UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Department of Information Engineering, Computer Science and Mathematics

Ph.D. Program in Information and Communication Technology
Curriculum Emerging computing models, software architectures,and intelligent systems

XXXV Cycle

A Model-Driven Approach for Early Verification and Validation of Embedded Systems

SSD INF/01

Doctoral Program Supervisor
Prof. Vittorio Cortellessa

Advisor
Prof. Vittorio Cortellessa
Dr. Luigi Pomante

Candidate
Vincenzo Stoico

A.Y. 2021-2022

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence

ASIC Application Specific Integrated Circuit

AW Architectural Weight

BAT Bagged Trees

BBB BeagleBone Black

BOT Boosted Trees

BRF Baseline Refactoring Factor

Clon Clone a Node

CoCoME Common Component Modeling Example

CPI Clock cycles Per Instruction

CTL Computational Tree Logic

D&V Design and Verification

DC Digital Camera

DSE Design Space Exploration

DSL Domain Specific Language

EP EPSILON

ESL Electronic System-Level

FMs Formal Methods

FOL First-Order Logic

FT Fine Trees

GCD Greatest Common Divisor

GPP General Purpose Processor

GSPREAD Generalized SPREAD

HLS High Level Synthesis

HV	Hypervolume
ICT	Information and Communication Technologies
IGD$^+$	Inverse Generational Distance plus
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
J4CS	Joule for C Statement
LOC	Source Lines of Code
LQN	Layered Queuing Network
LR	Linear Regression
M2M	Model-To-Model
MAPE	Mean Absolute Percentage Error
MDE	Model-Driven Engineering
ML	Machine Learning
MO2C	Move an Operation to a Component
MO2N	Move an Operation to a new Component deployed on a new Node
MoC	Model of Computation
NSGA-II	Non-dominated Sorting Algorithm II
PDM	Platform-Dependent Model
PF^c	computed Pareto Frontier
PF^{ref}	reference Pareto Frontier
PIM	Platform-Independent Model
PSM	Platform-Specific Model
ReDe	Deploy a Component on a new Node
RMSE	Root Mean Squared Error
RMSPE	Root Mean Square Percentage Error
RT	Register-Transfer
SDCC	Small Device C Compiler
SLOC	Source Lines of Code
SMC	Stochastic Model Checking
SPP	Special Purpose Processors

SVM	Support Vector Machine
TTBS	Train Ticket Booking Service
UML	Unified Modeling Language

Abstract

Context. At the beginning of the 21st century, the presence of embedded systems increased significantly. Their diffusion allowed the collection of large volumes of data to facilitate human activities, even in near real-time. Embedded systems have been widely adopted in various domains, and this has had an impact on embedded system requirements, which may be restrictive in certain circumstances (for example, in the case of safety critical systems). Embedded Systems complexity has also grown in the last twenty years, making the traditional embedded systems design flow ineffective.
Goal. In this thesis, we provide a novel design methodology called V&V-based HW/SW Co-Design, which main goal is to cope with design complexity. Our methodology is based on the principles of HW/SW Co-Design and uses a platform-agnostic model at the beginning of the design process to reduce complexity and facilitate the comparison of different HW/SW implementations. In V&V-based HW /SW Co-Design, the platform-agnostic model is subject to two formal verification and validation (V&V) at the beginning of the design flow. These two steps aim to verify functional correctness and validate the extra-functional requirements of the system. The model is optimized through formal V&V and then used for Design Space Exploration (DSE). The thesis covers the creation of the platform-agnostic model, Co-V&V, and Co-Analysis. Moreover, we provide methods and metrics for performance and energy consumption estimation for use during DSE.
Method. We built the platform-agnostic model using UML/MARTE, which is structured in three views: Application, Communication, and Time View. The Application View describes the logical structure and the behavior of systems elements. The Communication View outlines the communication protocol used by system elements. Instead, the Time View includes the time constraints of the systems. The UML/MARTE model is converted to multiple formalisms for verification and validation. During Co-V&V, the model is transformed into a network of timed automata for functional verification and timing validation. Instead, for Co-Analysis, we employed a closed-form model for reliability analysis and Layered Queuing Networks for performance optimization. Moreover, we implemented three approaches to evaluate a system prototype during DSE. We outlined an ML-based technique for performance estimation, a metric called J4CS to compare different HW/SW solutions according to their energy consumption, and an approach combining performance models and experimentation to optimize a system prototype toward energy consumption and performance.
Results. This thesis shows that using formal methods in an HW/SW Co-Design flow can improve overall design quality. In addition, the abstraction level of the platform-specific model may be appropriate to enable verification and validation at the beginning of the design flow. During Co-V&V, we found the scenarios that invalidated system-level timing requirements of the source UML/MARTE model. Instead, thanks to Co-Analysis, we obtained a set of model alternatives that are better in terms of performance and reliability. Promising results were obtained, as well as a roadmap for future research to evaluate performance and energy usage during DSE. Among the ML models tested for performance estimation, those based on regression trees resulted the most accurate. Instead, the results of J4CS validation provided large error ranges, indicating the need for additional refinement of the metric. Finally, we saw that combining the adoption of models and experimentation can reduce experimentation time and still provide good energy consumption and performance estimation accuracy.
Conclusions. This thesis can be the basis for HW/SW Co-Design methodologies integrating formal methods in the early stages. We explored how to include rigor in an embedded systems design methodology without renouncing the convenience of using non-formal models. We investigated three highly discussed research topics in the embedded systems design domain that need further validation and exploration: (i) setting the right level of abstraction in the early stages of the design, (ii) proving the effectiveness of formal analysis, and (iii) ease formal methods adoption to practitioners.

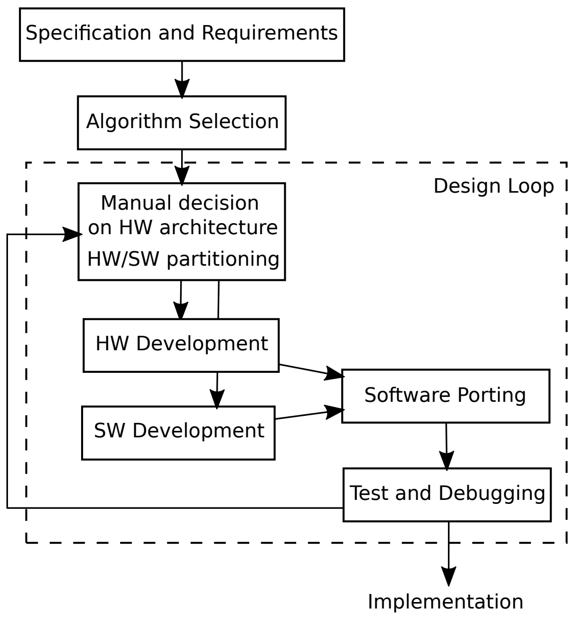
Chapter 1

Introduction

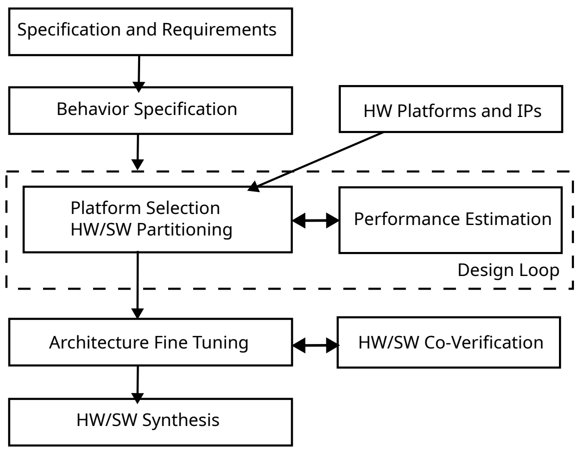
The end of the 20st century witnessed the birth of Personal Computers and Internet as major innovations, while the first two decades of the 2000s saw the diffusion of embedded systems and Artificial Intelligence (AI). Consumer electronics, medical devices, cars, and home appliances are just a few examples of embedded systems intertwined with our daily life in the last couple of decades. The main difference between embedded systems and other computing systems is that the former ones are included in other electronic systems or objects. As reported by Frank Vahid and Tony Givargis [188], it is difficult to give a precise definition of an embedded system, we can simplistically define them as systems working within bigger systems. Thus, they are tightly constrained to their application domain that influences, for example, size, power, and performance of an embedded system. The benefits introduced by embedded systems are many. Their ubiquity enabled to collect huge amounts of data in real-time and optimize daily activities. Today, we can monitor our performance during a jog, or house temperature to reduce heating energy expenditure. Besides the benefits, the unbridled desire to automate and optimize has led to inevitable downsides. By exploiting embedded systems, we can generate huge amounts of data that need additional computational power to be processed and stored in continuously powered devices. Carbon dioxide (CO_2) emissions from Information and Communication Technologies (ICT) become more significant as energy demand increases. Estimates provided by Belkhir et al. stated that ICT devices will produce 14% of global CO_2 emissions by 2040 [41]. While improving the quality of embedded systems by adding, for example, more computational power and memory, their diffusion has covered ever new functional requirements. As early as 2009, Christof Ebert and Capers Jones [81] emphasize substantial growth in the size of embedded systems software in several domains. The automotive domain is exemplar for this phenomenon, as software code grew from about a million lines to over 100 million during the period 2000-2010 [205].

As design complexity grows, traditional embedded systems design methodologies become demanding and time-consuming. Figure 1.1a depicts a commonly used methodology for designing single-functioned [1] embedded systems. The design starts with the requirements specification, which leads to the definition of the algorithm used to implement the functionalities. Thereafter, at the beginning of the design loop, designers manually partition the functionalities for deciding what to implement in hardware or software. The methodology splits into two branches dedicated to hardware and software development. The flow joins after completing the branches. Then, the software is integrated with the hardware, and the whole system is tested. The loop restarts in case defects are

[1] An embedded system that repeatedly performs a single operation.



(a) A traditional embedded systems design methodology



(b) Hardware/Software Co-Design

Figure 1.1: Embedded Systems Design Methodologies [100]

found or improvements are to be applied. Design defects can be caused by both technological choices and system behavior. Examples of defects can be unexpected deadlock states, exaggerated power consumption values, or memory overflows. As the complexity of the system grows, the number of iterations needed to output an acceptable design solution increases.

The integration of Model-Driven Engineering (MDE) practices into embedded systems design methodologies helped designers to cope with system complexity. Models can be used to represent only a subset of the system or a part of its characteristics. Moreover, they can be used for performance estimation, functional verification, and code generation. An example of a popular MDE process for hardware design is high-level synthesis, which consists of the automated generation of the register-transfer level behavior from a behavioral abstract model, e.g., finite-state machines.

Hardware/Software Co-Design methodologies exploit MDE practices to speed up and automate part of the DSE. HW/SW Co-Design envisages the adoption of a behavioral model, early in the design flow, which is independent of any implementation detail, i.e., Hardware/Software characteristics. This model is commonly also referred as platform-agnostic, technology-independent, or platform-independent model. Figure 1.1b describes the steps typically involved in a HW/SW Co-Design flow. HW/SW Co-Design introduces Behavior Specification, after the requirements specification step, where the above-mentioned platform-independent model is created. During DSE, it is decided which parts of the modeled behavior to implement in hardware and in software. For example, functionalities can be implemented as software and executed by a general-purpose processor or realized as a piece of hardware behaving as the desired function (i.e., an accelerator).

HW/SW allocation can be evaluated through Co-Simulation, where software is deployed and executed on a prototype of a hardware platform. The latter can be selected from a repository of predefined prototypes of hardware platforms. In this way, it is possible to evaluate different HW/SW partitioning solutions before implementing them. Such partitioning can be achieved through the optimization of a set of design metrics, such as performance, area, cost, and power consumption.

Since the improvement of one metric usually leads to the degradation of another one, the objective of DSE becomes finding good trade-offs in design metrics, as represented by allocations that lead to overall improvements in design metrics. The design resulting from a DSE is detailed during Architecture Fine-Tuning step, and it is subject to a further step of Co-Simulation, depicted in Figure 1.1b as HW/SW Co-Verification. This additional step is necessary to perform more detailed verification and validation before the implementation steps, which may include the automated generation of part of the implementation from the artifacts processed in the design steps (i.e., HW/SW Synthesis).

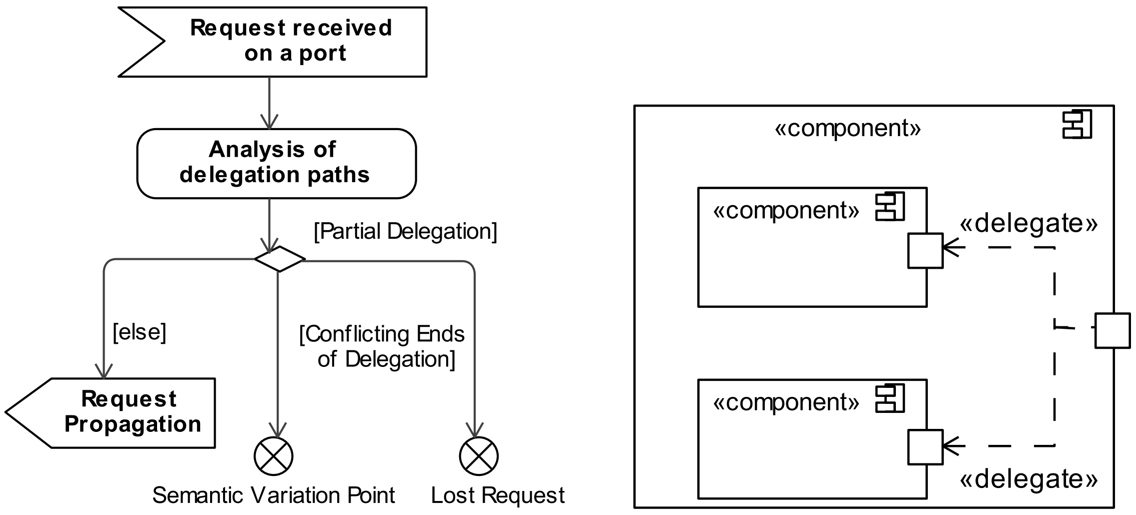
1.1 Problem Statement

HW/SW Co-Design methodologies have proven that the integration of MDE approaches into the design flow brings multiple benefits [183]. HW/SW Co-Design takes advantage from the high level of abstraction of a technology-independent model to make DSE semi-automatic, and to introduce concurrency in the hardware and software design. Thus, the analyses done during the DSE rely on the quality of the technology-independent model supplied as input. Errors made during the modeling step could easily bring to misleading analyses, and thus erroneous design decisions.

