COMPCON)*. IEEE, 1996. DOI: 10.1109/cmpcon.1996.501765.

[GSE18] Kyriakos Georgiou, Samuel Xavier de Souza, and Kerstin Eder. "The IoT Energy Challenge: A Software Perspective". In: *IEEE Embedded Systems Letters* 10.3 (Sept. 2018), pp. 53–56. DOI: 10.1109/les.2017.2741419.

[Gur21] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2021. URL: https://www.gurobi.com.

[GWW10] Adam Gontarz, Lukas Weiss, and Konrad Wegener. "Energy Consumption Measurement with a Multichannel Measurement System on a machine tool". In: *International Conference on Innovative Technologies (IN-TECH)*. ETH Zurich, Sept. 2010. DOI: 10.3929/ETHZ-A-007577653.

[Hec+03] Reinhold Heckmann, Marc Langebach, Stephan Thesing, and Reinhard Wilhelm. "The influence of processor architecture on the design and the results of WCET tools". In: *Proceedings of the IEEE* 91.7 (July 2003), pp. 1038–1054. DOI: 10.1109/jproc.2003.814618.

[Her+15] Carles Hernandez, Jaume Abella, Francisco J. Cazorla, Jan Andersson, and Andrea Gianarro. "Towards Making a LEON3 Multicore Compatible with Probabilistic Timing Analysis". In: *Data Systems in Aerospace Conference (DASIA)*. Mar. 2015. URL: https://people.ac.upc.edu/jabella/DASIA2015.pdf.

[HHM09] Jeffery P. Hansen, Scott A. Hissam, and Gabriel A. Moreno. "Statistical-Based WCET Estimation and Validation". In: *9th Workshop on Worst-Case Execution Time Analysis (WCET)*. Leibniz-Zentrum für Informatik GmbH, July 2009. URL: http://drops.dagstuhl.de/opus/volltexte/2009/2291.

[Hid07] Ariya Hidayat. *FastLZ - lightning-fast lossless compression library*. 2007. URL: http://www.fastlz.org.

[HJ98] Michael P. Hansen and Andrzej Jaszkiewicz. *Evaluating the quality of approximations to the non-dominated set*. Tech. rep. Mar. 1998. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.5279.

[HLHG14] José Miguel Hernández-Lobato, Matthew W. Hoffman, and Zoubin Ghahramani. *Predictive Entropy Search for Efficient Global Optimization of Black-box Functions*. 2014. arXiv: 1406.2541 [stat.ML].

[HME97] Robert Hinterding, Zbigniew Michalewicz, and Agoston E. Eiben. "Adaptation in evolutionary computation: a survey". In: *International Conference on Evolutionary Computation (ICEC)*. IEEE, 1997. DOI: 10.1109/icec.1997.592270.

[HMH18] 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)*. Leibniz-Zentrum für Informatik GmbH, 2018, 5:1–5:12. DOI: 10.4230/OASICS.WCET.2018.5.

[HP03] John Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. San Francisco, CA: Morgan Kaufmann Publishers, 2003. ISBN: 9788178672663.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer New York, 2009. DOI: 10.1007/978-0-387-84858-7.

[Huf52] David Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (Sept. 1952), pp. 1098–1101. DOI: 10.1109/jrproc.1952.273898.

[IBM98] IBM. *CodePack PowerPC Code Compression Utility. User's Manual Version 3.0*. 1998.

[Inf08] Infineon Technologies AG. *User Manual. Instruction Set*. Version 1.3.8. Volume 2. 2008. URL: https://www.infineon.com/dgdl/tc_v131_instructionset_v138.pdf?fileId=db3a304412b407950112b409b6dd0352.

[Inf12] Infineon Technologies AG. *User Manual. Core Architecture*. Version 1.6. Volume 1. 2012. URL: https://www.infineon.com/dgdl/tc1_6__architecture_vol1.pdf?fileId=db3a3043372d5cc801373b0f374d5d67.

[Inf14] Infineon Technologies AG. *Data Sheet TC1797*. Version 1.3. 2014. URL: https://www.infineon.com/dgdl/TC1797_DS_V13.pdf?fileId=db3a30431ed1d7b2011efeaa4ad16b6d.

[Jam+21] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer-Verlag GmbH, Aug. 2021. ISBN: 1071614177.

[JF19] Shashank Jadhav and Heiko Falk. "Multi-Objective Optimization for the Compiler of Real-Time Systems based on Flower Pollination Algorithm". In: *22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, May 2019, pp. 45–48. DOI: 10.1145/3323439.3323977.

[JKM92] Johannes Jahn, Jan Klose, and Andreas Merkel. "On the Application of a Method of Reference Point Approximation to Bicriterial Optimization Problems in Chemical Engineering". In: *Lecture Notes in Economics and Mathematical Systems*. Springer Berlin Heidelberg, 1992, pp. 478–491. DOI: 10.1007/978-3-642-51682-5_31.

[Jon07] Kenneth De Jong. "Parameter Setting in EAs: a 30 Year Perspective". In: *Parameter Setting in Evolutionary Algorithms*. Springer Berlin Heidelberg, 2007, pp. 1–18. DOI: 10.1007/978-3-540-69432-8_1.

[KD06] Saku Kukkonen and Kalyanmoy Deb. "Improved Pruning of Non-Dominated Solutions Based on Crowding Distance for Bi-Objective Optimization Problems". In: *International Conference on Evolutionary Computation (CEC)*. IEEE, 2006. DOI: 10.1109/cec.2006.1688443.

[KE97] James Kennedy and Russell C. Eberhart. "A discrete binary version of the particle swarm algorithm". In: *International Conference on Systems, Man, and Cybernetics (ICSMC)*. IEEE, 1997. DOI: 10.1109/icsmc.1997.637339.

[Kel15] Timon Kelter. "WCET Analysis and Optimization for Multi-Core Real-Time Systems". PhD thesis. TU Dortmund, Department of Computer Science, 2015.

[Kir03] Raimund Kirner. "Extending Optimising Compilation to Support Worst-Case Execution Time Analysis". PhD thesis. Technische Universität Wien, 2003.

[Kis+00] Toru Kisuki, Peter Knijnenburg, Michael F. P. O'Boyle, and Harry Wijshoff. *Iterative Compilation in Program Optimization*. May 2000. URL: https://liacs.leidenuniv.nl/assets/PDF/TechRep/tr00-03.pdf.

[KKMS99] Darko Kirovski, Johnson Kin, and William H. Mangione-Smith. In: *International Journal of Parallel Programming* 27.6 (1999), pp. 457–475. DOI: 10.1023/a:1018728216668.

[KKO02] Peter M. W. Knijnenburg, Toru Kisuki, and Michael F. P. O'Boyle. "Iterative Compilation". In: *Embedded Processor Design Challenges*. Springer Berlin Heidelberg, 2002, pp. 171–187. DOI: 10.1007/3-540-45874-3_10.

[KL04a] Saku Kukkonen and Jouni Lampinen. "An Extension of Generalized Differential Evolution for Multi-objective Optimization with Constraints". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 752–761. DOI: 10.1007/978-3-540-30217-9_76.

[KL04b] Saku Kukkonen and Jouni Lampinen. "Comparison of Generalized Differential Evolutions to Other Multi-objective Evolutionary Algorithms". In: *4th European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS)*. 2004. URL: https://lutpub.lut.fi/bitstream/handle/10024/76984/isbn9789522652362.pdf?sequence=4#page=129.

[KL05] Saku Kukkonen and Jouni Lampinen. "GDE3: the third evolution step of generalized differential evolution". In: *Congress on Evolutionary Computation (CEC)*. IEEE, 2005, pp. 443–450. DOI: 10.1109/cec.2005.1554717.

[Kle] Jan Christopher Kleinsorge. "WCET-centric code allocation for scratchpad memories". MA thesis. Dortmund University of Technology. URL: https://ls12-www.cs.tu-dortmund.de/daes/media/documents/theses/kleinsorge.pdf.

[Knu71] Donald E. Knuth. "An empirical study of FORTRAN programs". In: *Software: Practice and Experience* 1.2 (Apr. 1971), pp. 105–133. DOI: 10.1002/spe.4380010203.

[Kop11] Hermann Kopetz. *Real-Time Systems*. Springer-Verlag GmbH, Apr. 2011. ISBN: 9781441982377.

[Kos+20] Dmitry Kosolobov, Daniel Valenzuela, Gonzalo Navarro, and Simon J. Puglisi. "Lempel–Ziv-Like Parsing in Small Space". In: *Algorithmica* 82.11 (May 2020), pp. 3195–3215. DOI: 10.1007/s00453-020-00722-6.

[Kos88] Juhani Koski. "Multicriteria Truss Optimization". In: *Multicriteria Optimization in Engineering and in the Sciences*. Springer US, 1988, pp. 263–307. DOI: 10.1007/978-1-4899-3734-6_9.

[Kra10] Oliver Kramer. "Evolutionary self-adaptation: a survey of operators and strategy parameters". In: *Evolutionary Intelligence* 3.2 (Feb. 2010), pp. 51–65. DOI: 10.1007/s12065-010-0035-y.

[KT05] Min Kong and Peng Tian. "A Binary Ant Colony Optimization for the Unconstrained Function Optimization Problem". In: *Computational Intelligence and Security*. Springer Berlin Heidelberg, 2005, pp. 682–687. DOI: 10.1007/11596448_101.

[KTZ06] Joshua D. Knowles, Lothar Thiele, and Eckart Zitzler. *A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers*. Tech. rep. 2006. DOI: 10.3929/ETHZ-B-000023822.

[Kä+08] Daniel Kästner, Reinhard Wilhelm, Reinhold Heckmann, Marc Schlickling, Markus Pister, Marek Jersak, Kai Richter, and Christian Ferdinand. "Timing Validation of Automotive Software". In: *Communications in Computer and Information Science*. Springer Berlin Heidelberg, 2008, pp. 93–107. DOI: 10.1007/978-3-540-88479-8_8.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2004. DOI: 10.1109/cgo.2004.1281665.

[Lam01] Jouni Lampinen. *DE's Selection Rule for Multiobjective Optimization*. Tech. rep. Lappeenranta University of Technology, 2001. URL: http://www.it.lut.fi/kurssit/03-04/010778000/MODE.pdf.

[LGT08] Adam Wade Lewis, Soumik Ghosh, and Nian-Feng Tzeng. "Runtime Energy Consumption Estimation Based on Workload in Server Systems". In: *Workshop on Power Aware Computing and Systems (HotPower)*. USENIX Association, 2008. URL: http://www.usenix.org/events/hotpower08/tech/full_papers/lewis/lewis.pdf.

[LHJ02] Haris Lekatsas, Jörg Henkel, and Venkata Jakkula. "Design of an one-cycle decompression hardware for performance increase in embedded systems". In: *39th Conference on Design Automation (DAC)*. ACM, 2002. DOI: 10.1145/513918.513929.

[LHW01] Haris Lekatsas, Jörg Henkel, and Wayne Wolf. "Design and simulation of a pipelined decompression architecture for embedded systems". In: *14th International Symposium on Systems Synthesis (ISSS)*. ACM, 2001. DOI: 10.1145/500001.500015.

[LM11] Paul Lokuciejewski and Peter Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer Netherlands, 2011. DOI: 10.1007/978-90-481-9929-7.

[LM97] Yau-Tsun S. Li and Sharad Malik. "Performance analysis of embedded software using implicit path enumeration". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.12 (1997), pp. 1477–1487. DOI: 10.1109/43.664229.

[LMW95] Yau-Tsun S. Li, Sharad Malik, and Andrew Wolfe. "Efficient microarchitecture modeling and path analysis for real-time software". In: *16th Real-Time Systems Symposium (REAL)*. IEEE, 1995, pp. 298–307. DOI: 10.1109/REAL.1995.495219.

[Lok+09] Paul Lokuciejewski, Fatih Gedikli, Peter Marwedel, and Katharina Morik. "Automatic WCET Reduction by Machine Learning Based Heuristics for Function Inlining". In: *SMART*. 2009.

[Lok+11] Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. "Approximating Pareto optimal compiler optimization sequences – a trade-off between WCET, ACET and code size". In: *Software: Practice and Experience* 41.12 (May 2011), pp. 1437–1458. DOI: 10.1002/spe.1079.