Raising the level of abstraction increases the likelihood of producing unrealistic models that can be uncompliant with technology details. This may create a gap, i.e., reality gap [96], between the model and the implementation. However, accurately reproducing an entire system or its components is typically intricate and futile. Highly detailed models can lose their effectiveness and they can be as complex to interpret as the actual object they represent. Seeking a sufficient level of abstraction is a process that requires experience in the modeling activity and complete knowledge of the domain and properties to be represented [47].

The modeling language could also contribute to inaccuracies. Modeling languages are generally partitioned in formal and informal [2] [165]. A modeling language is formal when its semantics and syntax are rigorously defined, for example, using a mathematical Model of Computation (MoC). A MoC describes how the result of a mathematical function is obtained given an element of its domain. Famous examples of formal models are Timed Automata, Petri nets, and Queuing Networks. These models are generally used for verification and validation of embedded systems, as it is possible to mathematically prove system correctness or to quantitatively estimate extra-functional attributes of the system. However, their use remains limited due to the high-learning curve of formal modeling languages and tools. On the other hand, informal modeling languages are highly used. The majority of embedded system designers rely on UML and its extensions, such as SysML or MARTE [118]. Informal models are usually flexible and easy to use. UML, for example, can be easily extended to represent domain-specific concerns, and it is supported by a plethora of tools. However, the semantics of informal languages may not be precisely defined. For example, the semantics of MARTE presents variation points, which are pieces of the language semantics whose interpretation is delegated to the user. Figure 1.2 depicts one of the semantic variation points in MARTE, i.e., when a component port has multiple delegation associations. A delegation association is labeled with `«delegate»` and represents the situation in which the port delegates the handling of a request

[2]Literature classifies some modeling languages as semi-formal (e.g. Unified Modeling Language (UML)) [131], due to their well-defined syntax. However, semi-formal languages lack of a rigorous semantics as informal languages. For this reason, in this thesis we refer to semi-formal languages as informal.



(a) Activity diagram describing the semantics of a delegation association 1.2a.

(b) Component diagram with two delegation associations outgoing from the same port.

Figure 1.2: Semantic Variation Point of MARTE when multiple `«delegate»` dependencies originate from the same port.

to elements inside the component [152]. The component diagram in Figure 1.2b shows the routing of a request to the internals of a bigger component, while Figure 1.2 a describes what happens when a request is received on a port. A request is lost when none of the internals can handle it or directly forwarded when only a single internal component can handle it. Instead, the behavior is undefined when multiple internals can handle the request.

We believe that, *by improving the model used for the DSE, we can reduce the amount of defects found in later stages, thus shortening the design loop and making better design decisions.* As a technology-independent model lacks of implementation details, we must evaluate which analyses are significant to be conducted at such a high level of abstraction and if they are meaningful for later stages. We can verify the functional correctness of the model or anticipate the estimation of some quality attributes, such as performance.

Extra-functional attributes can be quantified at various levels of abstraction. Before the DSE, by using a technology-independent model as just described, or during the DSE to evaluate different implementations of the system, thus different HW/SW technologies. In the first case, these analyses are based on the structure and MoC of the model (as it lacks of implementation details), while in the second case it is necessary to fix a candidate implementation of the system and test or simulate it. However, defining and thoroughly evaluating a system prototype is burdensome due to the numerous implementation details involved in DSE. Task scheduling, memory allocation, interfacing protocols, and processors are few examples of the elements composing the design space of embedded systems. Consequently, to speed up DSE and consider a satisfactory amount of implementation details, metrics and approaches for comparing different HW/SW technologies are needed, along with strategies for testing a specific system prototype.

A plethora of existing embedded systems design methodologies leverage Formal Methods (FMs) for V&V [161, 168]. Indeed, the benefits of integrating FMs at design time have been highlighted by both researchers and industry practices [196]. However, the adoption of FMs during design, especially in the early stages, is still a poorly used and unrecognized practice [96, 118, 196]. Some of the most important causes are an insufficient evidence of FMs effectiveness and, as already mentioned for formal languages, the lack of training among practitioners. Furthermore, the scarce adoption of FMs is exacerbated by the necessity of integrating multiple formal methods to examine different characteristics of the system (e.g., energy, performance).

This thesis aims at improving HW/SW Co-Design through the analysis of a technology-independent model that guides the DSE in the process. We investigate if enhancing the quality of the initial model can improve the overall quality of the design. We considered FMs due to the potential for V&V and the need for a more rigorous design methodology. However, FMs adoption at this level of abstraction is still scarce. The main goals of this thesis can be the summarized as follows:

G1. Introduce formal V&V in HW/SW Co-Design. This should enhance modeling precision and lead to early discovering model defects; these improvements on the model lead to better designs.

G2. Identify functional and extra-functional aspects of the system that can be early evaluated in the methodology. This goal involves finding methods and metrics to quantify these aspects at a higher level of abstraction.

G3. Define a method to ease FM adoption to practitioners.

1.2 Thesis Contribution

We envisage an evolution of the classical HW/SW Co-Design flow that integrates, before DSE, several steps of formal V&V. Figure 1.3 shows our proposal for a V&V-based HW/SW Co-Design. As in the classical HW/SW Co-Design, in Figure 1.1b, our variant starts with the construction of a technology-independent model. This step, called Co-Specification, unlike the Behavioral Specification in traditional methodology, involves a more detailed model. The technology-independent model is structured into three views: Application, Communication, and Time. The separation of design concerns in different views provides reusability of design solutions and helps reducing complexity [199]. The Application View embodies the logical structure and the behavioral aspects of the application. The Communication View describes the exchange of information among components and was first introduced by Enrici et al. [84]. The main reason behind the adoption of the Communication View is to overcome communication mismatch in the later stages of the design. As highlighted by Enrici et al., communication logic is usually embedded in the application and platform description, but it is actualized by using different models of computation. Therefore, the designer may encounter inconsistency issues during the mapping stage. Finally, the Time View represents timing requirements as time domain(s) of the system, occurrence of events, and duration of behaviors. A duration is expressed as a constraint on the time that elapses between the occurrence of two events. As described in Section 1.1, embedded system designers pursue the need for formality into the design flow but, at the same time, they struggle with the high learning curve of formal methods. By creating a correlation between informal and formal languages, designers can take advantage of the strengths of both types of languages. The former, that are expressive and easy to learn, can be used to describe the system, while formal languages can be used to verify and validate the properties of the system. The technology-independent model serves, within the V&V-based HW/SW

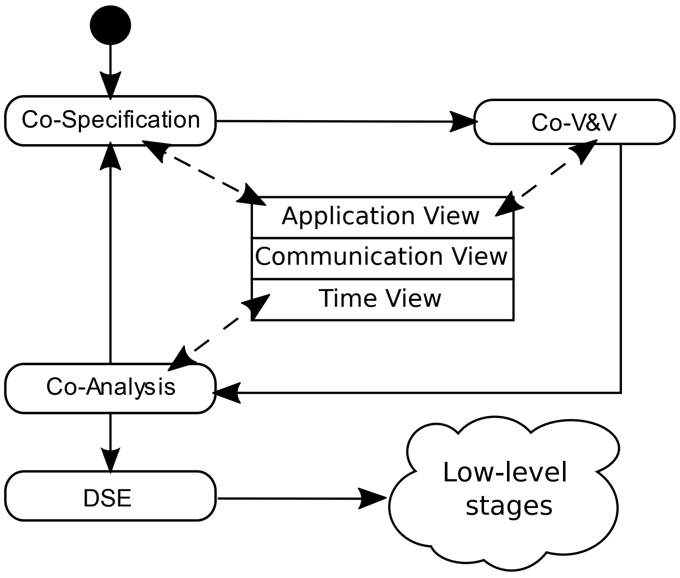


Figure 1.3: V&V-based Hardware/Software Co-Design flow

Co-Design flow, as a sole source of system design information that encompasses multiple views to delineate diverse facets of the system. Basing on Model-To-Model (M2M) transformations, various V&V tasks can be performed by deriving different formal models from the technology-independent model. Therefore, we exploit M2M transformations to perform functional verification and timing validation in the Co-V&V, as well as a preliminary performance and reliability estimation during Co-Analysis. The target formal models are generated by combining structural and behavioral concerns. For this reason, the source model views are defined according to design concerns orthogonal to structure and behavior, such as communication and time. Particularly, the communication concern illustrates how a view encompasses both structure and behavior. As mentioned before, the communication view describes the communication protocol of system components, which is part of the behavior of the system. But on the other hand, the structure defines which components are communicating and the number of links.

As previously stated, the spread of MDE practices has enabled the adoption of models for a range of formal analyses, such as verifying functional correctness and quantitative estimation. Therefore, all the steps involved in the V&V-based HW/SW Co-Design are implemented by exploiting MDE practices. We used a subset of UML/MARTE [152] to model the three views mentioned above. This model works as source of several M2M transformations. During Co-V&V, we transformed the UML/MARTE model into a network of Timed Automata to check design correctness and satisfiability of time constraints. These analysis have been carried by UPPAAL model checker [74]. In addition, during Co-Analysis, the UML/MARTE model is transformed into Layered Queuing Network (LQN) models [89] for performance evaluation, while for reliability we used a closed-form model [71]. Therefore, after defining how to evaluate system extra-functional attributes, it becomes easy to refactor the UML/MARTE model, and assess if the modifications led to improved design. During Co-Analysis, we systematically apply a set of refactoring actions to a UML/MARTE model to generate several model alternatives. Subsequently, performance and reliability of model alternatives are estimated and compared with those of the initial model, as well as the architectural distance, which quantifies the effort to obtain a model alternative from the initial one. Thus, Co-Analysis is formulated as a multi-objective optimization problem that exploits *NSGA-II* genetic algorithm to search optimal Pareto frontiers. After Co-Analysis, the UML/MARTE model is supplied to the DSE.

HW/SW Co-Design aims to identify metrics and methods for comparing different HW/SW technologies during DSE. Thus, techniques to quantitatively evaluate a prototype of the system, which consists of a candidate HW/SW partitioning, are needed. This thesis investigates the assessment of energy consumption and performance given a system prototype. We investigate two techniques for quantifying energy consumption. We study the potential of MDE approaches for estimating the energy consumption of a system prototype. Moreover, we define a metric called J4CS to compare different processors according to their energy expense to execute a given software. The former refers to an approach to compute the energy usage of an application based on its performance, in situations where energy and performance exhibit a linear relationship. Because technological factors, such as the selected hardware platform and programming language, greatly affect energy consumption, the study is conducted using a bottom-up approach. After defining a target application, it is run on a reference hardware platform, and its execution is observed under a certain workload. We created an LQN that models the behavior of an application during the experiment. The LQN is used to produce the energy and performance estimates by scaling the input workload. Instead, the J4CS metric quantifies the average energy consumed by a processor in relation to the ratio between executed C statements and assembly instructions. J4CS is connected to a processor and the software that is executed by the processor. Therefore, J4CS is useful to compare different HW/SW implementations of the system prototype. In addition to energy consumption, performance can also serve as a basis for comparing various HW/SW technologies. We defined a method that produces performance predictions, given a HW/SW prototype, and that reduces estimation time. We executed a set of software (micro-)benchmarks on several processors to obtain a dataset containing the execution time of each benchmark. We used the dataset to train a machine learning model which predicts the execution time of any input C-code function on a specific target processor. The machine learning model can be used during DSE to evaluate the performance of different software, given an HW prototype, and to reduce the overall DSE time.

Summarizing, we contribute to the following aspects of the current state of the art:

C1. A novel HW/SW Co-Design methodology that envisages multiple steps of formal V&V early in the design flow, including verifying functional correctness, as well as validating timing constraints and performance. The objective of these preliminary analyses is to remove eventual defects related to system behavior (e.g., deadlock state, bottlenecks, unexpected output).

C2. A view model made by three views (*i.e.,* Application, Communication, and Time) is defined on a subset of UML/MARTE. The views describe the logical structure, behavior, and timing constraints of a system through a technology-independent model.

C3. A set of M2M transformations to integrate different FMs in the methodology. The source UML/MARTE model is converted to a network of Time Automata for functional correctness and timing validation, and to LQNs for performance validation. Therefore, the insights gathered during the V&V steps are used to improve the source model.

C4. A method for design optimization that exploits model refactoring. We systematically apply a set of refactoring actions on a source UML/MARTE model to generate a set of design alternatives that are evaluated according to their performance and reliability. The design loop is implemented as a multi-objective optimization problem, which outputs the design alternatives leading to substantial improvements on the source model.

C5. A method that uses performance models to predict energy consumption. The method exploits a low effort experiment to create and parameterize an LQN, which is then employed to predict the energy consumption of the system subject to scaled workloads.

C6. A metric called J4CS to compare HW/SW technologies according to their energy consumption.

C7. A method based on a machine learning model to predict the execution time of a software running on a target processor.