[LPM00] Charles Lefurgy, Eva Piccininni, and Trevor Mudge. "Reducing code size with run-time decompression". In: *6th International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2000. DOI: 10.1109/hpca.2000.824352.

[LPMS97] Chunho Lee, M. Potkonjak, and W.H. Mangione-Smith. "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems". In: *30th Annual International Symposium on Microarchitecture (MICRO)*. IEEE, 1997. DOI: 10.1109/micro.1997.645830.

[Man+19] Arunmozhi Manimuthu, Anh Vu Le, Rajesh Elara Mohan, Prabahar Veerajagadeshwar, Nguyen Huu Khanh Nhan, and Ku Ping Cheng. "Energy Consumption Estimation Model for Complete Coverage of a Tetromino Inspired Reconfigurable Surface Tiling Robot". In: *Energies* 12.12 (June 2019), p. 2257. DOI: 10.3390/en12122257.

[Mar11] Peter Marwedel. *Embedded system design: embedded systems foundations of cyber-physical systems*. Dordrecht: Springer, 2011. ISBN: 9789400702561.

[Mar+14] André Maroneze, Sandrine Blazy, David Pichardie, and Isabelle Puaut. "A Formally Verified WCET Estimation Tool". In: *14th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Leibniz-Zentrum für Informatik GmbH, 2014, pp. 11–20. DOI: 10.4230/OASICS.WCET.2014.11.

[MAT22] MATLAB. Natick, Massachusetts: The MathWorks Inc., 2022.

[Maz+19] Atanu Mazumdar, Tinkle Chugh, Kaisa Miettinen, and Manuel López-Ibáñez. "On Dealing with Uncertainties from Kriging Models in Offline Data-Driven Evolutionary Multiobjective Optimization". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2019, pp. 463–474. DOI: 10.1007/978-3-030-12598-1_37.

[MC13] Saúl Zapotecas Martínez and Carlos A. Coello Coello. "MOEA/D assisted by RBF networks for expensive multi-objective optimization problems". In: *5th Annual Conference on Genetic and Evolutionary Computation (GECCO)*. ACM, 2013. DOI: 10.1145/2463372.2465805.

[MF20a] Kateryna Muts and Heiko Falk. "Compiler-based WCET prediction performing function specialization". In: *23rd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, May 2020. DOI: 10.1145/3378678.3391879.

[MF20b] Kateryna Muts and Heiko Falk. "Multi-Criteria Function Inlining for Hard Real-Time Systems". In: *28th International Conference on Real-Time Networks and Systems (RTNS)*. ACM, June 2020. DOI: 10.1145/3394810.3394819.

[MF21a] Kateryna Muts and Heiko Falk. "Predicting Objectives on a Reduced Search Space of Multiobjective Function Inlining". In: *24th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, Nov. 2021. DOI: 10.1145/3493229.3493303.

[MF21b] Kateryna Muts and Heiko Falk. "Predicting Worst-Case Execution Times During Multi-Criterial Function Inlining". In: *7th International Conference on Machine Learning, Optimization, and Data Science (LOD)*. Springer International Publishing, Oct. 2021. DOI: 10.1007/978-3-030-95467-3_21.

[Mis+15] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. "A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints". In: *ACM Special Interest Group on Programming Languages Notices* 50.4 (May 2015), pp. 267–281. DOI: 10.1145/2775054.2694373.

[MIY01] Achille Messac and Amir Ismail-Yahaya. "Required Relationship Between Objective Function and Pareto Frontier Orders: Practical Implications". In: *AIAA Journal* 39.11 (Nov. 2001), pp. 2168–2174. DOI: 10.2514/2.1213.

[MIYM03] Achille Messac, Amir Ismail-Yahaya, and Christopher A. Mattson. "The normalized normal constraint method for generating the Pareto frontier". In: *Structural and Multidisciplinary Optimization* 25.2 (July 2003), pp. 86–98. DOI: 10.1007/s00158-002-0276-1.

[MLF18] Kateryna Muts, Arno Luppold, and Heiko Falk. "Multi-Criteria Compiler-Based Optimization of Hard Real-Time Systems". In: *21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, May 2018. DOI: 10.1145/3207719.3207730.

[MLF19] Kateryna Muts, Arno Luppold, and Heiko Falk. "Compiler-Based Code Compression for Hard Real-Time Systems". In: *22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, May 2019. DOI: 10.1145/3323439.3323976.

[MM02] Achille Messac and Christopher A. Mattson. "Generating Well-Distributed Sets of Pareto Points for Engineering Design Using Physical Programming". In: *Optimization and Engineering* 3.4 (2002), pp. 431–450. DOI: 10.1023/a:1021179727569.

[Moh18] Mehryar Mohri. *Foundations of machine learning*. Cambridge, Massachusetts: The MIT Press, 2018. ISBN: 0262039400.

[Muc98] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. 1st ed. Morgan Kaufmann Publishers Inc., 1998, pp. 607–608, 657. ISBN: 1558603204.

[NE07] Volker Nannen and Agoston E. Eiben. "Efficient relevance estimation and value calibration of evolutionary algorithm parameters". In: *Congress on Evolutionary Computation (CEC)*. IEEE, Sept. 2007. DOI: 10.1109/cec.2007.4424460.

[Net+03] E. Wanderley Netto, Rodolfo Azevedo, Paulo Centoducatte, and Guido Araujo. "Mixed static/dynamic profiling for dictionary based code compression". In: *International Symposium on System-on-Chip (ISSOC)*. IEEE, 2003. DOI: 10.1109/issoc.2003.1267745.

[OLF17] Dominic Oehlert, Arno Luppold, and Heiko Falk. "Bus-Aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems". In: *29th Euromicro Conference on Real-Time Systems (ECRTS)*. Leibniz-Zentrum für Informatik GmbH, 2017, 1:1–1:22. DOI: 10.4230/LIPIcs.ECRTS.2017.1.

[OLF18] Dominic Oehlert, Arno Luppold, and Heiko Falk. "Compilation for real-time systems: an overview of the WCET-aware C compiler WCC". In: *9th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS)*. TUHH Universitätsbibliothek, 2018. DOI: 10.15480/882.2271.

[Omo89] Stephen M. Omohundro. *Five Balltree Construction Algorithms*. Tech. rep. ICSI Technical Report TR-89-063. 1989. URL: http://www.icsi.berkeley.edu/ftp/global/pub/techreports/1989/tr-89-063.pdf.

[Oza+09] Haluk Ozaktas, Karine Heydemann, Christine Rochange, and Hugues Cassé. *Impact of Code Compression on Estimated Worst-Case Execution Times*. Dec. 2009. URL: https://hal.inria.fr/inria-00441964/document.

[PC11] Antonin Ponsich and Carlos A. Coello Coello. "Differential Evolution performances for the solution of mixed-integer constrained process engineering problems". In: *Applied Soft Computing* 11.1 (Jan. 2011), pp. 399–409. DOI: 10.1016/j.asoc.2009.11.030.

[Pea01] Karl Pearson. "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (Nov. 1901), pp. 559–572. DOI: 10.1080/14786440109462720.

[Ped+12] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel Nothman, Gilles Louppe, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* (Jan. 2012). arXiv: 1201.0490v4 [cs.LG].

[Pet00] Stefan M. Petters. "Bounding the execution time of real-time tasks on modern processors". In: *7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2000. DOI: 10.1109/rtcsa.2000.896433.

[PHB13] James Pallister, Simon Hollis, and Jeremy Bennett. *BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms*. Aug. 2013. arXiv: 1308.5174 [cs.PF].

[PJ13] Suresh Purini and Lakshya Jain. “Finding good optimization sequences covering program space”. In: *ACM Transactions on Architecture and Code Optimization* 9.4 (Jan. 2013), pp. 1–23. DOI: 10.1145/2400682.2400715.

[PJZ08] Chen Peng, Li Jian, and Liu Zhiming. “Solving 0-1 Knapsack Problems by a Discrete Binary Version of Differential Evolution”. In: *2nd International Symposium on Intelligent Information Technology Application (IITA)*. IEEE, Dec. 2008. DOI: 10.1109/iita.2008.538.

[PP17] Alberto Policriti and Nicola Prezza. “LZ77 Computation Based on the Run-Length Encoded BWT”. In: *Algorithmica* 80.7 (July 2017), pp. 1986–2011. DOI: 10.1007/s00453-017-0327-z.

[PS17] Pramudita Satria Palar and Koji Shimoyama. “On multi-objective efficient global optimization via universal Kriging surrogate model”. In: *Congress on Evolutionary Computation (CEC)*. IEEE, June 2017. DOI: 10.1109/cec.2017.7969368.

[PSG10] Chunhua Peng, Huijuan Sun, and Jianfeng Guo. “Multi-objective optimal PMU placement using a non-dominated sorting differential evolution algorithm”. In: *International Journal of Electrical Power & Energy Systems* 32.8 (Oct. 2010), pp. 886–892. DOI: 10.1016/j.ijepes.2010.01.024.

[Pug16] Simon J Puglisi. “Lempel-Ziv Compression”. In: *Encyclopedia of Algorithms*. Springer New York, 2016, pp. 1095–1100. DOI: 10.1007/978-1-4939-2864-4_634.

[PW07] Shlomit S. Pinter and Israel Waldman. “Selective Code Compression Scheme for Embedded Systems”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 298–316. DOI: 10.1007/978-3-540-71528-3_19.

[Qui86] John Ross Quinlan. “Induction of decision trees”. In: *Machine Learning* 1.1 (Mar. 1986), pp. 81–106. DOI: 10.1007/bf00116251.

[Ras04] Carl Edward Rasmussen. “Gaussian Processes in Machine Learning”. In: *Advanced Lectures on Machine Learning*. Springer Berlin Heidelberg, 2004, pp. 63–71. DOI: 10.1007/978-3-540-28650-9_4.

[Rec94] Ingo Rechenberg. *Evolutionsstrategie '94*. Stuttgart: Frommann-Holzboog, 1994. ISBN: 9783772816420.

[Rek08] Ioannis T. Rekanos. "Shape Reconstruction of a Perfectly Conducting Scatterer Using Differential Evolution and Particle Swarm Optimization". In: *IEEE Transactions on Geoscience and Remote Sensing* 46.7 (July 2008), pp. 1967–1974. DOI: 10.1109/tgrs.2008.916635.

[RLF18] Mikko Roth, Arno Luppold, and Heiko Falk. "Measuring and Modeling Energy Consumption of Embedded Systems for Optimizing Compilers". In: *21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, May 2018. DOI: 10.1145/3207719.3207729.

[RRM18] Victor Henrique Alves Ribeiro and Gilberto Reynoso-Meza. "Multi-objective Support Vector Machines Ensemble Generation for Water Quality Monitoring". In: *Congress on Evolutionary Computation (CEC)*. IEEE, July 2018. DOI: 10.1109/cec.2018.8477745.

[San+17] Mário Santos, João Saraiva, Zoltán Porkoláb, and Dániel Krupp. "Energy Consumption Measurement of C/C++ Programs Using Clang Tooling". In: *6th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA)*. Sept. 2017. URL: http://ceur-ws.org/Vol-1938/paper-san.pdf.

[Sch+12] Simon Schubert, Dejan Kostic, Willy Zwaenepoel, and Kang G. Shin. "Profiling Software for Energy Consumption". In: *International Conference on Green Computing and Communications (GreenCom)*. IEEE, Nov. 2012. DOI: 10.1109/greencom.2012.86.

[Sch77] Hans-Paul Schwefel. "Evolutionsstrategien für die numerische Optimierung". In: *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*. Birkhäuser Basel, 1977, pp. 123–176. DOI: 10.1007/978-3-0348-5927-1_5.

[Sch85] J. David Schaffer. "Multiple Objective Optimization with Vector Evaluated Genetic Algorithms". In: *1st International Conference on Genetic Algorithms (ICGA)*. L. Erlbaum Associates Inc., 1985, 93–100. ISBN: 0805804269.

[SD94] Niranjan Srinivas and Kalyanmoy Deb. "Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms". In: *Evolutionary Computation* 2.3 (Sept. 1994), pp. 221–248. DOI: 10.1162/evco.1994.2.3.221.