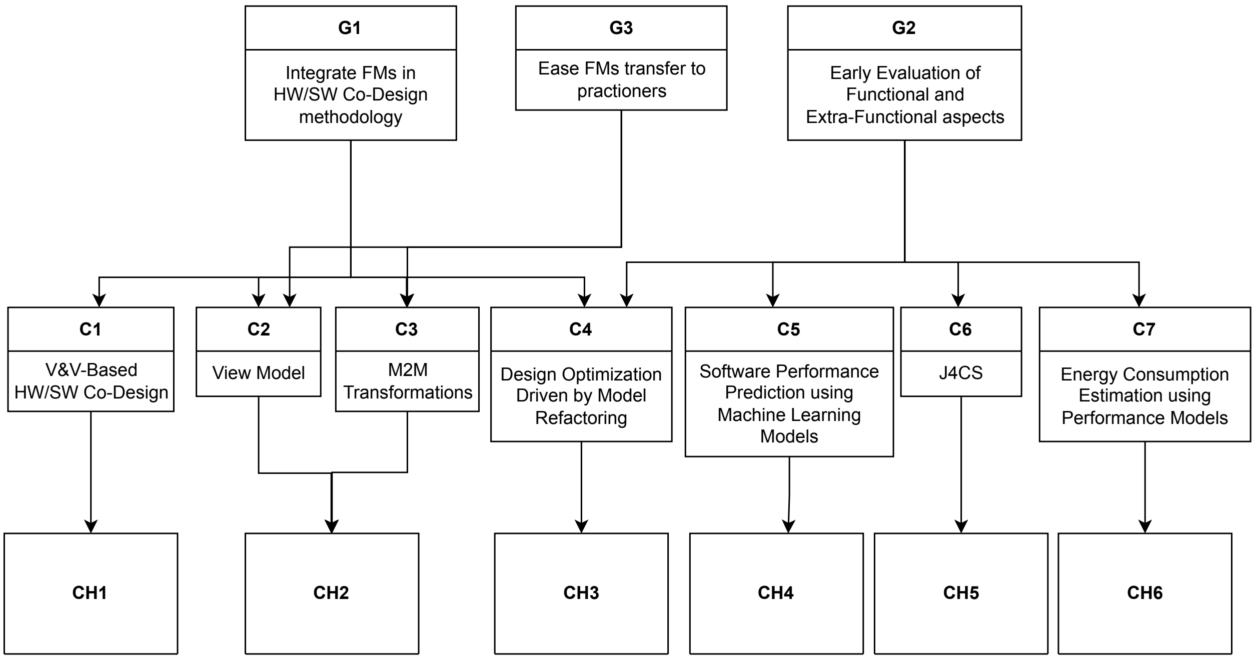

G1
Integrate FMs in HW/SW Co-Design methodology
G3
Ease FMs transfer to practioners
G2
Early Evaluation of Functional and Extra-Functional aspects
C1
V&V-Based HW/SW Co-Design
C2
View Model
C3
M2M Transformations
C4
Design Optimization Driven by Model Refactoring
C5
Software Performance Prediction using Machine Learning Models
C6
J4CS
C7
Energy Consumption Estimation using Performance Models
CH1
CH2
CH3
CH4
CH5
CH6

Figure 1.4: Goal (G) achieved through corresponding contributions (C), which are described in the associated chapters (CH).

Figure 1.4 shows how the goals stated in Section 1.1 are realized through the above listed contributions. G1 is achieved through the definition of the V&V-based HW/SW Co-Design and the implementation of the methodology. Thus, G1 comprises the definition of the view model (C2), the M2M transformations to convert the UML/MARTE model to the formalism enabling V&V (C3), and the refactoring-based optimization of the UML/MARTE defined for the Co-Analysis (C4). The view model and M2M transformations are the artifacts to ease designers access to formal methods, thus they contribute to G3. Each view of the source model embeds all the information to derive a subset of the features of the target model, for example, structure or behavior. This aspect facilitates the remapping of the improvements obtained from the formal analysis. For instance, the structure of the network of Timed Automata used for Co-V&V is created based on the logical structure defined in the Application View of the UML/MARTE model. Instead, each automaton is derived from the behavioral description of each element of the source model. Instead, thanks to M2M transformations, the designers can effortlessly acquire the formal models for V&V. Finally, we realized G2 during Co-Analysis, which provides a method to evaluate the performance and reliability of a UML/MARTE model, and through C5, C6, and C7, contributing with methods and metrics for performance and energy consumption analysis given a system prototype.

1.3 Related Work

The main motivations for using HW/SW Co-Design are to cope with increasing complexity systems, to reduce design time and, at the same time, to produce correct and efficient implementations. To address the mentioned challenges, the V&V-based HW/SW Co-Design, along with existing Co-Design methodologies, follow a divide-and-conquer approach that involves analyzing functional and extra-functional aspects of the system using an abstract model, which is systematically refined until system implementation. This philosophy is also embraced by ForSyDe [168] and S3D [105], which are two Co-Design methodologies similar to our V&V-based HW/SW Co-Design.

As it is emphasized in ForSyDe, this thesis highlights the necessity of integrating formal analysis in the design flow. ForSyDe is a Co-Design methodology that can be divided in two main sections: the functional and the implementation domain. In the functional domain, a technology-independent model, created on formal languages, undergoes the application of several design transformations towards a more detailed implementation model. The source and the resulting model adhere to the same formal semantics, which allows to execute the same verification and validation methods on both models. The implementation model is used, in the implementation domain, for Design Space Exploration, thus the model is mapped to a candidate HW/SW solution. The V&V-based HW/SW Co-Design differs from ForSyDe in the adoption of an informal language, i.e., UML/MARTE, to create a technology-independent model. This choice is made to facilitate the adoption of formal methods in practice, thus to ease the application of the methodology. Another substantial difference lies in Co-Analysis, which aims at tuning the abstract model according to estimated performance and reliability. In this way, we provide one or more optimized design solutions to the DSE.

Co-Analysis is also one of the differences that our design flow has with respect to S3D. Our proposal and the S3D methodology share the adoption of a multi-view UML/MARTE [152] model that includes the required design information. This model works as a single source for several design activities, such as schedulability analysis, software synthesis, and simulation. The views of the UML/MARTE model are hierarchically organized to address various design aspects at different levels of abstraction. The views are first divided into Design and Verification (D&V) views and Tool-Specific Views. D&V provides information about system design and those needed for verification and validation. The Tool-Specific Views, instead, refer to information needed by design tools. The D&V views enable three models: Platform-Independent Model (PIM), Platform-Dependent Model (PDM), Platform-Specific Model (PSM). The PIM describes system functionalities without providing any information about the implementation. In S3D, the PIM is used to generate platform-dependent code by using a tool called eSSYN [15]. This tool generates binaries embodying a possible implementation of system functionalities. Binaries can be directly loaded on the target embedded system. PDM describes the mapping between PIM and PSM, thus it details the implementation of the system. Our work has two main similarities with the S3D modeling approach. Firstly, the use of a UML/MARTE model as a single source of information, and the use of transformations, e.g., code generation in the case of eSSYN, to perform further analysis. However, the methodologies have different goals. S3D envisages a modeling methodology that contains all information needed by designers throughout the design phase, thus covering also implementation details. In this thesis, however, we dwell on the abstraction level of the PIM to investigate which analyses can be anticipated before moving on to DSE. We introduce stages aiming at the qualitative improvement of a PIM, i.e., Co-V&V and Co-Analysis. Finally, Co-V&V and Co-Analysis exploit additional (formal) models that keep the same level of abstraction as the source model, thus providing insights about the flaws of the source model without involving concerns at lower level of abstraction. It is worth noting that our study can be easily integrated into S3D or ForSyDe and vice versa. For example, we can employ the subset of UML/MARTE used to create a PIM in S3D to perform Co-V&V and Co-Analysis.

1.4 Thesis Outline

The thesis is structured in 2 parts. The first part describes Co-Specification, Co-V&V, and Co-Analysis, which embody the creation of a technology-independent model, functional verification and timing validation, as well as performance and reliability validation. The second part focuses on DSE and describes the set of methods and metrics outlined to evaluate a candidate system prototype. Figure 1.4 illustrates how the set of contributions (C) achieve the defined research goals (G). The contributions are linked to the chapters (CH) of the thesis that describe them. The main goal and contribution of the thesis is the V&V-based HW/SW Co-Design methodology (G1 and C1). A possible implementation of the methodology is outlined along the chapters of the thesis. Chapter 2 delves into Co-Specification and introduces Co-V&V, which consists of (i) a transformation from UML/MARTE to UPPAAL Timed Automata, and (ii) a preliminary functional verification and timing validation that exploits the UPPAAL verifier. Co-Specification introduces the view model, which is made by the Application, Communication, and Time view (C2). Chapter 3 presents Co-Analysis (C4) that, as described in Section 1.2, exploits a genetic algorithm to optimize a set of model alternatives. The latter are generated by refactoring a source UML/MARTE model. The evaluation process involves the quantification of performance and reliability (G2). Performance, for example, is quantified by using a M2M transformation from UML/MARTE models to LQNs. Thus, the UML/MARTE model becomes a single source of design information that can be systematically improved by converting it to Timed Automata and LQN (C3). Designers do not have to be proficient in Timed Automata and LQN, as the latter models are automatically generated from the UML/MARTE model (G3). The second part of this thesis introduces techniques and metrics to evaluate a given system prototype, i.e. a candidate HW/SW partitioning. Chapter 4 and Chapter 5 detail, respectively, a machine learning based performance prediction method (C5) and the J4CS metric (C6). Chapter 6 describes C7, *i.e.,* how to exploit performance models to predict energy consumption and reduce experimentation time. The thesis ends with conclusions and future works in Chapter 7.

Part I

Platform-Independent Techniques

Chapter 2

From UML/MARTE Specifications to Functional Verification & Timing Validation

This chapter describes a possible implementation of Co-Specification and Co-V&V. Co-Specification is realized using a subset of UML/MARTE for modeling reactive components, message-based communication, and time constraints on the duration of interactions. Co-V&V, instead, involves the problem of checking UML/MARTE models to assess control-flow correctness and time constraints satisfiability. These analyses are based (as mentioned in the introduction) on UPPAAL Timed Automata. The chapter provides a semantic mapping from UML/MARTE to a network of timed automata to extend the analysis done with UPPAAL to the source model. The transformation exploits both system architectural and behavioral descriptions as well as the organization in views of the model. Co-Specification and Co-V&V steps are shown in some details through the FIRGCD (Finite Impulse Response Greatest Common Divisor) toy example [160]. The verification and validation conducted with UPPAAL indicate that our approach is particularly effective in discovering design flaws located in the communication protocol as well as those arising from the internal behavior of components. This chapter is organized into the following sections: Section 2.1 describes UML/MARTE and the main features of UPPAAL, Section 2.2 illustrates the Co-Specification step, Section 2.3 deepens the semantic mapping between UML/MARTE and timed automata, Section 2.4 reports the properties verified using UPPAAL and the results of the verification on the considered toy example, Section 2.5 discusses related work while Section 2.6 concludes the chapter.

2.1 Background

2.1.1 UML/MARTE

The Unified Modeling Language (UML) [153] is a general-purpose modeling language, among the most used in the embedded systems domain [118]. Due to its vast semantics, UML enables the representation of structural and behavioral aspects at different levels of abstraction. For example, the UML Component Diagram is often employed to model component-based architectures, while the UML Activity Diagram is used to represent fine-grained behaviors. However, UML, as a general-purpose language, fails to represent domain-specific situations. For example, when modeling real-time behaviors where a more precise representation of timing properties is needed. For this purpose, UML can be extended using the Modeling and Analysis of Real-Time and Embedded systems (MARTE) [152] profile. Indeed, MARTE provides classes such as `TimedProcessing` to assign a duration to system entities. Moreover, MARTE includes the Value Specification Language (VSL) to specify time constants and expressions. For example, the tuple `(100,ms)` denotes 100 milliseconds. This tuple can be used to define a duration constraint such as `(endEvent - startEvent) <= (100, ms)`. This constraint binds the interval between two observed events to last at most 100 milliseconds.

2.1.2 UPPAAL

The UPPAAL model checker [40] aims at the verification of real-time systems represented via a network of timed automata. Automata execute in parallel and they can be synchronized using channels. A rendezvous is modeled by labeling two edges of two different automata with `c?` and `c!`. Moreover, UPPAAL automata are specified as parametric templates that may be extended with discrete and real-valued variables named clocks. A timed automaton is made by a set of locations and a set of edges. A location may have an invariant that denotes the property held while staying at that location. An initial location is unique and describes the starting condition of the automaton, while an urgent one defines a location with no delay. Edges may have guards to allow transitions, channel synchronizations, and variable assignments.

System properties are expressed using Computational Tree Logic (CTL) [74]. Examples of properties are deadlock freedom, program termination, and location reachability. A CTL formula may be defined over the states or the paths of the timed automata network. Properties over paths are written using two CTL quantifiers: All (A) and Exists (E). A ϕ says that ϕ should hold for all paths, while E ϕ describes a property valid for one or more paths. In UPPAAL, a path quantifier should be followed by a linear time operator between Finally (F) ϕ (i.e., $\Diamond$) and Globally (G) ϕ (i.e., $\Box$). They express statements specific to a path. Indeed, F ϕ is used when ϕ can eventually hold in the path, while, using G, ϕ should be satisfied in all the subsequent states.

Timed automata are insufficient for modeling non-deterministic decisions [74]. For this reason, UPPAAL has been extended to support Stochastic Model Checking (SMC) [74]. Probabilistic branches are defined by labeling outgoing edges of branching points with probabilities. This enhancement allows for more fine-grained analyses. It is possible to write properties over a fixed observation time and number of executions. For example, $Pr[<= 100; 200](\Diamond\phi)$ returns the probability to satisfy ϕ in 100 time units, and the expression is evaluated by performing 200 runs. The precision of the result is proportional to the number of repeated runs. Lastly, the SMC extension provides an operator to calculate the expected value of an integer clock variable with an interval of time. Such an expression is written as $E[<= 100; 200](max : \phi)$ that calculates the average of ϕ by considering, at each run, the maximum value that ϕ can assume.

2.2 Co-Specification

As explained in the Introduction, the proposed HW/SW Co-Design flow involves an initial modeling step called Co-Specification. Section 2.2.1 details the constructs of UML/MARTE used, in each view, to describe system structural, behavioral, and time concerns. Instead, Section 2.2.2 presents the FIRGCD toy example.

2.2.1 UML/MARTE Subset

The three views of the system embed, respectively, the application, the communication protocols, and the timing constraints. Aspects as structure and behavior are cross-cutting to the views. For example, system behavior is partly represented in the Application View, for what concerns a system element internal behavior, and partly represented in the Communication View, for what concerns the interactions among system elements. This section describes the elements of UML/MARTE selected to model FIRGCD structure, behavior, and time.

Structure Specification

The description of the logical structure of the system involves system entities and their interconnections. These static aspects are shown using a UML Component Diagram, which consists of a set of UML Components connected through UML Connectors. Components exchange data over connectors through one or more UML Ports. Each component may contain sub-components in a hierarchical fashion. Data are forwarded to sub-components by binding the ports of a component to those of its sub-components. Data forwarding is modeled annotating a UML association with `«delegate»`. Figure 2.1 reports the structure specification of FIRGCD, which is described in details in Section 2.2.2.

Behavior Specification

A designer can leverage the extensive semantics of UML to define fine-grained behaviors. Such a detailed behavioral specification reproduces the flow of operations executed by system components. In an activity diagram, the UML `CallOperationAction` element represents the action of calling of an operation. Moreover, the execution flow may involve branching points and possible exceptions. Branching points are denoted by a UML `DecisionNode`, while exceptions are indicated using the UML `RaiseExceptionAction` element.

Since embedded systems are typically reactive [188], it is necessary to model behaviors dealing with asynchronous events. The action of waiting for an event has been modeled using the UML `AcceptEventAction` element. This element has an attribute (i.e., Trigger) defining the kind of expected event. However, UML lacks semantics for representing the reception of specific types of messages. We employ the MARTE `«DataEvent»` stereotype to fill this lack. The `«DataEvent»` stereotype extends the UML `AnyReceiveEvent` element by adding a tag denoting the message type. After processing, data are written on ports using the UML `SendObjectAction` element. Instead, signals reception and sending are modeled, respectively, through UML `ReceiveSignalEvent` and `SendSignalEvent`.

Time Specification

In the time model of UML, time instants correspond to event occurrences. UML provides the concept of Observation to represent an instant (i.e., UML `TimeObservation`) during an execution. Observations can be combined with value specifications to define temporal constraints on

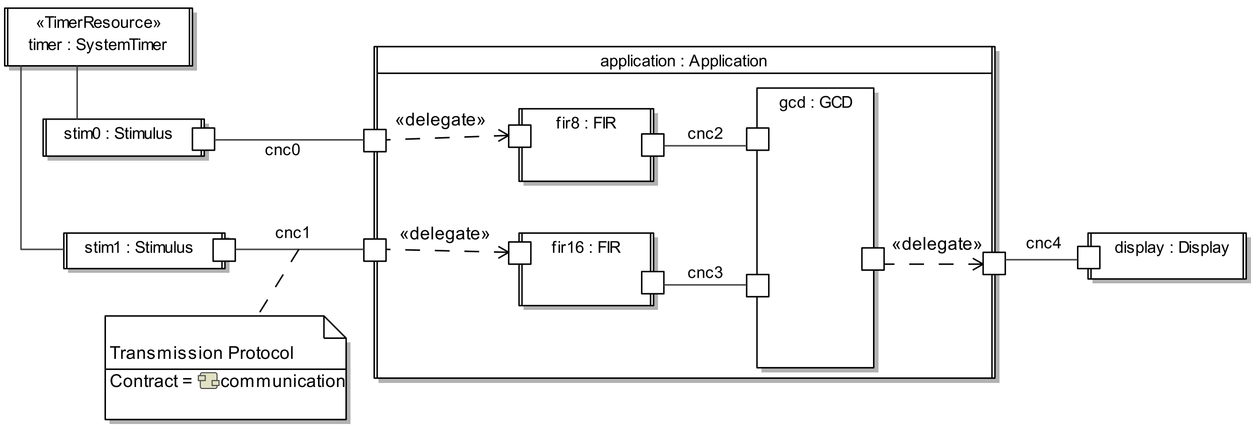


Figure 2.1: FIRGCD Component Diagram

system behavior. For this purpose, it is useful to exploit the syntax of the Value Specification Language (VSL) [152], embedded in MARTE, to define time values. For example, the VSL expression $endEvent - startEvent$ represents the duration between the occurrence of two events. In our model, $endEvent$ and $startEvent$ are two `TimeObservation` associated to the occurrence of two events. Therefore, we define constraints on execution chunks written as $event2 - event1 \backsim x$ where $event1$, $event2$ are instances of `TimeObservation`, $\backsim \in \{>=, <=, >, <, =\}$, and x is a real number or a VSL time value.

2.2.2 Modeling the FIRGCD Toy Example

The features of FIRGCD [160] made it suitable for V&V of reactive and distributed computation. FIRGCD [160] is made by a network of processes communicating via point-to-point channels. The processes react upon the reception of data or signals.

Application View

Figure 2.1 shows the structure of FIRGCD. The structure comprises a cluster of components dealing with the generation of inputs for the Application. This cluster includes two Stimulus (i.e., `stim0` and `stim1`) and a Timer. The stimuli generate a random integer every time they receive signals from the timer. The generated integers are passed to the FIR filters: `fir8` and `fir16`. The filters execute whenever an integer shows up at their input port. The GCD component waits until data is ready on both input ports before starting the processing. GCD receives the data from the filters and sends the result to a Display instance. The latter component shows the result of the Application. The behavioral specification of FIRGCD embodies the interactions among system entities (i.e., system-level behavior) plus the description of their internal behavior. The latter is specified by the class to which each element belongs to. For example, the `fir8` and `fir16` components, in Figure 2.1, have the same behavior since they are instances of the same class (i.e., the FIR class). In system-level behavior, they can be parameterized differently and associated with different time constraints.

Communication View

The Communication View describes the exchange of information among components and was first introduced by Enrici et al. [84]. The main reason behind the adoption of the Communication

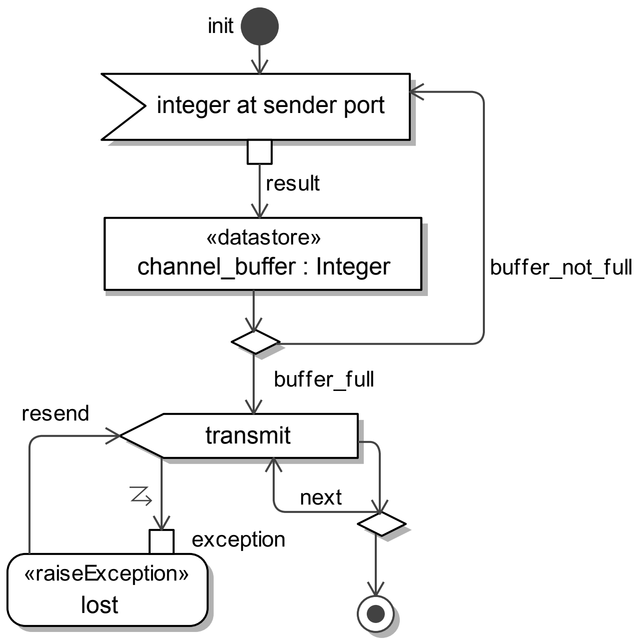


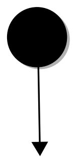
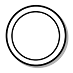
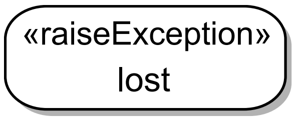
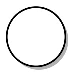
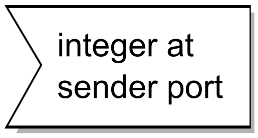
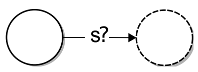
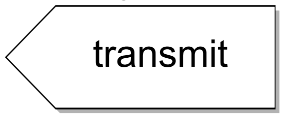
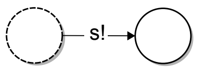
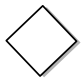
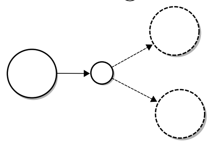
Figure 2.2: Communication Protocol adopted by the connectors of FIRGCD

View is to overcome communication mismatch in the later stages of the design. As highlighted by Enrici et al., communication logic is usually embedded in the application and platform description but it is realized using different models of computation. Therefore, the designer may encounter inconsistency issues during the mapping stage of a HW/SW Co-Design flow. For simplicity, we consider all the connectors of FIRGCD, i.e. the solid lines in Figure 2.1, implementing the same protocol. Figure 2.2 shows the communication protocol of FIRGCD. The protocol implemented in our toy example consists of a point-to-point communication. The exchange is unidirectional, so it involves a single sender and receiver. The data is kept until the buffer is filled. Once full, buffer elements are transmitted one at a time to the receiver. Data received while the buffer is full are lost. Moreover, data could be lost during transmission. In this case, the protocol provides for the re-transmission of the data. The flow ends whenever everything is successfully transmitted.

Time View

In FIRGCD, time flows continuously. The generation of integers is regulated by a Timer, which periodically sends a signal to `stim0` and `stim1` after a delay of 100 time units. To model the duration of the delay, we used the MARTE `TimedProcessing` stereotype. The activity diagram of the Timer includes an action named `delay` annotated with `TimedProcessing`. This stereotype has the tag `duration` set to 100. Moreover, MARTE helps in constraining the duration of actions and execution chunks. Such a constraint expresses the duration between the occurrence of two events. In FIRGCD, we impose a limit of 100 time units on the execution chunk that begins when the timer is triggered (i.e., receipt of `reset` on `stim0`), and ends when the output is shown to the user. This constraint, named *global_constraint* is defined using the following VSL expression: $displayed - reset < 100$.

Table 2.1: UML/MARTE to UPPAAL Timed Automata Transformation

UML/MARTE Element	UPPAAL Element
InitialNode 	InitialLocation 
Action 	Location 
AcceptEventAction 	Reception of Signal 
SendSignalAction 	Emission of Signal 
Decision 	Branching Point 

2.3 UML/MARTE to UPPAAL Timed Automata Transformation

In order to exploit UPPAAL for functional verification and timing validation, we defined a semantic mapping from UML/MARTE to UPPAAL timed automata network. The network is built considering the architecture of the model and the behavioral description of each component. Table 2.1 reports all the elements involved in the generation of an automaton of the network. Each activity diagram represents the internal behavior of a system element and it has a corresponding automaton. In our case, each process represents the behavior of a component. The node starting the flow of an activity diagram (i.e., `InitialNode`) is converted into the initial location of the target automaton. The remaining locations are obtained transforming each action found in the activity diagram. The edges of an automaton are obtained from the control flow of the activity. At this preliminary stage, we do not include the object flow in the verification. We used the architecture of the model, in Figure 2.1, to define the topology of the network of timed automata. In UPPAAL, we bound connectors by modeling the data exchange between components. Indeed, in the UML/MARTE model, components can write and read data to/from the connectors. This aspect is modeled in UPPAAL using channels. For each connector, we create two channels: `conn` and `data`. The `conn` channels are used to model rendezvous between components and connectors. We use `conn!` to denote a component sending data to a connector, while `conn?` embodies a connector waiting for data to be written. The `data` channels have the same semantics but the roles are inverted. In the same way, we use UPPAAL synchronizations to model the arrival and the emission of signals. In our model, the signal exchange is modeled using `AcceptEventAction` for the reception and `SendSignalAction` for the emission of signals. Finally, we translated UML decision nodes into UPPAAL branching points. Each outgoing edge of a branching point is annotated with a probability chosen by the designer.

The transformation is designed to involve only the semantics required to obtain a network of timed automata that fully reflects the source model. For example, specifying input/output (I/O) ports in the UML/MARTE model would not add semantics to the transformation. A designer may use I/O ports to model rendezvous between system components. However, component synchronization is already modeled with UML/MARTE signals and specified in the activity diagram of system components. Thus, we decided to keep the semantics of the ports of the source model as abstract as possible.

2.4 Co-V&V

The timed automata network resulting from the transformation is supplied to UPPAAL for verification and validation. We defined a set of properties, expressed in Computational Tree Logic (CTL) [74], to check functional correctness and validate the timing constraints defined the in UML/MARTE model. Table 2.2 reports the CTL formulae along with the results produced by the UPPAAL verifier. These results are used to improve the UML/MARTE model defined during Co-Specification. It is evident by looking at the description of FIRGCD, in Section 2.2.2, that the results of the V&V are conditioned by the quality of the communication protocol. Indeed, the latter implements the most sophisticated behavior. Thus, in the case of FIRGCD, the quality of the initial design can be improved by simplifying the structure of FIRGCD or the logic of the communication protocol, respectively, in Figure 2.1 and Figure 2.2. Therefore, before calculating the results, it is necessary to fix the parameters that can affect the results of the formulae. In FIRGCD, these parameters are the probability values labeling the edges of the branching points in Figure 2.2. As described in Section 2.2.2, the protocol starts the transmission when the buffer is full. We test the system considering a small capacity buffer to obtain many transmissions during the observation time frame. The situation involving a full buffer is represented setting a likelihood of 80% between `DataArrived` and `Transmit`. The functional correctness of the system can be verified by defining CTL formulae. CTL formulae represent different aspects of the system, for example, the reachability of a state or the presence of a deadlock. Properties such as deadlock freedom, fairness, safety, and liveness ensure behavior correctness and, thus, should hold across all kinds of systems. Instead, constraints involving quantities such as "component A should return a result within X milliseconds" may vary according to the kind of system. A well-known example is real-time systems, in which behavioral correctness also depends on when actions are performed. In the real-time systems domain, timing constraints can be hard, firm, and soft real-time. This classification is defined according to the consequences of violating a constraint which can be different with diverse real-time systems. In FIRGCD, behavior correctness is ensured by reachability of timed automata locations and deadlock freedom. During the simulation of the timed automata network, reaching a specific location can represent a correct or faulty behavior of the system. Besides, whether this is done within a time frame reveals the satisfiability of timing constraints. Before proceeding to deeper analyses, it is important to know if the system execution does not lead to a deadlock. Deadlock freedom of FIRGCD is guaranteed executing Equation 2.1 in Table 2.2 in the UPPAAL verifier.