[SMAL12] Renato de S. Motta, Silvana M. B. Afonso, and Paulo R. M. Lyra. "A modified NBI and NC method for the solution of N-multiobjective optimization problems". In: *Structural and Multidisciplinary Optimization* 46.2 (Jan. 2012), pp. 239–259. DOI: 10.1007/s00158-011-0729-5.

[SP10] Marc Schlickling and Markus Pister. "Semi-automatic derivation of timing models for WCET analysis". In: *ACM SIGPLAN Notices* 45.4 (Apr. 2010), pp. 67–76. DOI: 10.1145/1755951.1755899.

[SP97] Rainer Storn and Kenneth Price. "Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces". In: *Journal of Global Optimization* 11.4 (1997), pp. 341–359. DOI: 10.1023/a:1008202821328.

[SR96] Patrick D. Surry and Nicholas J. Radcliffe. "Inoculation to initialise evolutionary search". In: *Evolutionary Computing*. Springer Berlin Heidelberg, 1996, pp. 269–285. DOI: 10.1007/bfb0032789.

[SŠH09] András Szőllős, Miroslav Šmíd, and Jaroslav Hájek. "Aerodynamic optimization via multi-objective micro-genetic algorithm with range adaptation, knowledge-based reinitialization, crowding and ϵ-dominance". In: *Advances in Engineering Software* 40.6 (June 2009), pp. 419–430. DOI: 10.1016/j.advengsoft.2008.07.002.

[Sta08] Richard Stallman. *Using the GNU compiler collection: for GCC version 4.3.3*. Boston, MA: SoHo Books GNU Press, 2008. ISBN: 9781441412768.

[STD16] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. "A Reconfiguration Algorithm for Power-Aware Parallel Applications". In: *ACM Transactions on Architecture and Code Optimization* 13.4 (Dec. 2016), pp. 1–25. DOI: 10.1145/3004054.

[Tab+15] Mohammad Tabatabaei, Jussi Hakanen, Markus Hartikainen, Kaisa Miettinen, and Karthik Sindhya. "A survey on handling computationally expensive multiobjective optimization problems using surrogates: non-nature inspired methods". In: *Structural and Multidisciplinary Optimization* 52.1 (Mar. 2015), pp. 1–25. DOI: 10.1007/s00158-015-1226-z.

[Tea] *Final Report on Architecture-Level Energy Usage, Timing and Security Modeling and on Prototype*. Public deliverable D4.4 of the TeamPlay Horizon2020 project. 2019. URL: https://teamplay-h2020.eu/index.php?page=deliverables.

[Tho+10] John Thomson, Michael O'Boyle, Grigori Fursin, and Björn Franke. "Reducing Training Time in a One-Shot Machine Learning-Based Compiler". In: *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2010, pp. 399–407. DOI: 10.1007/978-3-642-13374-9_28.

[Tim00] Matt Timmermans. *BICOM BIjective COMpressor*. 2000. URL: http://www3.sympatico.ca/mt0000/bicom/.

[Tiw+96] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike T.-C. Lee. "Instruction level power analysis and optimization of software". In: *9th International Conference on VLSI Design (ICVD)*. IEEE, 1996. DOI: 10.1109/icvd.1996.489624.

[TN65] Ralph Turvey and A. Robert Nobay. "On Measuring Energy Consumption". In: *The Economic Journal* 75 (Dec. 1965). DOI: 10.2307/2229676.

[Tri+03] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. "Compiler optimization-space exploration". In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2003, 204–215. DOI: 10.1109/cgo.2003.1191546.

[Tur37] Alan Mathison Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230.

[VT04] Jakob Vesterstrom and René Thomsen. "A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems". In: *Congress on Evolutionary Computation (CEC)*. IEEE, 2004. DOI: 10.1109/cec.2004.1331139.

[Wan+10] Ling Wang, Xiping Fu, Muhammad Ilyas Menhas, and Minrui Fei. "A Modified Binary Differential Evolution Algorithm". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 49–57. DOI: 10.1007/978-3-642-15597-0_6.

[Wan+12] Ling Wang, Xiping Fu, Yunfei Mao, Muhammad Ilyas Menhas, and Minrui Fei. "A novel modified binary differential evolution algorithm and its applications". In: *Neurocomputing* 98 (Dec. 2012), pp. 55–75. DOI: 10.1016/j.neucom.2011.11.033.

[Wan+14] Ling Wang, Haoqi Ni, Weifeng Zhou, Panos M. Pardalos, Jiating Fang, and Minrui Fei. "MBPOA-based LQR controller and its application to the double-parallel inverted pendulum system". In: *Engineering Applications of Artificial Intelligence* 36 (Nov. 2014), pp. 262–268. DOI: 10.1016/j.engappai.2014.07.023.

[WBB13] Luc Wismans, Eric Van Berkum, and Michiel Bliemer. "Acceleration of Solving the Dynamic Multi-Objective Network Design Problem Using Response Surface Methods". In: *Journal of Intelligent Transportation Systems* 18.1 (Feb. 2013), pp. 17–29. DOI: 10.1080/15472450.2013.773250.

[WC92] Andrew Wolfe and Alex Chanin. "Executing compressed programs on an embedded RISC architecture". In: *ACM SIGMICRO Newsletter* 23.1-2 (Dec. 1992), pp. 81–91. DOI: 10.1145/144965.145003.

[Web+11] Geoffrey I. Webb, Eamonn Keogh, Risto Miikkulainen, and Michele Sebag. "Naïve Bayes". In: *Encyclopedia of Machine Learning*. Springer US, 2011, pp. 713–714. DOI: 10.1007/978-0-387-30164-8_576.

[Wen+08] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. "Measurement-Based Timing Analysis". In: *Communications in Computer and Information Science*. Springer Berlin Heidelberg, 2008, pp. 430–444. DOI: 10.1007/978-3-540-88479-8_30.

[Wil+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. "The worst-case execution-time problem—overview of methods and survey of tools". In: *ACM Transactions on Embedded Computing Systems* 7.3 (Apr. 2008), pp. 1–53. DOI: 10.1145/1347375.1347389.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann Publ Inc, May 1, 1999. ISBN: 1558605703.

[WTL17] Dongshu Wang, Dapei Tan, and Lei Liu. "Particle swarm optimization algorithm: an overview". In: *Soft Computing* 22.2 (Jan. 2017), pp. 387–408. DOI: 10.1007/s00500-016-2474-6.

[Wö+15] Leonard Wörteler, Michael Grossniklaus, Christian Grün, and Marc H. Scholl. "Function inlining in XQuery 3.0 optimization". In: *15th Symposium on Database Programming Languages (DBPL)*. ACM, Oct. 2015. DOI: 10.1145/2815072.2815079.

[Xu+19] Donna Xu, Yaxin Shi, Ivor W. Tsang, Yew-Soon Ong, Chen Gong, and Xiaobo Shen. "Survey on Multi-Output Learning". In: *IEEE Transactions on Neural Networks and Learning Systems* (2019), pp. 1–21. DOI: 10.1109/tnnls.2019.2945133.

[XWL03] Yuan Xie, Wayne Wolf, and Haris Lekatsas. "Profile-driven selective code compression [embedded systems]". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2003. DOI: 10.1109/date.2003.1253652.

[Yan12] Xin-She Yang. "Flower Pollination Algorithm for Global Optimization". In: *Unconventional Computation and Natural Computation*. Springer Berlin Heidelberg, 2012, pp. 240–249. DOI: 10.1007/978-3-642-32894-7_27.

[YYN08] Yeboon Yun, Min Yoon, and Hirotaka Nakayama. "Multiobjective optimization based on meta-modeling by using support vector regression". In: *Optimization and Engineering* 10.2 (Nov. 2008), pp. 167–181. DOI: 10.1007/s11081-008-9063-1.

[Zha+10] Qingfu Zhang, Wudong Liu, Edward Tsang, and Botond Virginas. "Expensive Multiobjective Optimization by MOEA/D With Gaussian Process Model". In: *IEEE Transactions on Evolutionary Computation* 14.3 (June 2010), pp. 456–474. DOI: 10.1109/tevc.2009.2033671.

[Zit+03] Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M. Fonseca, and Viviane G. da Fonseca. "Performance assessment of multiobjective optimizers: an analysis and review". In: *IEEE Transactions on Evolutionary Computation* 7.2 (2003), pp. 117–132. DOI: 10.1109/tevc.2003.810758.

[Zit99] Eckart Zitzler. "Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications". PhD thesis. Swiss Federal Institute of Technology Zurich, 1999.

[ZK04] Eckart Zitzler and Simon Künzli. "Indicator-Based Selection in Multiobjective Search". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 832–842. DOI: 10.1007/978-3-540-30217-9_84.

[ZL07] Qingfu Zhang and Hui Li. "MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition". In: *IEEE Transactions on Evolutionary Computation* 11.6 (Dec. 2007), pp. 712–731. DOI: 10.1109/tevc.2007.892759.

[ZL77] Jacob Ziv and Abraham Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (May 1977), pp. 337–343. DOI: 10.1109/tit.1977.1055714.

[ZLT01] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*. Research rep. Eidgenössische Technische Hochschule Zürich (ETH), Institut für Technische Informatik und Kommunikationsnetze (TIK), 2001. URL: https://www.research-collection.ethz.ch/handle/20.500.11850/145755.

[ZT98a] Eckart Zitzler and Lothar Thiele. *An Evolutionary Algorithm for Multiobjective Optimization: The Strength Pareto Approach*. Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), May 1998.

[ZT98b] Eckart Zitzler and Lothar Thiele. "Multiobjective optimization using evolutionary algorithms — A comparative case study". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998, pp. 292–301. DOI: 10.1007/bfb0056872.

[ZY15] Rui Zhang and Enjian Yao. "Electric vehicles' energy consumption estimation with real driving condition data". In: *Transportation Research Part D: Transport and Environment* 41 (Dec. 2015), pp. 177–187. DOI: 10.1016/j.trd.2015.10.010.

[ZZG09] Hong-Zhen Zheng, Xiao-Dong Zhang, and Hao-Yan Guo. "Using SVM to Learn the Efficient Set in Multiple Objective Discrete Optimization". In: *6th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*. IEEE, 2009. DOI: 10.1109/fskd.2009.15.

Acronyms and Notational Conventions

Acronyms

aiT	Static WCET Analyser.
BCET	Best-Case Execution Time.
BGDE3	Binary GDE3.
CFG	Control Flow Graph.
CPU	Central Processing Unit.
DE	Differential Evolution.
EnergyAnalyser	Static Energy Consumption Analyser.
EVT	Extreme Value Theory.
FPA	Flower Pollination Algorithm.
GDE	Generalized Differential Evolution.
GDE2	The Second Version of GDE.
GDE3	The Third Version of GDE.
IBEA	Indicator-Based Evolutionary Algorithm.
ILP	Integer Linear Programming.
IPET	Implicit Path Enumeration Technique.
ISA	Instruction Set Architecture.
MAE	Mean Absolute Error.
MBDE	Modified Binary Differential Evolution.
MBPOA	Multiobjective Binary Probability Optimization Algorithm.
MBPTA	Measurement-Based Probabilistic Timing Analysis.
MOGA	Multi-Objective Genetic Algorithm.
NSGA	Nondominated Sorting Genetic Algorithm.
PMC	Performance Monitoring Counters.
RBF	Radial Basis Function.
RISC	Reduced Instruction Set Computer.
SOC	System on a Chip.
SPEA	Strength Pareto Evolutionary Algorithm.
SPM	Scratchpad Memory.
SPTA	Static Probabilistic Timing Analysis.

SRA	Static Resource Analysis.
SVC	Support Vector Classifier.
VEDE	Vector Evaluated Differential Evolution.
VEGA	Vector Evaluated Genetic Algorithm.
WCC	WCET-Aware C Compiler.
WCEP	Worst-Case Execution Path.
WCET	Worst-Case Execution Time.

Commonly Used Symbols

$\mathbb{B}$ Space consisting of binary vectors.
X Search (decision) space.
Y Objective space.
n Dimension of X.
m Dimension of Y.
$\mathbf{f}$ Objective vector function.
f Objective scalar function.
$|\cdot|$ Size of a set.
$i = \overline{1, m}$ Set $\{\forall i : i \in \{1, 2, \cdots, m\}\}$.
S_{limit} Upper bound of MAE.