Table 2.2: System Properties along with corresponding Computational Tree Logic (CTL) Formulae and UPPAAL Verifier Output. Legend: $\overline{m}$: average messages, $\overline{tu}$: time units

Property		CTL Formula	Verifier Output
Deadlock Freedom	(2.1)	$A\Box\, not\, deadlock$	Satisfied
Reachability	(2.2)	$E\Diamond\, display.Displayed$	Satisfied
	(2.3)	$A\Diamond\, display.Displayed$	Unsatisfied
	(2.4)	$Pr[<= 300; 2000](\Diamond\, display.Displayed)$	$82\% - 85\%$
Communication	(2.5)	$E\Diamond\, forall\,(i : conn\_t)\, Connector(i).Transmitted$	Satisfied
	(2.6)	$Pr[<= 200; 500](\Diamond\, Connector(4).Transmitted)$	$42\% - 51\%$
	(2.7)	$E[<= 300; 500](max : sum(i : conn\_t)Connector(i).Transmitted)$	$2.078\overline{m}$
	(2.8)	$E[<= 300; 500](max : sum(i : conn\_t)Connector(i).Lost)$	$1.48\overline{m}$
Time	(2.9)	$E[<= 100; 2000](max : Stopwatch.x)$	$130.75\overline{tu}$

2.4.1 Reachability of Display

In FIRGCD, Display reachability can be considered evidence of successful execution. Using the Stochastic Model Checking features of UPPAAL, we can calculate the likelihood of reaching the `displayed` location. Equation 2.2 and Equation 2.3 express, respectively, the reachability of the `displayed` location in some and all the execution traces. The satisfiability of these formulae suggests whether the `displayed` location is eventually reached. Therefore, in case the formulae are satisfied, a designer can perform a deeper analysis checking the likelihood to reach `displayed`. Equation 2.4 calculates the probability that the `displayed` location is reached within the first 300 time units. A low probability value can be a symptom of problems occurring before arriving at Display.

The UPPAAL verifier validates the Equation 2.2, while the compliance of Equation 2.3 is not guaranteed. The non-compliance of Equation 2.3 implies that Display might never be reached. This behavior may stem from the communication protocol, since the protocol models the situation in which there is a loss of data. Thus, in the worst case, data could be infinitely lost and never reach Display. The execution of the Equation 2.4 outputs that the probability to reach the `displayed` location within 300 time units lies between 82% and 85% measured in 2000 runs.

Figure 2.5a reveals that the `displayed` location is periodically reached. This phenomenon may originate from the behavior of Timer. Indeed, the Timer starts the execution flow by sending a signal to `stim0` and `stim1` each 50 time units. In order to prove it, we have shortened the Timer by decreasing the delay to 20 time units. The results show that the probability increased to a range of 89% to 92%. This hypothesis is supported by comparing Figure 2.5a and Figure 2.5b. Indeed, Figure 2.5b shows a chart shifted to the left with more concentrated values than Figure 2.5a. However, the average number of data lost by all the connectors, calculated using the Equation 2.8, increases from 1.48 to 1.84 average messages. This issue can be fixed by changing the structure of the UML/MARTE model, in Figure 2.1, by merging the connectors linking the filters with the GCD component. Moreover, the resulting connector will implement a larger buffer to store the results of the filters. A larger buffer can be represented by decreasing the probability that the buffer is full from 80% to 60%. Thus, in Figure 2.3, we change the probability value on the edge connecting `DataArrived` to `Transmit`. The reachability of the `displayed` location becomes almost certain within 300 time units, while the average number of data lost decreases from 1.84 to 0.91.

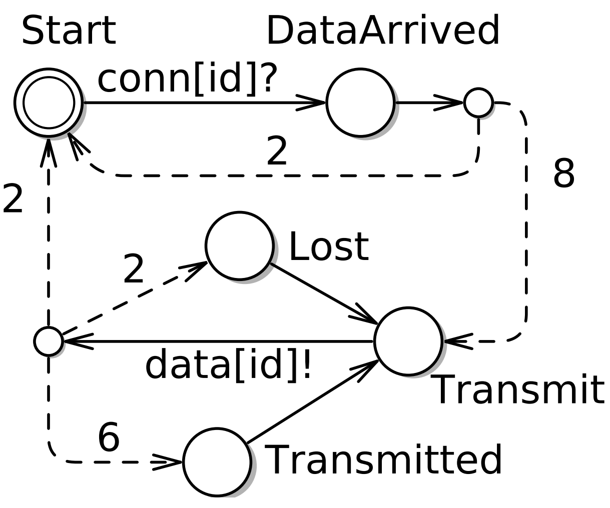


Figure 2.3: Connector Automata

2.4.2 Communication Correctness

The benefits of introducing a Communication View become evident in the Co-V&V step. Throughout the transformation, each communication protocol is translated into an independent UPPAAL template. In this way, every protocol results in a different timed automaton that can be analyzed separately. This modeling decision reduces V&V complexity since the designer may check communication properties independently from the network behavior. The architecture of FIRGCD, in Figure 2.1, presents five connectors implementing a common communication protocol. Therefore, the resulting network contains a unique UPPAAL template, embodying the protocol behavior, and a UPPAAL process for each connector. The functional correctness of the protocol implies that messages are eventually transmitted. This test can be done by checking the location representing the system state after the transmission of a message. However, due to the shape of the Activity Diagram, the obtained protocol automaton does not include such a location. Therefore, we add a location to the protocol automaton called `Transmitted`. Figure 2.3 depicts the protocol automaton including the `Transmitted` location. This situation can happen whenever a designer represents a behavior at a high level of abstraction and desires to perform a more fine-grained verification during Co-V&V. In our case, the `Transmit` location denotes the system state when ready to send data. However, it does not inform about what happened before (i.e., if data has just been sent or not). Equation 2.5 guarantees that data is eventually transmitted by all the connectors. Instead, Equation 2.6 returns the probability of transmitting data from the Connector(4), namely `cnc4` in Figure 2.1, within the first 200 time units. Consequently, the same property can be checked when data is lost. Expressions involving time values, like Equation 2.5, are well suited for checking the satisfiability of timing constraints defined in the Time View.

The UPPAAL verifier outputs a range between 41% and 50% for Equation 2.6, while a likelihood within 38% and 47% that information is lost. Further insights about the system model may arise by checking the average number of data items transmitted and lost. Equation 2.7 outputs 2.078 that denotes the maximum average messages transmitted within 300 time units. The same property can be written for the loss of data. In this case, the verifier returns an average of 1.48 messages.

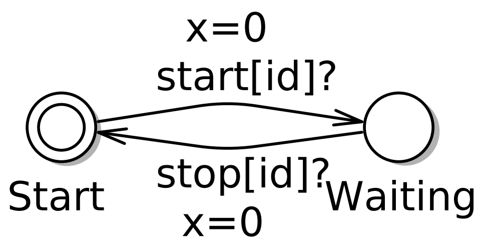
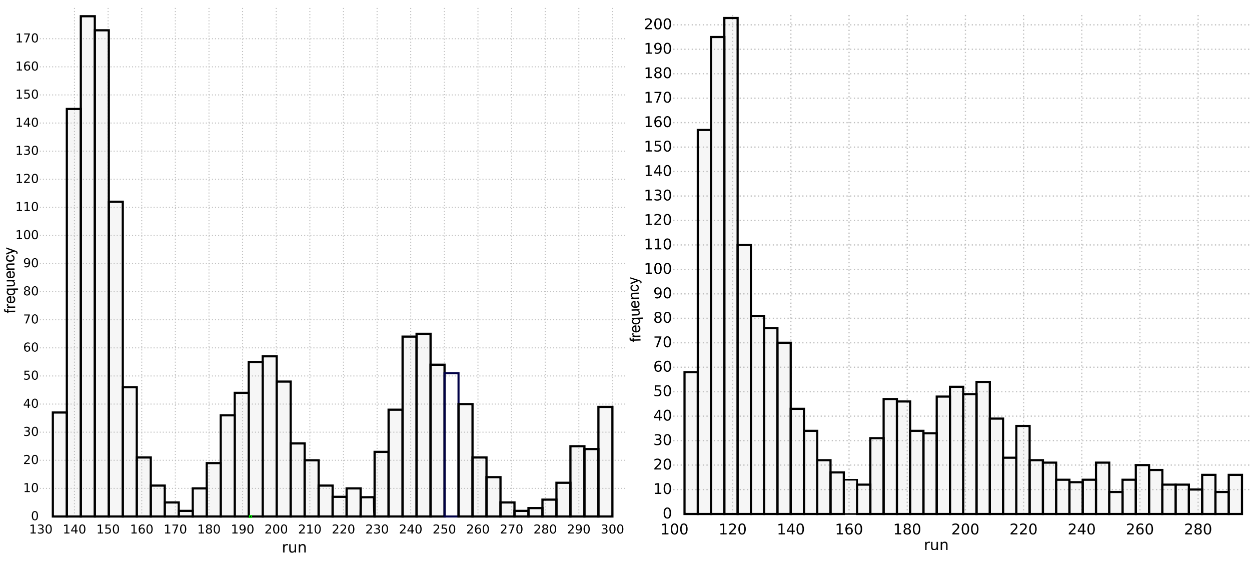


Figure 2.4: Stopwatch Automaton

2.4.3 Timing Validation

The Time View describes the definition of constraints on the time elapsing between the occurrence of two events. One example of such a constraint, i.e. the `global_constraint`, is described in Section 2.2.2. This constraint binds time from when `stim0` generates an integer to the moment this integer will be displayed. An event can occur multiple times during an execution. Therefore, to measure a time interval, we need to track the time elapsing between the *ith* occurrences of two events. The introduction of variables or locking mechanisms to control the occurrence of events seemed a solution that would substantially complicate the network of timed automata. For this reason, we introduced an additional timed automaton behaving as a stopwatch. Figure 2.4 depicts the automaton of the stopwatch. The stopwatch switches to the waiting state upon the arrival of *start*! and exits at the reception of *stop*!. Hence, the duration of intervals can be checked by measuring how much time the stopwatch spends in the waiting state.

We check the satisfiability of *global_constraint* through Equation 2.9. This formula includes the expression $max : Stopwatch.x$ that denotes the maximum value of x over a run. The variable x is a clock that is reset as the stopwatch enters and exits the waiting state. The UPPAAL verifier evaluates Equation 2.9 for 2000 runs, each lasting for 300 time units. At each run, the verifier takes the maximum value of x, and then it returns the average of all values chosen in 2000 runs. Equation 2.9 outputs that the average duration of the interval between `reset` and `displayed` equals to 130.75 time units. So, *global_constraint* is not satisfied since the duration between these two events should not exceed 100 time units. The same thing happens considering the improved structure of FIRGCD as described in Section 2.4.1. In this case, Equation 2.9 results in an average of 118.59 time units. It is worth noting that the satisfiability of *global_constraint* is remarkably affected by the delay of the timer as well as the duration of the `shifting` and `evaluation` operations of the filters. Consequently, it is sufficient to reduce these duration values to meet *global_constraint*. This is supported by the UPPAAL verifier that outputs an average of 96.51 time units just decreasing the duration of the `evaluation` operation from 50 to 25 on `fir8` time units and from 45 to 25 time units on `fir16`. Such analyses suggests, to the designer, possible operation duration values to consider for subsequent design phases.



(a) Timer's period equal to 50 time unit

(b) Timer's period equal to 20 time units

Figure 2.5: Number of times the displayed state is reached in 1000 time units by stimulating the application with different timers.

2.5 Related Work

UML/MARTE is a widely used language for modeling the characteristics of real-time embedded systems. However, it lacks support for functional verification and validation of time constraints. A popular strategy to achieve verification consists in the creation of a semantic mapping towards UPPAAL timed automata. This approach has already been investigated by several works [61, 62, 75, 99, 101, 182, 204].

Some of the most representative work start from a state-based representation of system behavior for the conversion into timed automata. Surdyevara et al. [182] propose a mapping between UML Statemachine and UPPAAL timed automata. The UML state machines are annotated using MARTE+CCSL to express causal constraints of clock instants. A similar work is Chen et al. [61] which focuses on verification of component-based systems. They propose a novel mapping from MARTE+CCSL to UPPAAL-TIGA timed I/O automata to model components interfaces. These works differ from ours by the level of abstraction of the source model. Their conversion maps a state to a location. In this way, a location of a timed automaton represents the state of the system when it assumes a particular configuration (i.e., values of its variables or set of active objects) and thus when an invariant condition holds. In this paper, a location represents a fundamental unit of executable functionality. Hence, the semantics of actions in activity diagrams. This is necessary to examine specific execution flows, including those internal to the state of a system. Besides, state representation would lead to inaccurate time estimation and thus invalid timing validation. Both [182] and [61] employ CCSL to model causal constraints on event occurrences. They constrain event order or their timing. Our work diverges since the control flow of events is expressed using the constructs of activity diagrams. Instead, we constrain the duration of actions and executions rather than events timing.

Another peculiarity of our work is the adoption of stochastic model checking to analyze the likelihood of reaching a location. Gu et al. [99] has commonalities with our work both for the adoption of stochastic model checking and the usage of activity diagrams. They exploit stochastic modeling to represent user inputs and action duration. Gu et al. use a different strategy to model actions duration. Indeed, they annotate actions with a range denoting their execution time variation. Instead, we express action duration constraints using MARTE/VSL. Both of the approaches use properties written as $Pr[<= T](\psi)$ to check whether constraints are validated. The works differ in how action durations are described in UPPAAL. Gu et al. use a two-dimensional array storing the execution duration variation for each node. We implement the duration constraint as a guard on transition. This choice was made considering that we perform timing validation in the early stages of the design. So, action execution time is estimated and modeled as a delay in UPPAAL. Finally, to the best of our knowledge, it is a novelty to involve both structural and dynamic aspects of a UML/MARTE model to obtain a network of timed automata. Indeed, the component diagram defines the topology of the network, while the activity diagrams induce each automaton composing the network. Moreover, arranging the system model in views allows to reduce verification complexity and analyze system properties independently (e.g., communication).