Compression

CS Code size.
DS Data size.
PS Program size.
G ILP objective function.
R Compression ratio.
N Number of compression candidates.
K Total number of functions.
U Set of compression candidates.

Evolutionary Algorithm

$\prec$ Dominance operator.
$\preceq$ Weak dominance operator.
NR Nondominance ratio.
C Coverage.
b Bandwidth parameter.
F Scaling factor.
CR Crossover parameter.
NP Population size.

Prediction of Objectives

p	Probability distribution function.
$MBPOA_{AbsInt}$	MBPOA with WCET and energy consumption estimated by aiT and EnergyAnalyser.
$MBPOA_{LogReg}$	MBPOA with WCET and energy consumption predicted by logisitic regression.
$MBPOA_{DTree}$	MBPOA with WCET and energy consumption predicted by decision tree classifier.
$MBPOA_{ABoost}$	MBPOA with WCET and energy consumption predicted by AdaBoost classifier based on decision trees.

L	Inverse of regularization strength in logistic regression.
ρ	Elastic-Net mixing parameter in logistic regression.

Search Space Reduction

$DIFF$	Difference introduced in Algorithm 4.
S_{red}	Reduction score.
$Outliers$	Reduction stratetegy: keep outliers.
25%	Reduction stratetegy: keep 25% of features.
50%	Reduction stratetegy: keep 50% of features.
75%	Reduction stratetegy: keep 75% of features.
$GTH0$	Reduction stratetegy: keep features with $DIFF > 0$.

Reduced Search Space and Prediction Model

$MBPOA_{orig}$	MBPOA executed on the original search space and WCET and energy consumption estimated by aiT and EnergyAnalyser.
$MBPOA_{pred}$	MBPOA executed on the original search space and WCET and energy consumption predicted by decision tree classifier fitted on the original search space.
$MBPOA_{red}$	MBPOA executed on a reduced search space and WCET and energy consumption estimated by aiT and EnergyAnalyser.
$MBPOA_{red+pred}$	MBPOA executed on a reduced search space and WCET and energy consumption predicted by decision tree classifier fitted on the reduced search space.

List of Figures

2.1	An exemplary distribution of execution time	8
2.2	WCET and energy analyses framework	13
2.3	Example of a control flow graph	14
2.4	Example of a WCEP switch	15
2.5	von Neumann architecture with a scratchpad memory	18
2.6	Harvard architecture with scratchpad memories	19
3.1	Simplified structure of the WCET-Aware C Compiler (WCC)	22
4.1	Exemplary Pareto optimal front for a minimization problem with two contradicting objectives	29
4.2	Exemplary true Pareto front and approximated Pareto front for a minimization problem with two contradicting objectives	31
5.1	Workflow of compile-time code compression and runtime decompression	46
5.2	Compression of a function at compile time	53
5.3	Compiler-based preparation phase for runtime decompression	54
5.4	Statistics for functions subject to compile-time compression for benchmarks with at least one function selected as a compression candidate after the prephase	59
5.5	Statistics for the program size of benchmarks after compression	60
5.6	Statistics for WCET when decompressing compressed functions to the scratchpad memory	61
5.7	Statistics for WCET when decompressing compressed functions to the main memory	63
6.1	Example of a cuboid to calculate the crowding distance of an individual in a 2-dimensional objective space	74
6.2	Example of function inlining	79
6.3	Program from Figure 6.2(b) with applied redundant path elimination and constant propagation	80
6.4	Dependence between WCET and energy consumption while performing function inlining	82
6.5	Runtime required to estimate WCET and energy consumption 1,500 times by using the static analysers aiT and EnergyAnalyser	86

6.6 The mean and 95 % confidence interval of nondominance ratio and coverage over all considered benchmarks when solving the inlining problem with MBPOA. 87
6.7 The mean and 95 % confidence interval for nondominance ratio and coverage over all considered benchmarks when solving the inlining problem with BGDE3 88
6.8 Average runtime of BGDE3 and MBPOA 89
6.9 Quality indicators of BGDE3 and MBPOA with fixed control parameters 91
6.10 Solutions returned by BGDE3 and MBPOA for the benchmark `3mm` in 10 runs 92
6.11 Solutions returned by BGDE3 and MBPOA for the benchmark `cjpeg_wrbmp` in 10 runs 93

7.1 Unique objectives' values while performing function inlining for three exemplary benchmarks 100
7.2 Example of K-D tree 104
7.3 Example of ball tree 106
7.4 Example of support vector classifier 107
7.5 Example of kernel trick 108
7.6 Example of a prior distribution, a posterior distribution, and a likelihood function in the Bayes rule 111
7.7 Example of decision tree 112
7.8 Example of AdaBoost classifier with decision trees 113
7.9 New components of WCC 119
7.10 Average MAE of classifiers for all considered benchmarks 124
7.11 Average MAE from Figure 7.10 with the range of the y-axis changed to $[0, 15]$ 126
7.12 MBPOA runtimes 129
7.13 Runtimes of $\mathsf{MBPOA}_{\mathsf{LogReg}}$, $\mathsf{MBPOA}_{\mathsf{DTree}}$, and $\mathsf{MBPOA}_{\mathsf{ABoost}}$. 131
7.14 Quality indicators of Pareto fronts returned by MBPOA with estimated and predicted objectives 132
7.15 Pareto fronts returned by MBPOA with estimated and predicted objectives for `3mm` 134
7.16 Pareto fronts returned by MBPOA with estimated and predicted objectives for `cjpeg_wrbmp` 136

8.1 The number of unique WCET and energy consumption values among 100 random search vectors while performing function inlining 141
8.2 Example of WCEP switch due to function inlining 144

8.3 Example of the strategies from Table 8.2 for the benchmark md5 and objective WCET . . . 147
8.4 Average MAE for all considered benchmarks. . . . 152
8.5 Average MAE from Figure 8.4 with the range of the y-axis changed to $[0, 12]$ . . . 153
8.6 Statistics of resulting strategies for WCET . . . 155
8.7 Statistics of resulting strategies for energy consumption . . . 156
8.8 The number of selected important features . . . 157
8.9 Average dimensions of the reduced search spaces . . . 158
8.10 MAE for WCET and energy consumption after reducing search spaces . . . 159
8.11 Quality indicators of MBPOA executed on original and reduced search spaces . . . 160
8.12 Quality indicators of MBPOA with reduced search spaces and MBPOA based on predictions . . . 163
8.13 Solutions returned by MBPOA when solving the function inlining problem on the original and reduced search spaces for 3mm . . . 164
8.14 Solutions returned by MBPOA when solving the function inlining problem on the reduced and original search spaces for cjpeg_wrbmp 165
8.15 Runtimes of MBPOA executed on reduced and original search spaces . . . 166

9.1 Quality indicators of Pareto fronts returned by MBPOA executed on reduced search spaces and with WCET and energy consumption predicted by logistic regression, decision tree, and AdaBoost classifiers . . . 173
9.2 Quality indicators of Pareto fronts returned by MBPOA with configurations from Table 9.1 . . . 176
9.3 Runtimes of MBPOA with configurations from Table 9.1 . . . 177
9.4 MBPOA runtime from Figure 9.3 with the range of the y-axis changed to $[0, 0.3]$ . . . 178
9.5 Training set sizes when fitting decision tree classifier for WCET and energy consumption on the original and reduced search spaces 179
9.6 Solutions returned by MBPOA when solving the function inlining problem on the original and reduced search spaces with and without predictions for 3mm . . . 180
9.7 Solutions returned by MBPOA when solving the function inlining problem on the original and reduced search spaces with and without predictions for cjpeg_wrbmp . . . 181

D.1 Frequency of WCET while performing function inlining . . . 237

D.2 (continued) Frequency of WCET while performing function inlining 238
D.3 (continued) Frequency of WCET while performing function inlining 239
D.4 (continued) Frequency of WCET while performing function inlining 240
D.5 Frequency of energy consumption while performing function inlining 241
D.6 (continued) Frequency of energy consumption while performing function inlining 242
D.7 (continued) Frequency of energy consumption while performing function inlining 243
D.8 (continued) Frequency of energy consumption while performing function inlining 244

List of Tables

2.1 Counters of energy model 16

5.1 Server specifications 57
5.2 Statistics for compression of the benchmark *jetbench1* 62
5.3 Compression runtime 64

7.1 Maximum training set sizes and the corresponding number of benchmarks for which the sizes were used by Algorithm 3 127

8.1 WCET changes due to function inlining applied to the program from Figure 8.2 145
8.2 Strategies for the procedure GetLowerBound from Algorithm 4 . 146

9.1 MBPOA configurations 175
9.2 Coverage for the benchmark `cjpeg_wrbmp` 182

A.1 Compression: benchmarks 225

B.1 Function inlining: benchmarks 231

E.1 Training set size and average MAE for logistic regression, decision tree, and AdaBoost classifiers 245

Appendices

A Compression: Benchmarks

Table A.1 presents benchmarks to evaluate a compression technique described in Chapter 5. The table lists benchmark suites, benchmarks, and the total number of functions in a benchmark.

Table A.1: Compression: benchmarks.

Benchmark suite	Benchmark	Total number of functions
DSPstone	adpcm_g721_board_test	20
	adpcm_g721_verify	20
	complex_multiply_fixed	2
	complex_multiply_float	2
	complex_update_fixed	2
	complex_update_float	2
	convolution_fixed	2
	convolution_float	2
	dot_product_fixed	2
	dot_product_float	2
	fft_1024_13	7
	fft_1024_7	7
	fft_16_13	7
	fft_16_7	7
	fir2dim_fixed	2
	fir2dim_float	2
	fir_fixed	2
	fir_float	2
	iir_biquad_N_sections_fixed	2
	iir_biquad_N_sections_float	2
	iir_biquad_one_section_fixed	2
	iir_biquad_one_section_float	2
	lms_fixed	2
	lms_float	2
	matrix1_fixed	2
	matrix1_float	2

Continued on next page

Benchmark suite	Benchmark	Total number of functions
	matrix1x3_fixed	1
	matrix1x3_float	2
	matrix2_fixed	2
	matrix2_float	2
	n_complex_updates_fixed	2
	n_complex_updates_float	2
	n_real_updates_fixed	2
	n_real_updates_float	2
	real_update_fixed	2
	real_update_float	2
	startup_fixed	2
JETBENCH	jetbench1	18
MRTC	adpcm	17
	bs	2
	bsort100	3
	cnt	6
	compress	9
	cover	4
	crc	3
	duff	3
	edn	9
	expint	3
	fac	2
	fdct	2
	fft1	6
	fibcall	2
	fir	2
	insertsort	1
	janne_complex	2
	jfdctint	2
	lcdnum	2
	lms	8
	ludcmp	3
	matmult	6
	minver	4
	ndes	5
	ns	2
	nsichneu	1

Continued on next page

Benchmark suite	Benchmark	Total number of functions
	petrinet	1
	prime	5
	qsort-exam	2
	qurt	4
	recursion	4
	select	2
	sqrt	3
	st	10
	statemate	8
	ud	2
MediaBench	cjpeg_jpeg6b_transupp	6
	cjpeg_jpeg6b_wrbmp	4
	gsm	52
	gsm_decode	34
	h264dec_ldecode_block	4
	h264dec_ldecode_macroblock	2
MiBench	basicmath_small	17
	bitcount	10
	dijkstra	5
	sha	12
PolyBench	adi	4
	fdtd-2d	4
	fdtd-apml	4
	floyd-warshall	4
	jacobi-1d-imper	4
	jacobi-2d-imper	4
	seidel-2d	4
StreamIt	filterbank	3
UTDSP	adpcm	10
	compress	11
	edge_detect	3
	fft_1024	2
	fft_256	2
	fir_256_64	2
	fir_32_1	2
	g721.marcuslee_decoder	1
	g721.marcuslee_encoder	1
	histogram	1

Continued on next page

Benchmark suite	Benchmark	Total number of functions
	iir_1_1	2
	iir_4_64	2
	latnrm_32_64	2
	latnrm_8_1	2
	lmsfir_32_64	2
	lmsfir_8_1	2
	lpc	7
	mult_10_10	2
	mult_4_4	2
	qmf_receive	2
	qmf_transmit	2
	spectral	10
	trellis	11
	v32.modem_achop	2
	v32.modem_bencode	8
	v32.modem_cnoise	8
	v32.modem_ddecode	8
	v32.modem_eglue	2
linear-algebra	bicg	4
	doitgen	4
	durbin	4
	dynprog	4
	gemm	4
	gemver	4
	gesummv	4
	gramschmidt	6
	lu	4
	mvt	4
	symm	4
	syr2k	4
	syrk	4
	trisolv	4
misc	codecs_codrle1	6
	codecs_dcodhuff	11
	codecs_dcodrle1	7
	g721_encode	26
	g723_encode	26
	hamming_window	1

Continued on next page

Benchmark suite	Benchmark	Total number of functions
	pm	14
	searchmultiarray	2
	selection_sort	2

B Function Inlining: Benchmarks

Table B.1 presents benchmarks used to evaluate approaches described in Chapters 6–9. The table lists benchmark suites, benchmarks, and their dimensions of the search spaces for a function inlining problem formulated in Section 6.3.

Table B.1: Function inlining: benchmarks.

Benchmark suite	Benchmark	Dimension of the search space
DSPstone	adpcm_board	6
	adpcm_verify	6
	fft_1024_13	88
	fft_1024_7	88
	fft_16_13	88
	fft_16_7	88
	fir_float	88
	iir_biq_1s_flt	89
JETBENCH	jetbench1	87
	jetbench2	87
	jetbench3	87
MRTC	adpcm	13
	cnt	89
	expint	87
	fft1	91
	lms	88
	ludcmp	87
	qurt	88
	sqrt	87
	st	90
MediaBench	cjpeg_wrbmp	10
	epic	87
	gsm_encode	6
NetBench	md5	7
PolyBench	3mm	87
	atax	87

Continued on next page

Benchmark suite	Benchmark	Dimension of the search space
	bicg	87
	cholesky	88
	correlation	88
	covariance	87
	doitgen	87
	durbin	87
	fdtd-2d	87
	floyd-warshall	87
	gemm	87
	gemver	87
	gesummv	87
	gramschmidt	87
	jacobi-1d-imper	87
	jacobi-2d-imper	87
	lu	87
	ludcmp	87
	mvt	87
	seidel-2d	87
	symm	87
	syr2k	87
	syrk	87
	trisolv	87
	trmm	87
StreamIt	filterbank	86
	fmref	91
UTDSP	lpc	88
	spectral	87
	trellis	94
	v32mod_cnoise	90
misc	codecs_codhuff	12
	codecs_codrle1	24
	codecs_dcodhuff	9
	codecs_dcodrle1	6
	g721_encode	10
	g723_encode	10
	pm	95

C The Third Version of the Generalized Differential Evolution

Algorithm 6 presents the procedure of The Third Version of the GDE algorithm (GDE3) that consists of three main parts:

1. a new decision vector is generated at Lines 16–23;
2. an old decision vector, the new decision vector, or both of them are selected to survive in the next generation at Lines 25–32;
3. a population is reduced at Lines 34–39 if the size of the population was increased during the selection phase.

During algorithm execution, two user-defined parameters remain fixed:

- *crossover parameter* CR represents the probability of assigning a coordinate of a trial vector $\mathbf{u}_{i,g}$ to a linear combination of the old vectors' coordinates at Line 21 or to the coordinate of the old vector $\mathbf{x}_{i,g}$ at Line 23. The second condition at Line 20 ($j = j_{rand}$) guarantees that at least one coordinate of the trial vector $\mathbf{u}_{i,g}$ differs from the coordinate of the old vector $\mathbf{x}_{i,g}$;
- *scaling factor* F controls robustness and speed when exploring a search space at Line 21. With a lower value of F, the algorithm converges faster to an optimum but may get stuck in a local optimum.

To reduce population size at Lines 34–39 if it exceeds a predefined limit NP, individuals are sorted with respect to rank and crowding distance. The rank of an individual represents the number of individuals in the population dominating the current individual. A lower rank indicates a higher quality of the individual.

Algorithm 7 presents a procedure to assign ranks to individuals of a population. For each individual, the algorithm compares the individual to another one (Line 6) and increases its rank if the current individual is dominated by the other one (Line 7).

Crowding distance estimates the density of surrounding solutions for each individual in a given population. Algorithm 8 presents a procedure to assign

crowding distances to individuals. For each objective, the algorithm sorts individuals by objective values and identifies the maximum and minimum values of the objective (Lines 5–7). Then, for each individual, it increases its crowding distance by a value proportional to the length of the corresponding side of the cuboid formed by the nearest neighbours[1] (Line 11).

[1]Figure 6.1 on Page 74 shows an example of the cuboid.

Algorithm 6 The third version of Generalized Differential Evolution (GDE3). Adapted from Kukkonen and Lampinen [KL05].

1: **Input:** n – search space dimension,
2: G – number of generations, NP - population size,
3: $F \in \mathbb{R}_{\geqslant 0}$ – scaling factor, $CR \in [0, 1]$ – crossover parameter,
4: $\mathbf{x}^{min}, \mathbf{x}^{max}$ - initial bounds,
5: $rand[0, 1]$ – operator returning a random number from $[0, 1]$.
6: **Output:** approximated Pareto front.
7: **Remark:** in index tuple (j, i, g), we denote by j the coordinate of search vector, by i individual, and by g generation.
8:
9: **Initial population initialization:**
10: $\forall i = \overline{1, NP}, j = \overline{1, n} \quad x_{j,i,0} = x_j^{min} + rand[0, 1] \cdot (x_j^{max} - x_j^{min})$
11:
12: **GDE3 execution:**
13: **for** $g = \overline{1, G}$ **do** ▷ For each generation.
14: **for** $i = \overline{1, NP}$ **do** ▷ For each individual from generation g.
15:
16: *Crossover and mutation:*
17: Select randomly $r_1, r_2, r_3 \in \{1, 2, \ldots NP\} : r_1 \neq r_2 \neq r_3 \neq i$
18: and $j_{rand} \in \{1, 2, \ldots, n\}$
19: **for** $j = \overline{1, n}$ **do** ▷ For each dimension of the search space.
20: **if** $rand[0, 1) < CR$ OR $j = j_{rand}$ **then**
21: $u_{j,i,g} := x_{j,r_3,g} + F \cdot (x_{j,r_1,g} - x_{j,r_2,g})$
22: **else**
23: $u_{j,i,g} := x_{j,i,g}$
24:
25: *Selection:*
26: **if** $\mathbf{u}_{i,g} \preceq \mathbf{x}_{i,g}$ **then**
27: $\mathbf{x}_{i,g+1} := \mathbf{u}_{i,g}$ ▷ $\mathbf{u}_{i,g}$ weakly dominates $\mathbf{x}_{i,g}$.
28: **else**
29: $\mathbf{x}_{i,g+1} := \mathbf{x}_{i,g}$
30: **if** $\mathbf{x}_{i,g} \not\prec \mathbf{u}_{i,g}$ **then**
31: $m = m + 1$ ▷ $\mathbf{x}_{i,g}$ and $\mathbf{u}_{i,g}$ are incomparable, so
32: $\mathbf{x}_{NP+m,g+1} := \mathbf{u}_{i,g}$ ▷ $\mathbf{u}_{i,g}$ is selected for the next generation.
33:
34: *Reduce the population, if it has m additional points because of Lines 30-32.*
35: Assign ranks to individuals by using Algorithm 7.
36: Assign crowding distances to individuals by using Algorithm 8.
37: Sort individuals by rank in ascending order.
38: For each rank, sort individuals by crowding distance in descending order.
39: Remove the last m individuals.
40: **return** $\{\mathbf{x}_{i,G}\}_{i=\overline{1,NP}}$.

Algorithm 7 Rank assignment. Adapted from Deb et al. [Deb+02a].

1: **Input:** population P.
2: **Output:** rank for each individual of P.
3: **Initialization:** $\forall p \in P \quad \mathrm{Rank}_p := 0$.
4: **for** $p \in P$ **do**
5:     **for** $q \in P$ **do**
6:         **if** $q \prec p$ **then**
7:             $\mathrm{Rank}_p = \mathrm{Rank}_p + 1$
    return $\{\mathrm{Rank}_p\}_{p \in P}$.

Algorithm 8 Crowding distance assignment. Adapted from Deb et al. [Deb+02a].

1: **Input:** population P, number of objectives m.
2: **Output:** crowding distance for each individual of P.
3: **Initialization:** $NP := |P|$ and $\forall i = \overline{1, NP} \quad \mathrm{CDist}_i := 0$.
4: **for** $j = \overline{1, m}$ **do**
5:     Sort P by the values of the j-th objective.
6:     $f_j^{max} := \max_{p \in P} p[j]$ ▷ $p[j]$ is the j-th objective of the individual p.
7:     $f_j^{min} := \min_{p \in P} p[j]$
8:     **for** $i = \overline{2, (NP-1)}$ **do**
9:         $p := P_{i+1}$ ▷ Individual with index $(i+1)$.
10:         $q := P_{i-1}$ ▷ Individual with index $(i-1)$.
11:         $\mathrm{CDist}_i = \mathrm{CDist}_i + (p[j] - q[j]) / (f_j^{max} - f_j^{min})$
    return $\{\mathrm{CDist}_1, \mathrm{CDist}_2, \ldots, \mathrm{CDist}_{NP}\}$.

D Unique WCET and Energy Consumption Values for Function Inlining

The following figures present the frequency of unique WCET and energy consumption values observed for 100 samples randomly generated for a function inlining problem formulated in Section 6.3. The x-axis presents the frequency of the objectives, the y-axis presents relative WCET and energy consumption computed by the static analysers aiT [Abs22] and EnergyAnalyser [Tea]. The objectives were normalized to lie in the interval $[1,2]$ for the sake of legibility. The results are presented for all considered benchmarks listed in Appendix B; the dimension of the search space is specified next to a benchmark name (dim).

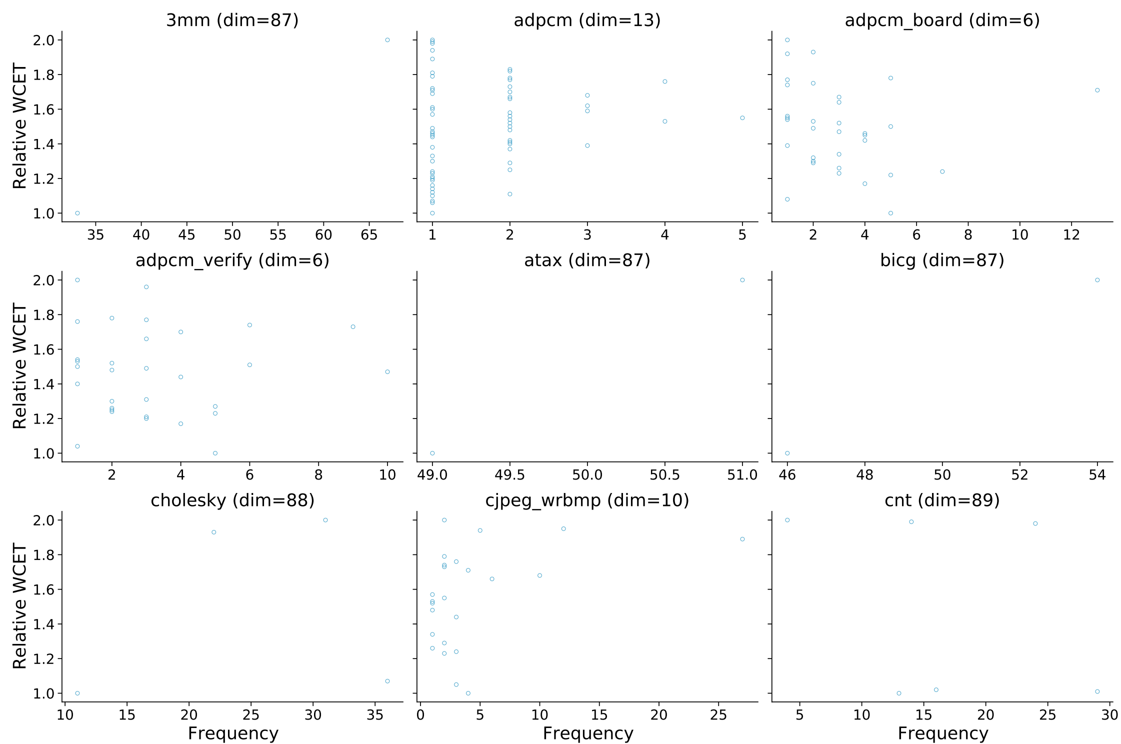


Figure D.1: Frequency of WCET while performing function inlining.