2.6 Conclusion

This chapter presented the first two steps of the V&V-based HW/SW Co-Design. During the first step, i.e., Co-Specification, we exploited UML/MARTE to model the functional and non-functional requirements of the system. The UML/MARTE model is made by three views: Application, Communication, and Time. We introduced a new verification and validation step, namely Co-V&V. Co-V&V uses the UPPAAL model checker for functional verification and timing validation of the system. To use UPPAAL, Co-V&V includes an M2M transformation from UML/MARTE to a network of timed automata. We showed Co-Specification and Co-V&V through the FIRGCD toy example. FIRGCD is characterized by a component-based architecture where each component runs asynchronously at the reception of data or signals. After the transformation into timed automata, we used UPPAAL to check the reachability of the `displayed` state, the correctness of the communication protocol, and the satisfiability of the timing constraints. The Co-V&V showed how reachability verification can be beneficial to improve the structure of the system. Indeed, we achieved a higher probability value of reaching the `displayed` location eliminating the two connectors linking the FIR filters to the GCD component. We proved that the *global_constraint* is not satisfiable with the initial design of FIRGCD. The validation revealed that the constrained execution chunk is mostly affected by the delay of the timer and the duration of the operations of the two FIR and poorly influenced by the structure of FIRGCD. The transformation, at the current stage, is performed manually. Consequently, the designers need extensive knowledge of the semantics of UML/MARTE and Timed Automata elements, UPPAAL, and CTL formulae. We plan to automate the transformation and provide the designers with a framework to build CTL formulae. We hypothesize that, in this way, we can reduce the effort needed to perform verification and validation. However, this hypothesis should be confirmed or disproved by conducting a user study. Moreover, we plan to examine case studies with more stringent non-functional requirements (e.g., real-time constraints [141]) or more complex behavior (e.g., self-adaptive [73]). Finally, it will be necessary to study how to associate UPPAAL time units with more concrete metrics such as milliseconds.

Chapter 3

Many-objective optimization of non-functional attributes based on refactoring of UML models

As stated in the Introduction, the complexity of HW/SW systems is growing and, as a result, techniques to tackle complexity are becoming increasingly important. Model-Driven approaches are widely used in different domains to reduce complexity at design time [76, 118, 134, 192]. Indeed, models can be employed to evaluate a subset of system aspects and optimize candidate design solutions. For example, to verify the functional correctness, and to estimate performance and reliability of design solutions. The main goal of this thesis is to integrate such analyses before the Design Space Exploration (DSE) of an HW/SW Co-Design flow. Integration is possible exploiting model-to-model transformations in which a UML/MARTE model is converted to different formalisms for V&V. For example, the initial model can be converted into performance models and validate performance constraints using a simulator or a solver. Therefore, the insights gathered during validation are used to improve the initial model. Thus, the optimization of the initial model mainly involves 3 stages: (i) transformation to a target formalism, (ii) analysis, and (iii) refactoring of the initial model. In this chapter, we describe Co-Analysis, which aims to automate the optimization of the initial model. The idea consists of applying a set of refactoring actions to the initial model to generate design alternatives. Therefore, evaluate the generated alternatives according to performance and reliability. This process can be seen as a search problem that aims at finding the design solutions that improve performance and reliability.

Search-based techniques have been widely used for software optimization [36, 110, 128, 154, 155, 163, 164], and they have proven to suit within the non-functional analysis due to the quantifiable nature of non-functional attributes [18, 19, 130]. Among the search-based techniques, those related to multi-objective optimization have been recently applied to model refactoring optimization problems [63, 149]. A common aspect of multi-objective optimization approaches applied to model-based refactoring problems is that they search among design alternatives (*e.g.,* through architectural tactics [111, 149]).

In this chapter, we present an approach based on a many-objective evolutionary algorithm (*i.e., NSGA-II* [77]) that searches sequences of refactoring actions, to be applied on models, leading to the optimization of four objectives: i) performance variation (analyzed through Layered Queuing

Networks [148]), ii) reliability (analyzed through a closed-form model [71]), iii) number of performance antipatterns (automatically detected [28]) and iv) architectural distance [26]. A performance antipattern is a bad design decision that might lead to a performance degradation [175, 176].[1] In particular, we analyze the composition of model alternatives generated through the application of refactoring actions to the initial model, and we analyze the contribution of the architectural distance to the generation of Pareto frontiers. Furthermore, we study the impact of performance antipatterns on the quality of refactoring solutions. Since it has been shown that removing performance antipatterns leads to systems that show better performance than the ones affected by them [28, 176, 177], we aim at studying if this result persists in the context of many-objective optimization, where performance improvement is not the only objective.

Our approach applies to UML models augmented by MARTE [97] and DAM [43] profiles that allow to embed performance and reliability properties. However, UML does not provide native support for performance analysis, thus we introduce a model-to-model transformation that generates Layered Queueing Networks (LQN) from annotated UML models. The solution of LQN models feeds the performance variation objective. It is worth recalling that the models considered are technology-independent. This means that any refactoring concerns the behavior or logical structure modeled. This model is refined in the later stages of HW/SW Co-Design.

Here, we consider refactoring actions that are designed to improve performance in most cases. Since such actions may also have an impact on other non-functional properties, we introduce the reliability among the optimization objectives to study whether satisfactory levels of performance and reliability can be kept at the same time. In order to quantify the reliability objective, we adopt an existing model for component-based software systems [71] that can be generated from UML models. This reliability model originated for software systems, but it can also be applied to technology independent models since it takes into account the elements of the UML model (such as the number of communication links).

We also minimize the distance between the initial UML model and the ones resulting from applying refactoring actions. Indeed, without an objective that minimizes such distance, the proposed solutions could be impractical because they could require to completely disassemble and re-assemble the initial UML model.

In a recent work [65], we extended the approach in [26, 63], by investigating UML models optimization, thus widening the scope of eligible models. In this chapter, we extensively apply the approach to two illustrative examples from the literature: Train Ticket Booking Service [80, 203], and CoCoME [104]. In spite of the fact that these examples fall into the software domain, we exploit their specification to create behavioral models which do not include implementation details. Therefore, during DSE, it can be evaluated if their behavior or part of it could be also implemented as HW components. We analyze the sensitivity of the search process to configuration variations. We refine the cost model of refactoring actions, introduced in [65], and we investigate how it contributes to the generation of Pareto frontiers. Also, we analyze the characteristics of computed Pareto frontiers in order to extract common properties for both examples.

This study answers the following research questions:

- *RQ1*: To what extent do experimental configurations affect quality of Pareto frontiers?

[1]We provide more detail in 3.1.

  - *RQ1.1*: How antipattern detection contribute to find better solutions compared to the case where antipatterns are not considered at all?
  - *RQ1.2*: How the probabilistic nature of fuzzy antipatterns detection help to include higher quality solutions in Pareto frontiers with respect to deterministic one?
  - *RQ1.3*: To what extent does the architectural distance contribute to find better alternatives?

- *RQ2*: Is it possible to increase reliability without performance degradation?
- *RQ3*: What type of refactoring actions are more likely to lead to better solutions?

The experimentation lasted approximately *200* hours and generated more than *70,000* model alternatives.

Generally, multi-objective optimization is beneficial when the solution space is so large that an exhaustive search is impractical. Hence, due to the search of the solution space, multi-objective optimization requires a lot of time and resources.

Our results show that, by considering the reduction of performance antipatterns as an objective, we are able to obtain model alternatives that show better performance and, in the majority of cases, better reliability as well. We also find that a more sophisticated architectural distance objective estimation helps the optimization process to generate model alternatives showing better quality indicators. Also, we strengthen the idea that performance antipatterns are promising proxies of performance degradation of models. Finally, to encourage reproducibility, we publicly share the implementation of the approach [2], as well as the data gathered during the experimentation [3].

The structure of the chapter is the following: Section 3.1 introduces basic concepts, Section 3.2 describes the approach, Section 3.3 describes the two involved illustrative examples, and Section 3.4 details used configurations, in Section 3.5 we evaluate our approach and discuss the results, threats to validity are described in Section 3.6, Section 3.7 reports related work, and Section 3.8 concludes the chapter.

3.1 Background

We identify four competing objectives of our evolutionary approach as follows: *perfQ* is a performance quality indicator that quantifies the performance improvement/detriment between an initial model and one obtained by applying the refactoring actions of a solution (Section 3.1); *reliability* is a measure of the reliability of the model (Section 3.1); *performance antipatterns* is a metric that quantifies the amount of performance antipattern occurrences while considering the intrinsic uncertainty rising from thresholds used by the detection mechanism (Section 3.1); *#changes* represents the distance between an initial model and one obtained by applying the refactoring actions of a solution (Section 3.1).

We employ the Non-dominated Sorting Algorithm II (NSGA-II) as our genetic algorithm [77], since it is extensively used in the software engineering community, *e.g.,* [111, 127]. *NSGA-II* randomly creates an initial population of model alternatives, and it used to create the offspring population by applying the *Crossover* with probability $P_{crossover}$, and the *Mutation* with probability

[2]`https://github.com/SEALABQualityGroup/EASIER`

[3]`https://github.com/SEALABQualityGroup/2022-ist-replication-package`

$P_{Mutation}$ operators. The union of the initial and the offspring populations is sorted by the *Non-dominated sorting* operator, which identifies different Pareto frontiers with respect to considered objectives. Finally, the *Crowding distance* operator cuts off the worse half of the sorted union population. Hence, the remaining model alternatives become the initial population for the next step.

Performance Quality Indicator (*perfQ*)

perfQ quantifies the performance improvement/detriment between two models, and it is defined as follows:

$$perfQ(M) = \frac{1}{c}\sum_{j=1}^{c} p_j \cdot \frac{F_j - I_j}{F_j + I_j}$$

where M is a model obtained by applying a refactoring solution to the initial model, F_j is the value of a performance index in M, and I_j is the value of the same index on the initial model. $p \in \{-1, 1\}$ is a multiplying factor that holds: i) 1 if the j–th index has to be maximized (i.e., the higher the value, the better the performance), like the throughput; ii) -1 if the j–th index has to be minimized (i.e., the smaller the value, the better the performance), like the response time.

Notice that, for performance measures representing utilization, p also holds 1 but we define a *utilization correction factor* Δ_j to be added to each j–th term above, as defined in [26]. The utilization correction factor penalizes refactoring actions that push the utilization too close to 1, i.e., its maximum value. Finally, the global *perfQ* is computed as the average across the number c of performance indices considered in the performance analysis.

As mentioned in the introduction, in order to obtain performance indices of a UML model, the analysis has been conducted on Layered Queueing Networks (LQNs) [148][4] that are obtained through a model transformation approach from UML to LQN, which we have introduced in [65]. We chose Layered Queueing Networks as our performance model notation because it is extensively used in the literature and it allows a more explicit representation of software and hardware components (and their interactions) than the one of conventional Queueing Networks [111, 115, 149].

Reliability model

The reliability model that we adopt here to quantify the *reliability* objective is based on the model introduced in [71]. The mean failure probability θ_S of a software system S is defined by the following equation:

$$\theta_S = 1 - \sum_{j=1}^{K} p_j \left(\prod_{i=1}^{N} (1-\theta_i)^{InvNr_{ij}} \cdot \prod_{l=1}^{L} (1-\psi_l)^{MsgSize(l,j)} \right)$$

This model takes into account failure probabilities of components (θ_i) and communication links (ψ_l), as well as the probability of a scenario to be executed (p_j). Such probabilities are combined to obtain the overall reliability on demand of the system (θ_S), which represents how often the system is not expected to fail when its scenarios are invoked.

The model is considered to be composed of N components and L communication links, whereas its behavior is made of K scenarios. The probability (p_j) of a scenario j to be executed is multiplied

[4]`http://www.sce.carleton.ca/rads/lqns/LQNSUserMan-jan13.pdf`

by an expression that describes the probability that no component or link fails during the execution of the scenario. This expression is composed of two terms: $\prod_{i=1}^{N}(1-\theta_i)^{InvNr_{ij}}$, which is the probability of the involved components not to fail raised to the power of their number of invocations in the scenario (denoted by $InvNr_{ij}$), and $\prod_{l=1}^{L}(1-\psi_l)^{MsgSize(l,j)}$, which is the probability of the involved links not to fail raised to the power of the size of messages traversing them in the scenario (denoted by $MsgSize(l,j)$).

Performance Antipatterns

A performance antipattern describes bad design practices that might lead to performance degradation in a system. Smith and Williams have introduced the concepts of performance antipatterns in [176, 178]. These textual descriptions were later translated into a First-Order Logic (FOL) equations [69].

Performance antipattern	Description
Pipe and Filter	Occurs when the slowest filter in a "pipe and filter" causes the system to have unacceptable throughput.
Blob	Occurs when a single component either i) performs the greatest part of the work or ii) holds the greatest part of the data. Either manifestation results in excessive message traffic that may degrade performance.
Concurrent Processing System	Occurs when processing cannot make use of available processors.
Extensive Processing	Occurs when extensive processing in general impedes overall response time.
Empty Semi-Truck	Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.
Tower of Babel	Occurs when processes use different data formats and they spend too much time in convert them to an internal format.

Table 3.1: Detectable performance antipatterns in our approach. Left column lists performance antipattern names, while right column lists performance antipattern descriptions [177].

A performance antipattern FOL is a combination of multiple literals, where each one represents a system aspect (*e.g.,* the number of connections among components). These literals must be compared to thresholds in order to reveal the occurrence of a performance antipattern. The identification of such thresholds is a non-trivial task, and using deterministic values may result in an excessively strict detection where the smallest change in the value of a literal determines the occurrence of the antipattern. For these reasons, we employ a fuzzy detection [30], which assigns to each performance antipattern a probability to be an antipattern. An example of a performance antipattern fuzzy detection is the following:

$$1-\frac{UB(literal)-literal}{UB(literal)-LB(literal)}$$

The upper (UB) and the lower (LB) bounds, in the above equation, are the maximum and minimum values of the *literal* computed on the entire system. Instead of detecting a performance antipattern in a deterministic way, such thresholds lead to assign probabilities to antipattern occurrences. In this study, we detect the performance antipatterns listed in Table 3.1.

Architectural distance

The architectural distance, that we express here as *#changes*, represents the distance of the model obtained by applying refactoring actions from the initial one [26]. On one side, a Baseline Refactoring Factor (BRF) is associated to each refactoring action in our portfolio, and it expresses the refactoring effort to be spent when applying the action. On the other side, an Architectural Weight (AW) is associated to each model element on the basis of the number of connections to other elements in the model. Hence, we quantify the effort needed to perform a refactoring as the product between the *baseline refactoring factor* of an action and the *architectural weight* of the model element on which that action is applied. *#changes* is obtained by summing the efforts of all refactoring actions contained in a solution.

Furthermore, *BRF* and AW can assume any positive value (*i.e.,* zero is a non-admitted value because it would lead the optimizer to always select only actions by that type).