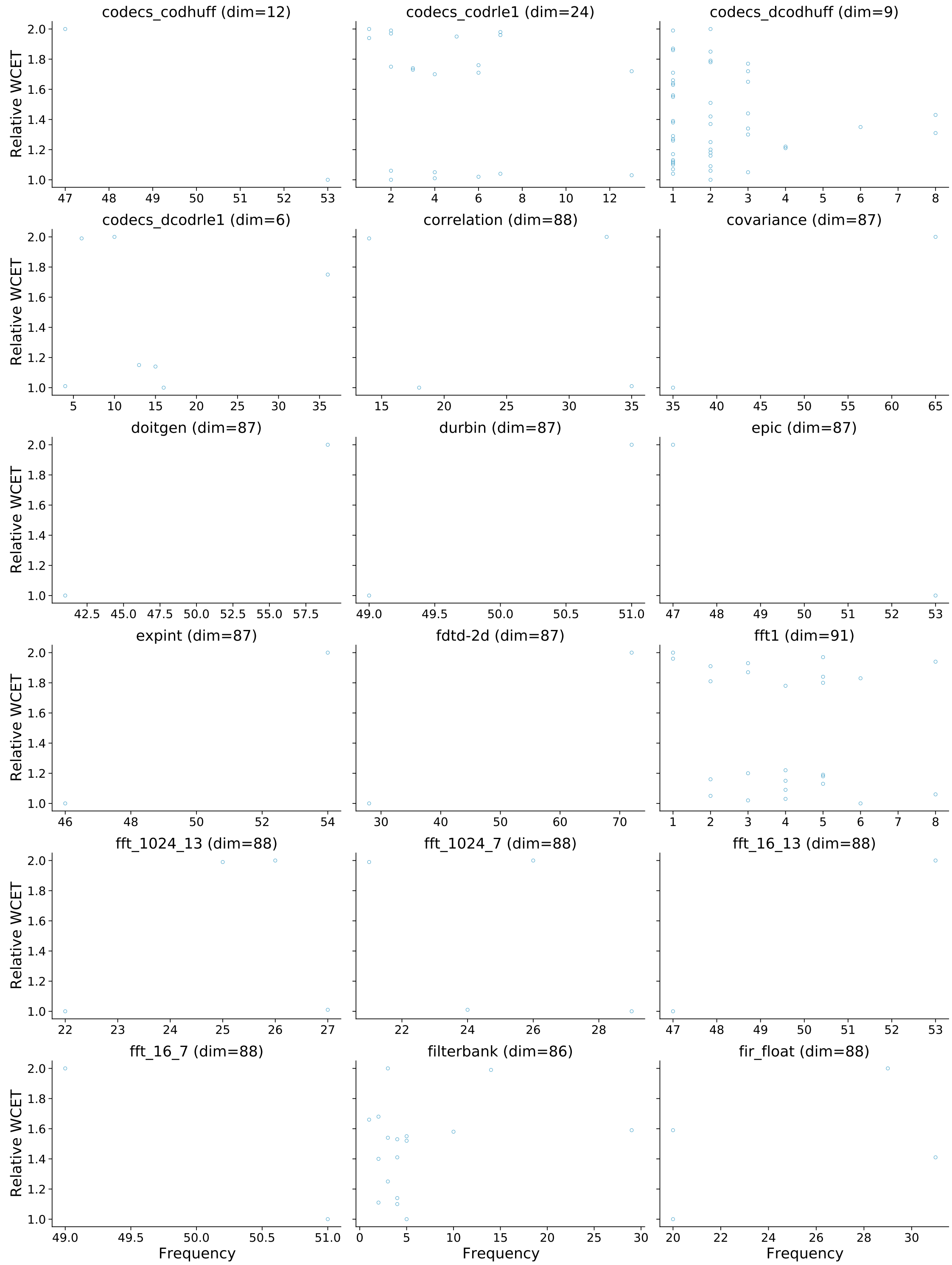


Figure D.2: (continued) Frequency of WCET while performing function inlining.

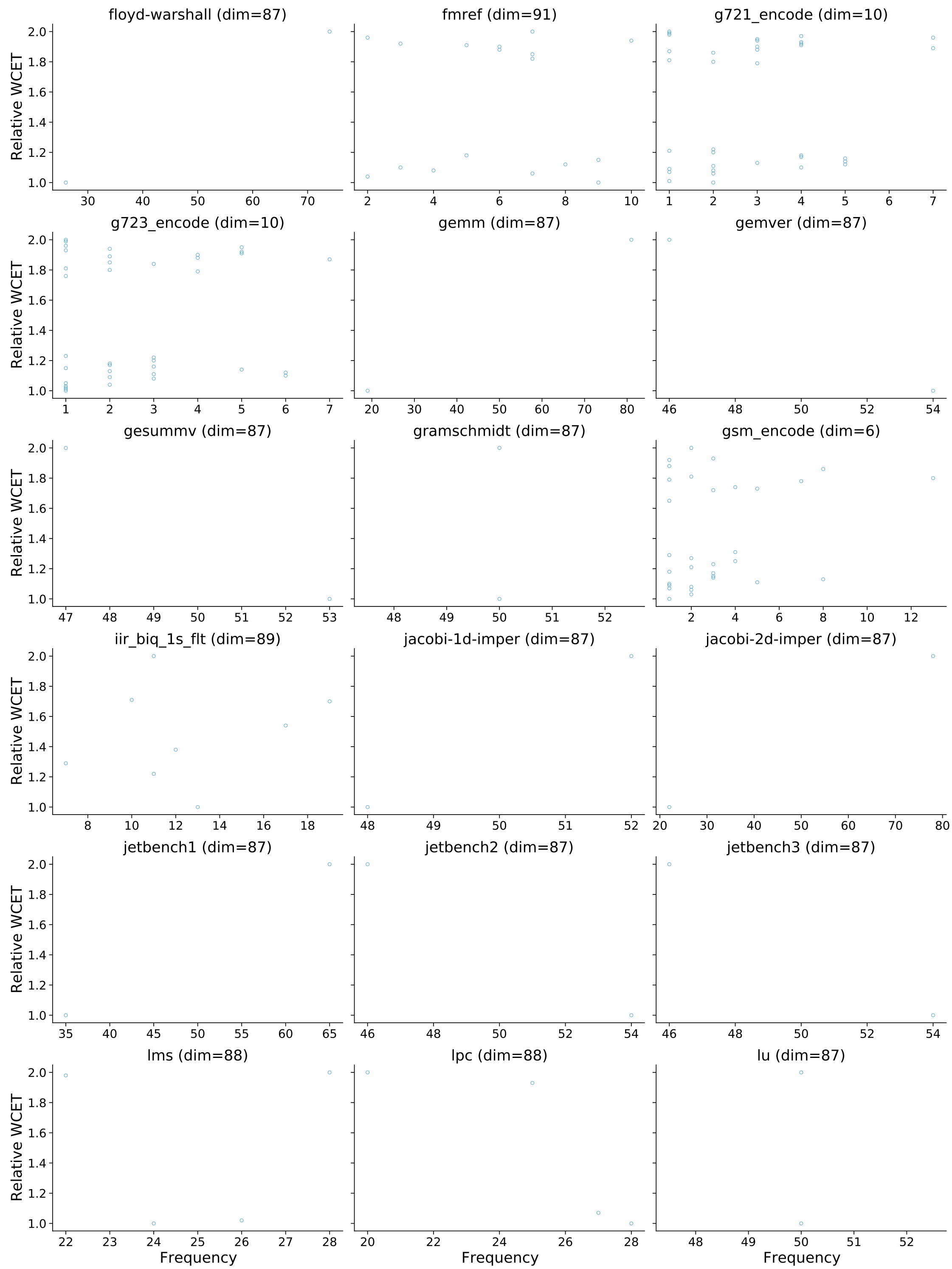


Figure D.3: (continued) Frequency of WCET while performing function inlining.

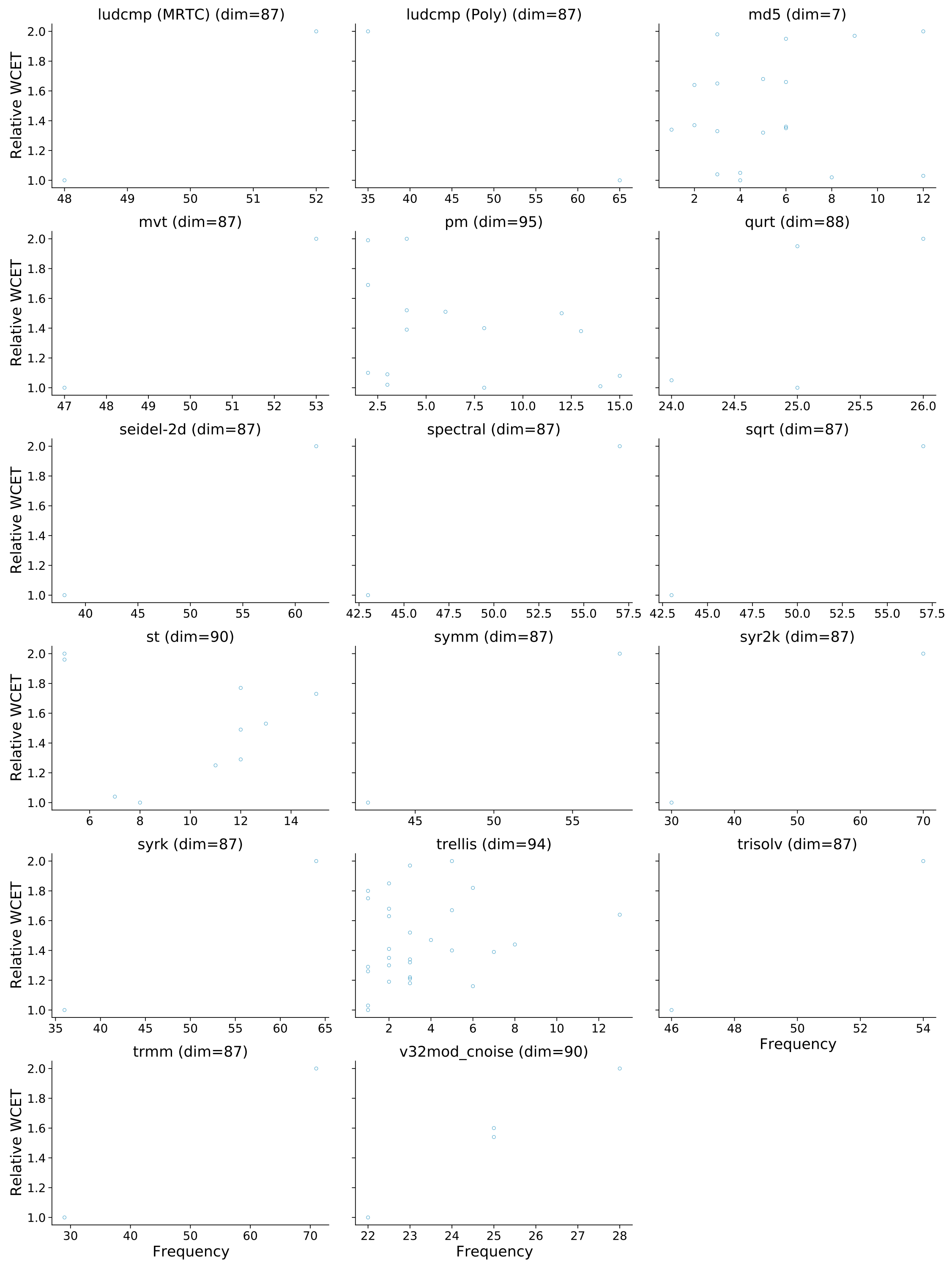


Figure D.4: (continued) Frequency of WCET while performing function inlining.

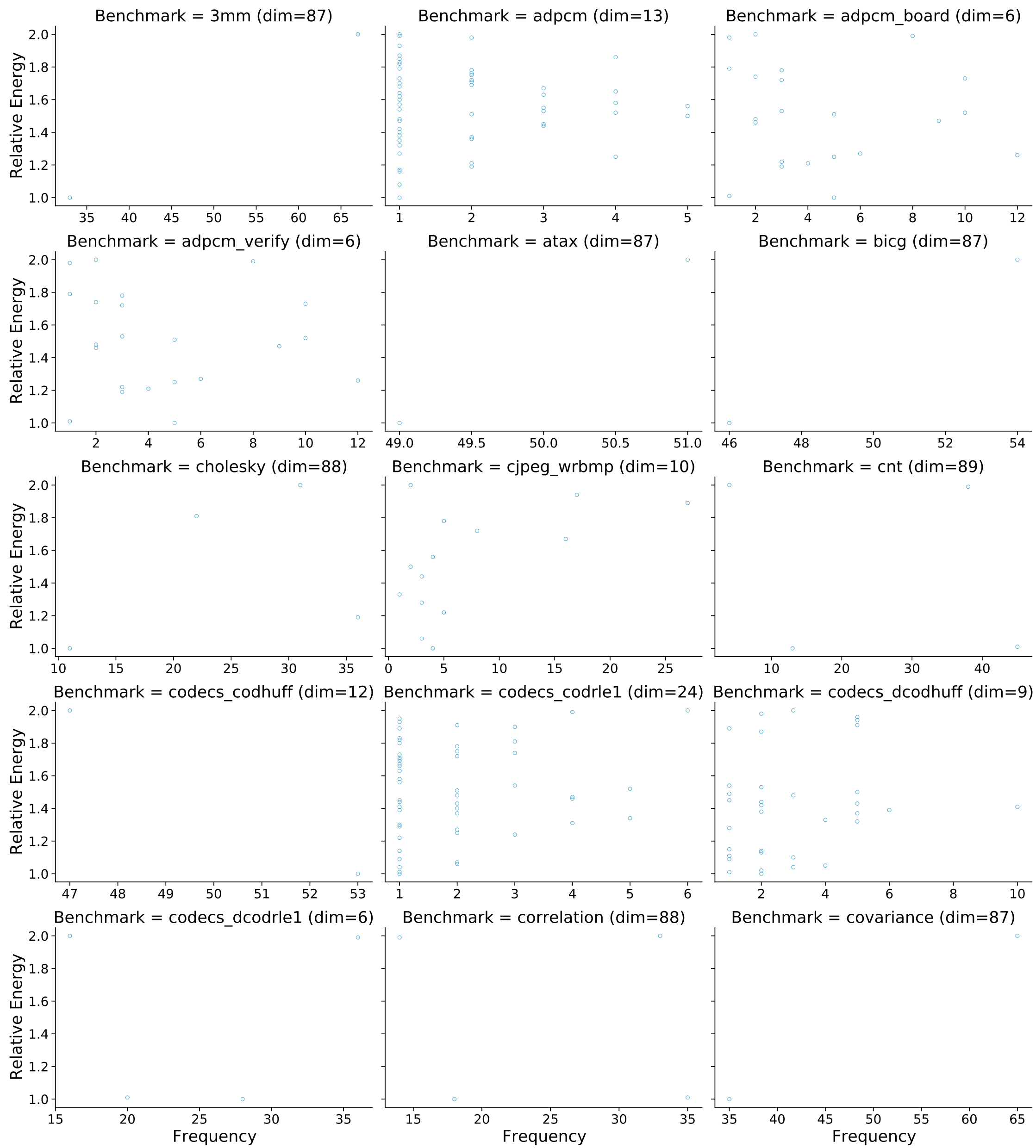


Figure D.5: Frequency of energy consumption while performing function inlining.

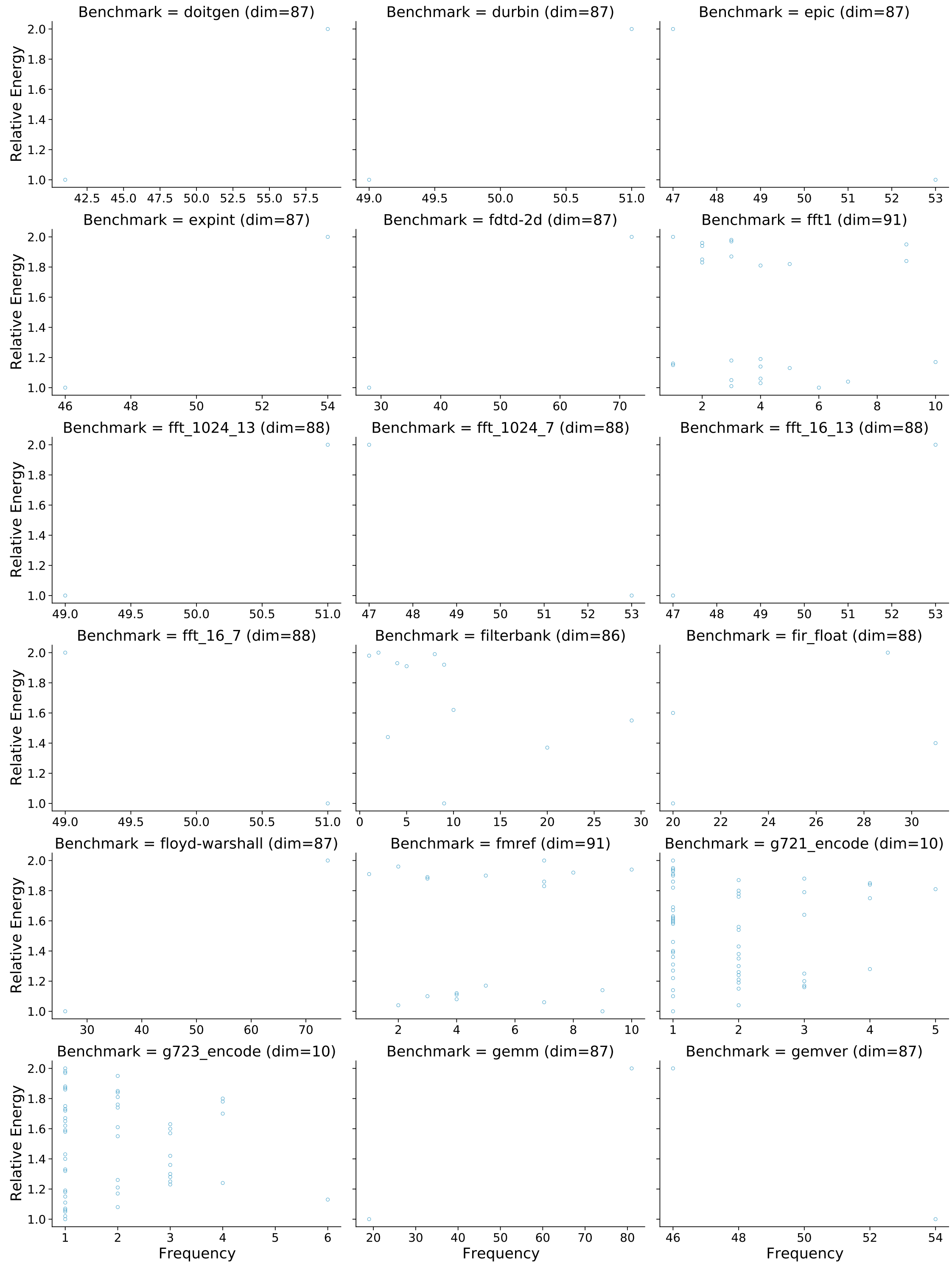


Figure D.6: (continued) Frequency of energy consumption while performing function inlining.

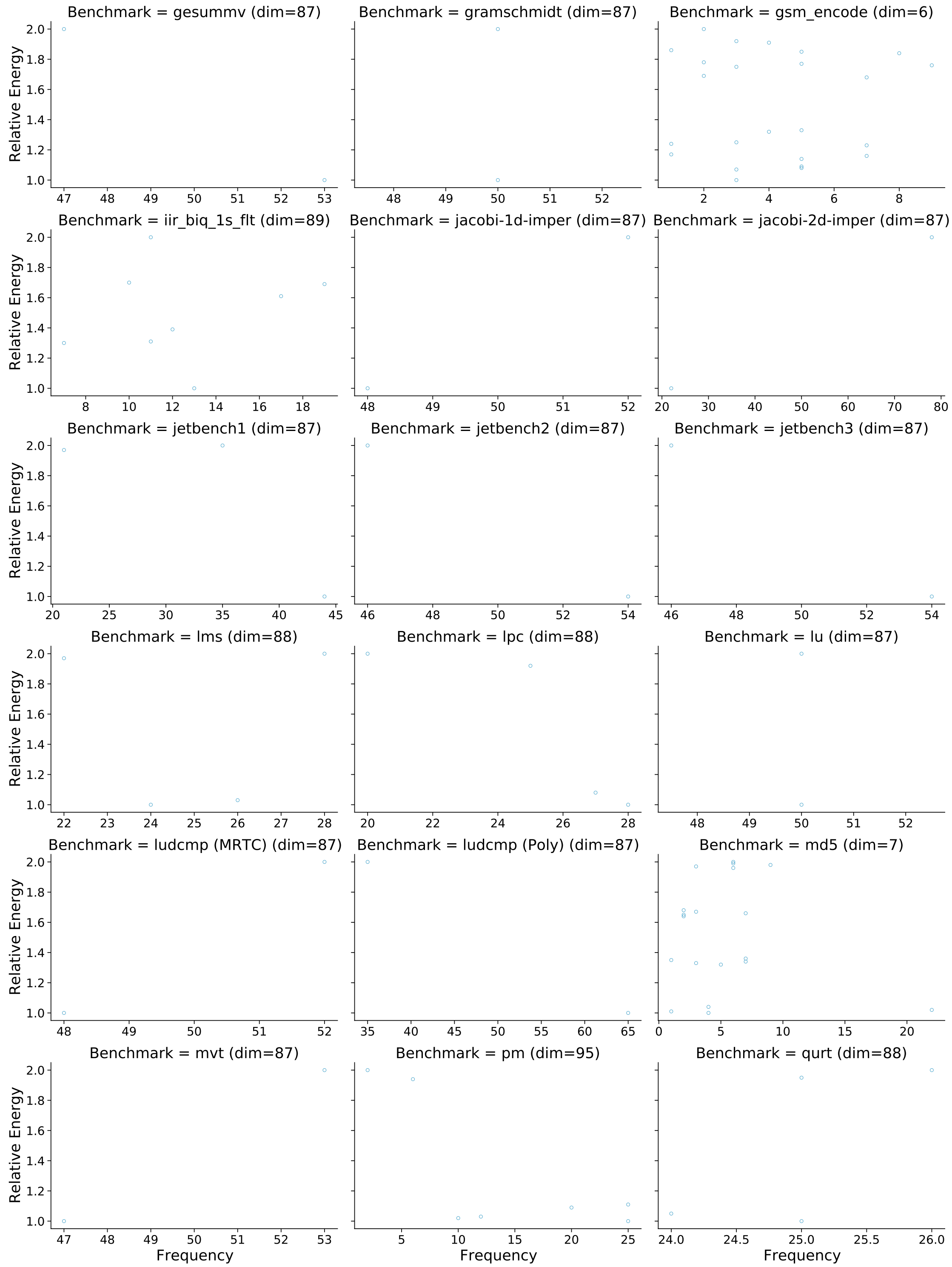


Figure D.7: (continued) Frequency of energy consumption while performing function inlining.