As an example, let us assume that a refactoring sequence is made up of two refactoring actions: A1 with $BRF(A1) = 1.23$, and A2 with $BRF(A2) = 2.3$. For each refactoring action, the algorithm randomly selects a target element in the model. For instance, let those target elements be: E1 with $AW(E1) = 1.43$, and E2 with $AW(E2) = 1.32$. The resulting *#changes* of A1 and A2 would be:

$$\#changes(A1, A2) = 1.23 \cdot 1.43 + 2.3 \cdot 1.32$$

Details about the *baseline refactoring factor* for each considered refactoring action are provided in Section 3.2.

3.2 Approach

Figure 3.1 depicts the process we present in this paper. The process uses a UML model and a set of refactoring actions as input. The *Initial Model* and the *Refactoring Actions* are involved within the *Create Combined Population* step, where mating operations (*i.e.,* selection, mutation, and crossover) are put in place to create *Model Alternatives.* The mating operations randomly apply the refactoring actions, which generate alternatives functionally equivalent to the initial model. Therefore, the *Evaluation* step is applied to each model alternative. Subsequently, the model alternatives are ranked (*Sorting* step) according to four objectives: *perfQ*, *reliability*, *#changes*, and *performance antipatterns*. The optimal model alternatives (*i.e.,* non-dominated alternatives) become the input of the next iteration. The process continues until the stopping criteria are met. Finally, the process generates a *Pareto Frontier*, which contains all non-dominated model alternatives.

Assumptions on UML models

In our approach, we consider UML models including three views, namely *static, dynamic* and *deployment* views. The static view is modeled by a UML Component diagram in which static connections among components are represented by interface realizations and their usages. The

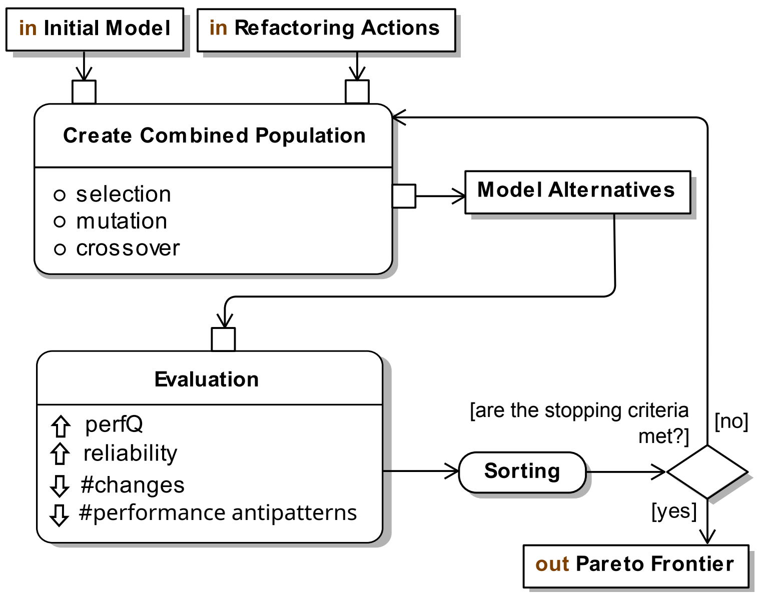


Figure 3.1: Our multi-objective evolutionary approach

dynamic view is described by UML Use Case and Sequence diagrams. A Use Case diagram defines user scenarios, while a Sequence diagram describes the behavior inside a single scenario through component operations (as defined in their interfaces) and interactions among them. A Deployment diagram is used to model platform information and map Components to Deployment Nodes. As mentioned before, we use an augmented UML notation by embedding two existing profiles, namely MARTE [97] that expresses performance concepts, and DAM [43] that expresses reliability concepts.

Although our assumptions on UML models seem to require an upfront modeling phase, the accuracy of results is affected by the quality of model and annotations. We mitigate the modeling effort through the usage of UML. In fact, a plethora of UML modeling tools is available, each equipped with entry-level or advanced capabilities that differently help models design.[5]

The Refactoring Engine

The automated refactoring of UML models is a key point when evolutionary algorithms are employed in order to optimize some model attributes. For the sake of full automation of our approach, we have implemented a refactoring engine that applies refactoring actions on UML models [29].

Each solution that our evolutionary algorithm produces is a sequence of refactoring actions that, once applied to an initial model, leads to a model alternative that shows different non-functional properties. Since our refactoring actions are combined during the evolutionary approach, we exploit the feasibility engine that verifies in advance whether a sequence of refactoring actions is feasible or not [27].

Our refactoring actions are equipped with pre- and post-condition. While the pre-condition represents the model state for enabling the action, the post-condition represents the model state when the action has been applied. The approach extracts a refactoring action and adds it to the sequence. As soon as the action is selected, it randomly extracts a model element (*i.e.,* the target element). Thus, the refactoring engine checks the feasibility of the (partial) sequence of refactoring

[5]`https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools`

actions.When the latest added action makes the sequence unfeasible, the engine discards the action and replaces it with a new one. The engine reduces a sequence of refactoring actions to a single refactoring action, which includes all the changes (see Equation (3.1a)).

For example, considering two refactoring actions (M_i, and M_j), then the global pre-condition is obtained by logical ANDing the first action pre-condition (${}^{Pr}M_i$) and all the parts of M_j pre-condition that are not yet verified by M_i post-conditions (M_j^{Pr} / ${M_i}^{Po}$) (see Equations (3.1b)). Since the status of the model after a refactoring is synthesized by its post-condition, we can discard the parts of a subsequent refactoring pre-condition that, by construction, are already verified by its post-condition. The global post-condition is obtained by logical ANDing all post-conditions within the sequence ($M_i^{Po} \wedge M_j^{Po}$) (see Equation (3.1c)).

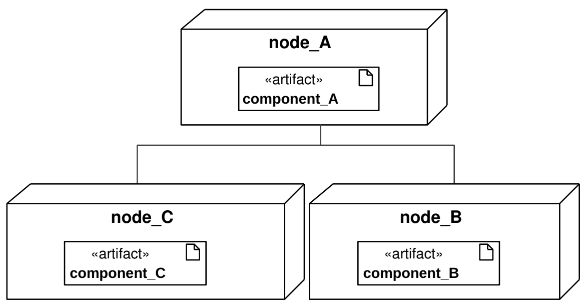
$$\begin{align} {}^{Pr}M_i^{Po} \wedge {}^{Pr}M_j^{Po} &\longmapsto {}^{Pr}M^{Po} \tag{3.1a}\\ {}^{Pr}M_i \wedge M_j^{Pr} \;/\; {M_i}^{Po} &\longmapsto {}^{Pr}M \tag{3.1b}\\ M_i^{Po} \wedge M_j^{Po} &\longmapsto M^{Po} \tag{3.1c} \end{align}$$

Our feasibility engine also allows to reduce the number of invalid refactoring sequences, thus reducing the computational time.

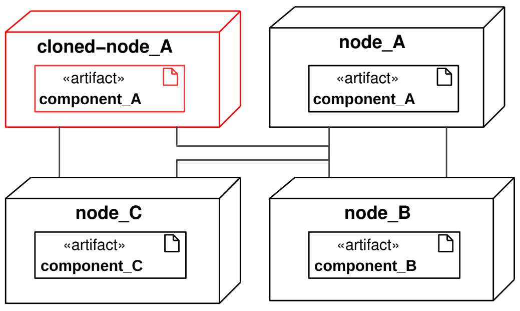
Refactoring Action portfolio

Figure 3.2 through Figure 3.5 show a graphic representation of each refactoring action. Each figure's left side shows the original model (*e.g.,* static view in Figure 3.3a, dynamic view in Figure 3.3c, and deployment view in Figure 3.3e), while the refactored version is shown on the right side (*e.g.,* static view in Figure 3.3b, dynamic view in Figure 3.3d, and deployment view in Figure 3.3f). The red highlights indicate changes.

Clone a Node (Clon) This action is aimed at introducing a replica of a Node. Adding a replica means that every deployed artifact and every connection of the original Node has to be in turn cloned. Stereotypes and their tagged values are cloned as well. The rationale of this action is to introduce a replica of a platform device with the aim of reducing its utilization.



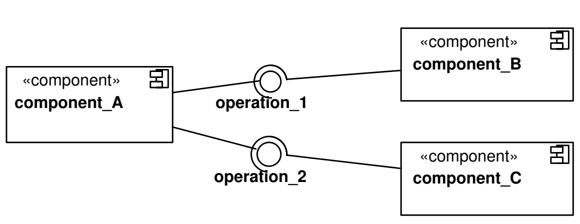
(a) Initial



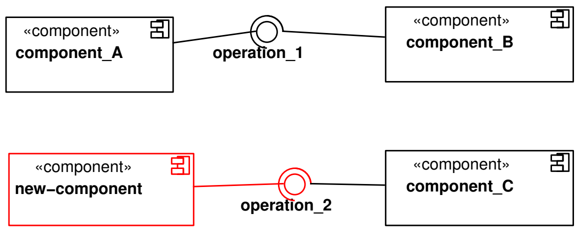
(b) Refactored

Figure 3.2: The *Clon* refactoring action example on *node_A* through a UML Model

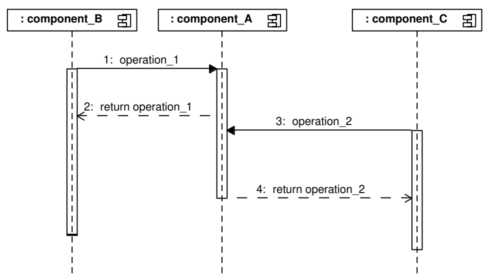
Move an Operation to a new Component deployed on a new Node (MO2N) This action is in charge of randomly selecting an operation and moving it to a new Component. All the elements related to the moving operation (*e.g.,* links) will move as well. Since we adopt a multi-view model, and coherence among views has to be preserved, this action has to synchronize dynamic and deployment views. A lifeline for the newly created Component is added in the dynamic view, and messages related to the moved operation are forwarded to it. In the deployment view, instead, a new Node, a new artifact, and related links are created. The rationale of this action is to lighten the load of the original Component and Node.



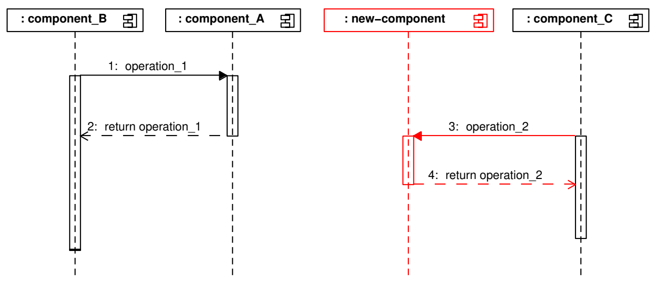
(a) Initial



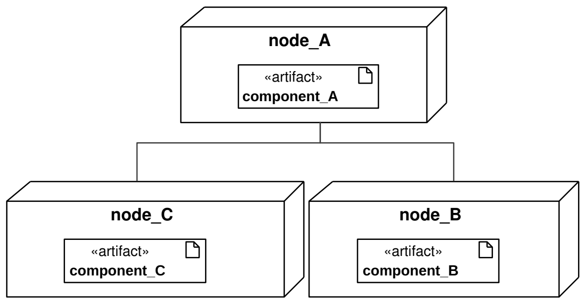
(b) Refactored



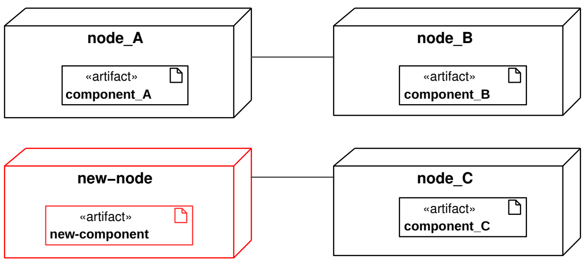
(c) Initial



(d) Refactored



(e) Initial

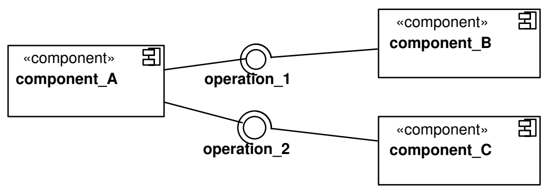


(f) Refactored

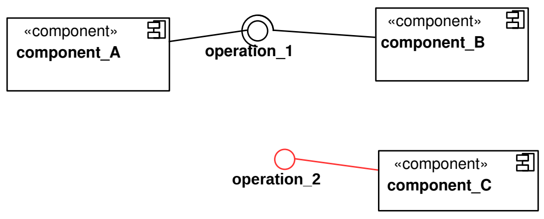
Figure 3.3: The *MO2N* refactoring action example on *operation_ 2* through a UML Model

Move an Operation to a Component (MO2C) This action is in charge of randomly selecting and transferring an Operation to an arbitrary existing target Component. The action consequently modifies each UML Use Case in which the Operation is involved. Sequence Diagrams are also updated to include a new lifeline representing the Component owning the Operation, but also to re-assign the messages invoking the operation to the newly created lifeline. The rationale of this action is quite similar to the previous refactoring action, but without adding a new UML Node to the model.