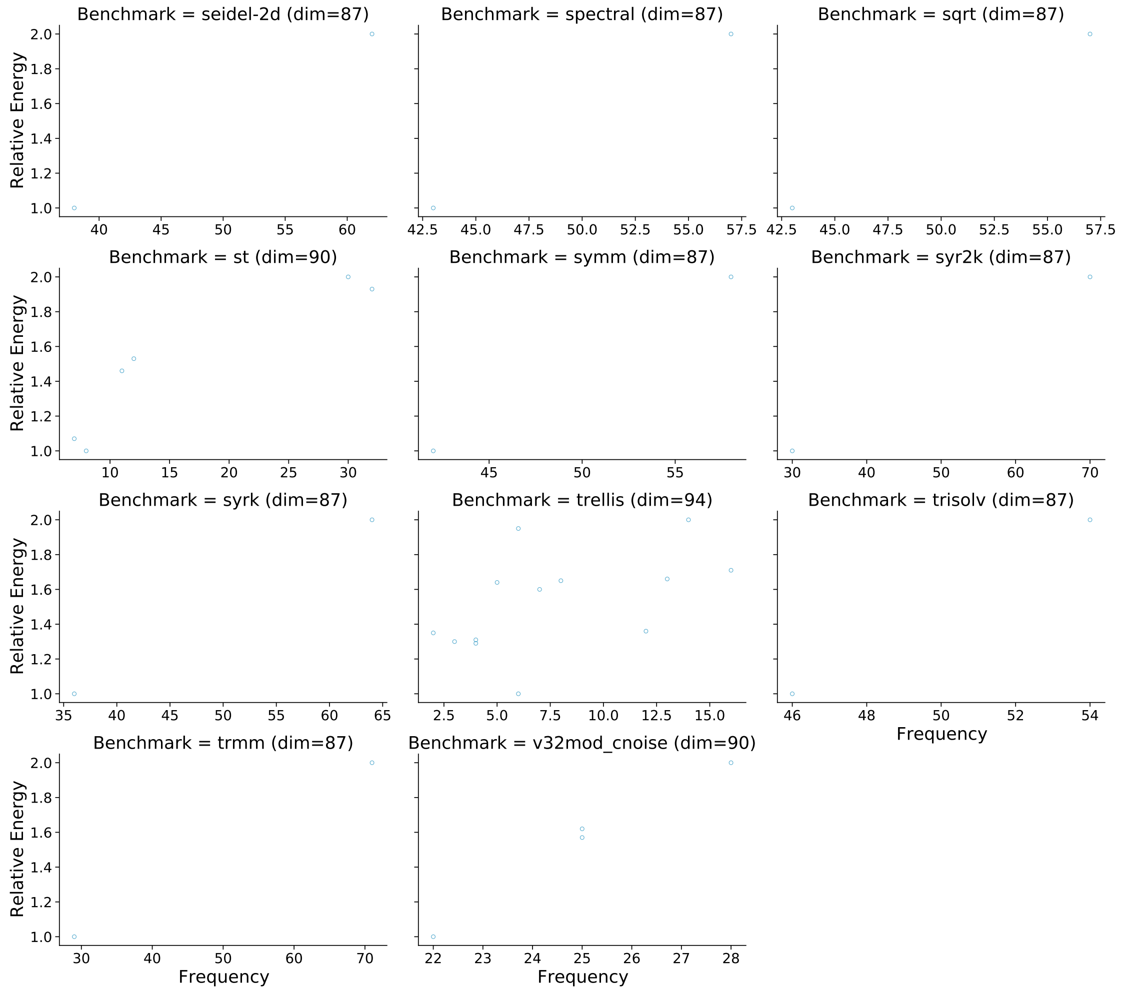


Figure D.8: (continued) Frequency of energy consumption while performing function inlining.

E Scores of Logistic Regression, Decision Tree, and AdaBoost

Table E.1 presents the maximum size of training sets required to build prediction models by using logistic regression, decision tree, and AdaBoost classifiers in Algorithm 3 and the average MAE of the models. Algorithm 3 was repeated 10 times for each benchmark due to randomly generated training sets.

Table E.1: Final training set size and average MAE for all benchmarks.

Benchmark suite	Benchmark	Classifier	Objective	Training set size	Average MAE
DSPstone	adpcm_board	AdaBoost	Energy cons.	30	0.03
			WCET	60	0.03
		Dec. tree	Energy cons.	30	0.05
			WCET	60	0.03
		Log. regr.	Energy cons.	30	0.04
			WCET	60	0.04
	adpcm_verify	AdaBoost	Energy cons.	30	0.05
			WCET	60	0.02
		Dec. tree	Energy cons.	30	0.06
			WCET	60	0.02
		Log. regr.	Energy cons.	30	0.04
			WCET	60	0.04
	fft_1024_13	AdaBoost	Energy cons.	40	0.00
			WCET	40	0.00
		Dec. tree	Energy cons.	40	0.00
			WCET	40	0.00
		Log. regr.	Energy cons.	40	0.00
			WCET	40	0.00
	fft_1024_7	AdaBoost	Energy cons.	40	0.00
			WCET	40	0.00
		Dec. tree	Energy cons.	40	0.00
			WCET	40	0.00

Continued on next page

Benchmark suite	Benchmark	Classifier	Objective	Training set size	Average MAE
		Log. regr.	Energy cons.	40	0.00
			WCET	40	0.00
	fft_16_13	AdaBoost	Energy cons.	40	0.00
			WCET	40	0.00
		Dec. tree	Energy cons.	40	0.00
			WCET	40	0.00
		Log. regr.	Energy cons.	40	0.00
			WCET	40	0.02
	fft_16_7	AdaBoost	Energy cons.	40	0.00
			WCET	40	0.00
		Dec. tree	Energy cons.	40	0.00
			WCET	40	0.00
		Log. regr.	Energy cons.	40	0.00
			WCET	40	0.00
	fir_float	AdaBoost	Energy cons.	40	0.00
			WCET	40	0.22
		Dec. tree	Energy cons.	40	0.21
			WCET	40	0.00
		Log. regr.	Energy cons.	40	0.07
			WCET	40	0.93
	iir_biq_1s_flt	AdaBoost	Energy cons.	50	2.61
			WCET	20	3.54
		Dec. tree	Energy cons.	30	4.65
			WCET	60	4.27
		Log. regr.	Energy cons.	60	4.32
			WCET	30	4.73
JETBENCH	jetbench1	AdaBoost	Energy cons.	30	0.00
			WCET	30	1.20
		Dec. tree	Energy cons.	30	0.00
			WCET	30	1.20
		Log. regr.	Energy cons.	30	0.00
			WCET	30	3.61
	jetbench2	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00

Continued on next page

Benchmark suite	Benchmark	Classifier	Objective	Training set size	Average MAE
			WCET	20	0.00
	jetbench3	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
MRTC	cnt	AdaBoost	Energy cons.	20	0.05
			WCET	30	0.01
		Dec. tree	Energy cons.	30	0.00
			WCET	20	0.07
		Log. regr.	Energy cons.	30	1.43
			WCET	20	3.82
	expint	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	fft1	AdaBoost	Energy cons.	60	0.01
			WCET	60	0.01
		Dec. tree	Energy cons.	60	0.01
			WCET	50	0.01
		Log. regr.	Energy cons.	50	0.09
			WCET	40	0.20
	lms	AdaBoost	Energy cons.	40	0.00
			WCET	40	0.00
		Dec. tree	Energy cons.	40	0.00
			WCET	40	0.00
		Log. regr.	Energy cons.	40	0.00
			WCET	40	0.00
	ludcmp	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00

Continued on next page

Benchmark suite	Benchmark	Classifier	Objective	Training set size	Average MAE
	qurt	AdaBoost	Energy cons.	40	0.00
			WCET	40	0.00
		Dec. tree	Energy cons.	40	0.00
			WCET	40	0.00
		Log. regr.	Energy cons.	40	0.00
			WCET	40	0.00
	sqrt	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	st	AdaBoost	Energy cons.	20	1.25
			WCET	30	0.84
		Dec. tree	Energy cons.	30	0.45
			WCET	20	1.99
		Log. regr.	Energy cons.	30	1.34
			WCET	30	1.34
MediaBench	cjpeg_wrbmp	AdaBoost	Energy cons.	50	4.77
			WCET	30	1.67
		Dec. tree	Energy cons.	30	3.66
			WCET	30	3.39
		Log. regr.	Energy cons.	20	4.30
			WCET	70	5.48
	epic	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	gsm_encode	AdaBoost	Energy cons.	60	0.00
			WCET	70	0.00
		Dec. tree	Energy cons.	70	0.00
			WCET	70	0.00
		Log. regr.	Energy cons.	70	0.00
			WCET	70	0.00
PolyBench	3mm	AdaBoost	Energy cons.	20	0.00

Continued on next page

Benchmark suite	Benchmark	Classifier	Objective	Training set size	Average MAE
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	atax	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	bicg	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	cholesky	AdaBoost	Energy cons.	40	0.00
			WCET	40	0.00
		Dec. tree	Energy cons.	40	0.00
			WCET	40	0.00
		Log. regr.	Energy cons.	40	0.00
			WCET	40	0.00
	correlation	AdaBoost	Energy cons.	40	0.00
			WCET	40	0.00
		Dec. tree	Energy cons.	40	0.00
			WCET	40	0.00
		Log. regr.	Energy cons.	40	0.00
			WCET	40	0.00
	covariance	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	doitgen	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00

Continued on next page

Benchmark suite	Benchmark	Classifier	Objective	Training set size	Average MAE
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	durbin	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	fdtd-2d	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	floyd-warshall	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	gemm	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.01
			WCET	20	0.00
	gemver	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	gesummv	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00

Continued on next page

Benchmark suite	Benchmark	Classifier	Objective	Training set size	Average MAE
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	gramschmidt	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	jacobi-1d-imper	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	jacobi-2d-imper	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	lu	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	ludcmp	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	mvt	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00

Continued on next page

Benchmark suite	Benchmark	Classifier	Objective	Training set size	Average MAE
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	seidel-2d	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.04
	symm	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	syr2k	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	syrk	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	trisolv	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	trmm	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00

Continued on next page

Benchmark suite	Benchmark	Classifier	Objective	Training set size	Average MAE
			WCET	20	0.00
StreamIt	filterbank	AdaBoost	Energy cons.	20	0.07
			WCET	30	0.89
		Dec. tree	Energy cons.	20	0.06
			WCET	30	0.73
		Log. regr.	Energy cons.	20	0.04
			WCET	40	0.72
	fmref	AdaBoost	Energy cons.	30	0.33
			WCET	50	0.02
		Dec. tree	Energy cons.	60	0.42
			WCET	60	0.42
		Log. regr.	Energy cons.	40	0.52
			WCET	50	0.17
UTDSP	lpc	AdaBoost	Energy cons.	40	0.00
			WCET	40	0.00
		Dec. tree	Energy cons.	40	0.00
			WCET	40	0.00
		Log. regr.	Energy cons.	40	0.00
			WCET	40	0.00
	spectral	AdaBoost	Energy cons.	20	0.00
			WCET	20	0.00
		Dec. tree	Energy cons.	20	0.00
			WCET	20	0.00
		Log. regr.	Energy cons.	20	0.00
			WCET	20	0.00
	trellis	AdaBoost	Energy cons.	20	0.04
			WCET	40	0.19
		Dec. tree	Energy cons.	20	0.05
			WCET	60	0.20
		Log. regr.	Energy cons.	40	0.08
			WCET	50	0.24
	v32mod_cnoise	AdaBoost	Energy cons.	40	1.84
			WCET	40	0.19
		Dec. tree	Energy cons.	40	0.18
			WCET	40	0.09
		Log. regr.	Energy cons.	40	1.04
			WCET	40	1.05

Continued on next page

Benchmark suite	Benchmark	Classifier	Objective	Training set size	Average MAE
misc	codecs_codhuff	AdaBoost	Energy cons.	70	39.99
			WCET	70	20.00
		Dec. tree	Energy cons.	70	20.00
			WCET	70	39.99
		Log. regr.	Energy cons.	70	29.99
			WCET	70	20.00
	codecs_dcodrle1	AdaBoost	Energy cons.	70	7.31
			WCET	70	8.92
		Dec. tree	Energy cons.	70	7.26
			WCET	70	7.32
		Log. regr.	Energy cons.	70	3.65
			WCET	70	8.54