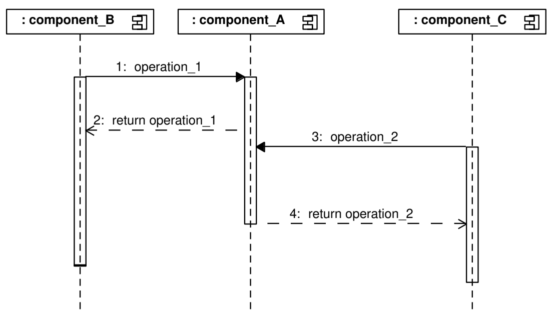
Deploy a Component on a new Node (ReDe) This action simply modifies the deployment view by redeploying a Component to a newly created Node. In order to be consistent with the initial



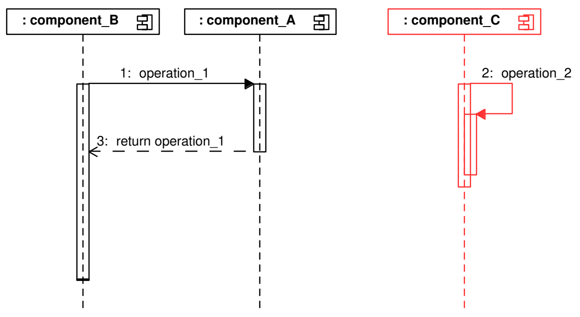
(a) Initial



(b) Refactored



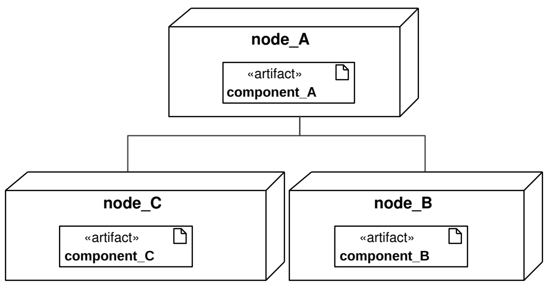
(c) Initial



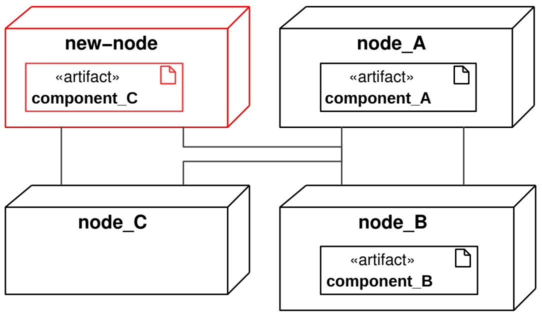
(d) Refactored

Figure 3.4: The *MO2C* refactoring action example on *operation_ 2* and *component_ C* through a UML Model

model, the new Node is connected with all other ones directly connected to the Node on which the target Component was originally deployed. The rationale of this action is to lighten the load of the original UML Node by transferring the load of the moving Component to a new UML Node. The ReDe and Clon actions have a similar rationale, i.e., to redirect the load to a new deployment node, but they differ on the type of node on which they move the load. A Clon action transfers the load to a new node, which is offloaded and created specifically for the operation. Instead, a ReDe action targets existing nodes that may already be handling other loads.



(a) Initial



(b) Refactored

Figure 3.5: The *ReDe* refactoring action example on *component_ C* through a UML Model

Baseline Refactoring Factor

As described in Section 3.1, we measure the architectural distance by summing the products of baseline refactoring factor (BRF) and architectural weight (AW) for each refactoring action $a_i(el_j)$

Action	*BRF*	TTBS	CoCoME
MO2N	1.80	70	$\approx 4.8 \times 10^3$
MO2C	1.64	$\approx 1.5 \times 10^6$	$\approx 1.3 \times 10^8$
ReDe	1.45	$\approx 3 \times 10^2$	$\approx 7 \times 10^2$
Clon	1.23	$\approx 3 \times 10^2$	70
Ω		9.45×10^{12}	3.05×10^{16}

Table 3.2: A detailed size of the solution space (Ω) computation.

within a sequence ($\mathbb{A}$).

$$\#changes(\mathbb{A}) = \sum_{a_i(el_j) \in \mathbb{A}} BRF(a_i) \times AW(el_j)$$

AW is the weight of the target of the refactoring action, *BRF* is the intrinsic cost that one should pay in order to apply the specific action on a model element. There are different ways to compute the effort for implementing artefacts or maintaining them (*e.g.,* COCOMO-II [45], and CoBRA [186]). Nevertheless, we consider the cost in terms of the effort that one should spend on the model to complete a refactoring action, and we assign *BRF* values on the basis of our past experience in manual refactoring. We have not used a cost estimator model, such as CoBRA, because it requires to collect business information that is not available for non-industrial case studies. Table 3.2 lists the *BRF* values used in this study. It is worth remarking that, in our optimization problem, the ratio among *BRF* values is more important than how each single value has been extracted.

Computing reliability on UML models

The reliability parameters of the model introduced in Section 3.1 are annotated on UML models by means of the MARTE-DAM profile. The probability of executing a scenario (p_j) is specified by annotating UML Use Cases with the *GaScenario* stereotype. This stereotype has a tag named *root* that is a reference to the first *GaStep* in a sequence. We use the *GaScenario.root* tag to point to the triggering UML Message of a Sequence Diagram and the *GaStep.prob* to set the execution probability. Failure probabilities of components (θ_i) are defined by applying the *DaComponent* stereotype on each UML Component and by setting, in the *failure* tag, a *DaFailure* element with the failure probability specified in the *occurrenceProb* tag. Analogously, failure probabilities of links (ψ_l) are defined in the *failure.occurrenceProb* tag of the *DaConnector* stereotype that we apply on UML CommunicationPath elements. Such elements represent the connection links between UML Nodes in a Deployment Diagram. Sequence Diagrams are traversed to obtain the number of invocations of a component i in a scenario j (denoted by $InvNr_{ij}$ in our reliability model), but also to compute the total size of messages passing over a link l in a scenario j (denoted by $MsgSize(l, j)$). The size of a single UML Message is annotated using the *GaStep.msgSize* tag. The Java implementation of the reliability model is available online.[6]

[6] `https://github.com/SEALABQualityGroup/uml-reliability`

	Configuration	Eligible values
Experiment settings	Baseline Refactoring Factor	no, yes
	Performance Antipattern fuzziness	0.55, 0.80, 0.95
	Illustrative Examples	`TTBS`, `CoCoME`
NSGA-II	Number of genetic evolutions	72, 82, 102
	Population Size	16
	Number of independent runs	3
	Selection operator	Binary Tournament Selection
	$P_{crossover}$	0.80
	Crossover Operator	Single Point
	$P_{mutation}$	0.20
	Mutation Operator	Simple Mutation

Table 3.3: Eligible configuration values.

Pareto Frontier Quality Indicators

We compare the performance of the *NSGA-II* while varying the configuration eligible values listed in Table 3.3. We used well-established quality indicators also provided in the JMetal framework [147]. We use quality indicators to quantify the difference among computed Pareto Frontier (PF^c) with respect to the reference Pareto Frontier (PF^{ref}) [20]. Therefore, we can declare which configuration outperform the others.

In the following, we recall some characteristics for each quality indicator.

GSPREAD The Generalized SPREAD (GSPREAD) is a quality indicator to be minimized, and it measures the spread of solution within PF^c [202]. It is computed as follows:

$$GSPREAD(PF^c) = \frac{\sum_{i=1}^{m} d(e_i, PF^c) + \sum_{s \in PF^c} \left| id(s, PF^c) - \bar{id} \right|}{\sum_{i=0}^{m} d(e_i, PF^c) + |PF^c| * \bar{id}}$$

where e_i is the optimal value for the objective f_i, *i.e.,* $(e_1, \ldots, e_m)$ is the extreme solution in PF^{ref}, $id(s, PF^c) = d(s, PF^c \backslash \{s\})$ is the minimal distance of a solution s from the solutions in PF^c, and $\bar{id}$ is the mean value of $id(s, PF^c)$ across the solutions s in PF^{ref}.

IGD$^+$ The Inverse Generational Distance plus (IGD$^+$) is a quality indicator to be minimized. It measures the distance from a solution in PF^{ref} to the nearest solutions in PF^c [108]. It is computed as follows:

$$IGD^+(PF^c) = \frac{\sqrt{\sum_{s \in PF^{ref}} d(s, PF^c)^2}}{|PF^{ref}|}$$

Hypervolume The Hypervolume (HV) indicator is to be maximized and it measures the volume of the solution space Ω covered by PF^c [206]. It is computed as follows:

$$HV(PF^c) = volume(\cup_{s_i \in PF^c} hc(s_i))$$

where s_i is a solution within the PF^c, $hc(s_i)$ is the hypercube having s_i and w as diagonal points. The variable w is the reference point computed using the worst objective function values among all the possible solutions in PF^c.

EPSILON The EPSILON (EP) quality indicator measures the smallest distance that each solution within PF^c should be translated so that PF^c dominates PF^{ref} [207]. EPSILON is a quality indicator to be minimized, and it uses the notation of epsilon-dominance $\succ_\epsilon$. It is computed as follows:

$$EP(PF^c) = inf\{\epsilon \in \mathbb{R} | (\forall x \in PF^{ref}, \exists y \in PF^c : y \succ_\epsilon x)\}$$

In our study, we have computed a PF^{ref} for each illustrative example by extracting every non-dominated solutions across each PF^c, *i.e.,* one for each configuration. Hence, the quality indicators in Table 3.5 and Table 3.6 have been computed with respect to the PF^{ref} for the `TTBS`, and `CoCoME` illustrative example respectively.

3.3 Illustrative Examples

In this section, we apply our approach to Train Ticket Booking Service (TTBS) [80, 203], and to the well-established Common Component Modeling Example (CoCoME), whose UML model has been derived by the specification in [104].

3.3.1 Train Ticket Booking Service

Train Ticket Booking Service (`TTBS`) is a web-based booking application, whose architecture is based on the microservice paradigm. The system is made up of 40 microservices, and it provides different scenarios through users that can perform realistic operations, *e.g.,* book a ticket or watch trip information like intermediate stops. The application employs a docker container for each microservice, and connections among them are managed by a central pivot container.

Our UML model of `TTBS` is available online.[7] The static view is made of **11** UML Components, where each component represents a microservice. In the deployment view, we consider **11** UML Nodes, each one representing a docker container.

Among all `TTBS` scenarios shown in [80], in this chapter we have considered **3** UML Use Cases, namely *login*, *update user details* and *rebook.* We selected these three scenarios because they involve different TTBS components and operations. Consequently, we obtain heterogeneous sequences of refactoring actions since they are applied to diverse model elements. A refactoring action applied to different model elements can reveal whether the results are affected by that element type, that refactoring action, or a combination of both. Each scenario is described by a UML Sequence Diagram. Furthermore, the model comprises two user categories: simple and admin users. The simple user category can perform the login and the rebook scenarios, while the admin category can perform the login and the update user details scenarios.

[7] `https://github.com/SEALABQualityGroup/2022-ist-replication-package/tree/main/case-studies/train-ticket`

Illustrative Example	UML Node	UML Component	UML Message	Ω
`TTBS`	11	11	8	1.20×10^{13}
`CoCoME`	8	13	20	3.26×10^{16}

Table 3.4: Number of UML elements in our Illustrative Examples, and the size of the relative solution space (Ω).

3.3.2 `CoCoME`

The component-based system engineering domain has always been characterized by a plethora of standards for implementing, documenting, and deploying components. These standards are well-known as component models. Before the birth of the common component modeling example (`CoCoME`) [104], it was hard for researchers to compare different component models. `CoCoME` acts as a single specification to be implemented using different component models.

`CoCoME` describes a Trading System containing several stores. A store might have one or more cash desks for processing goodies. A cash desk is equipped with all the tools needed to serve a customer (e.g., a Cash Box, Printer, Bar Code Scanner). `CoCoME` covers possible scenarios performed at a cash desk (e.g., scanning products, paying by credit card, generating reports, or ordering new goodies). A set of cash desks forms a cash desk line. The latter is connected to the store server for registering cash desk line activities. Instead, a set of stores is organized in an enterprise having its server for monitoring stores operations.

`CoCoME` describes 8 scenarios involving more than 20 components. We have modeled `CoCoME` using UML and following the structure described in Section 3.2. From the `CoCoME` original specification, we analyzed different operational profiles, *i.e.,* scenarios triggered by different actors (such as Customer, Cashier, StoreManager, StockManager), and we excluded those related to marginal parts of the system, such as scenarios of the *EnterpriseManager* actor. Thus, we selected **3** UML Use Cases, **13** UML Components, and **8** UML Nodes from the `CoCoME` specification. Beside this, we focused on three scenarios, namely: UC1 that describes the arrival of a customer at the checkout, identification, and sale of a product; UC4 that represents how products are registered in the store database upon their arrival; UC5 that represents the possibility of generating a report of store activities.

We computed the size of the solution space (Ω) as the Cartesian product of the combination of refactoring actions $C_{n,k} = \binom{n}{k}$ where n is the number of target model elements, and k is the length of the chromosome (*i.e.,* the length of the sequence of refactoring actions, which is 4 in our case), and we summarize data in Table 3.2. We remark that a manual investigation of the solution space is unfeasible due to its size. Hence, the evolutionary search is helpful for looking for model alternatives showing better quality than the initial one. Table 3.4 summarizes characteristics of the illustrative examples.

3.4 Experimental setup

A configuration is defined by the combination of parameters related to the genetic algorithm, and the ones related to the specific optimization model. The eligible configuration values in our approach are listed in Table 3.3. In order to investigate which configuration produces better Pareto frontiers, we have executed multiple tuning runs to find a set of optimal configurations.

In order to set the parameters related to the genetic algorithm, we have performed a tuning phase with the intent of increasing the quality of the Pareto frontiers. In particular, we have set the length of refactoring sequences to four actions, which represents a good approximation of the number of refactoring actions usually applied by a designer in a single session. We have set the $P_{crossover}$ and $P_{mutation}$ probabilities to 0.8 and 0.2, respectively, following common configurations [32]. The higher the values of these two probabilities, the greater the chance of generating an unfeasible sequence of refactoring actions, which in turn causes a longer simulation time due to a higher number of discarded sequences. For example, the $P_{crossover}$ increase could cause a lot of permutation among sequences, and it might lead to wrong or unfeasible sequences of refactoring actions.

The initial population size might drive the genetic algorithm in local minima, and thus result in stagnant solutions. In general, a densely populated initial population minimizes the probability of stagnant solutions in local minima. However, the generation of a crowded initial population is computational demanding and, in case of rare local minima, the computational cost represents a clear slowdown for the evolutionary approach [31]. For that reason, we set the population size to **16** elements (i.e., 16 different UML model alternatives), which did not show stagnant issues in our tuning phase. Furthermore, we will investigate in a future work the impact of denser populations in our analysis, in terms of computational time and quality of the computed Pareto frontiers (PF^c). In addition, multiple runs have been executed for each configuration in order to reduce the randomness of the genetic algorithm.

We considered three fuzziness thresholds, *i.e.,* {0.55, 0.80, 0.95}, to study the impact of performance antipatterns on computed Pareto frontiers. Since we are considering a fuzzy detection of performance antipatterns, we should use values greater than 50% to reduce the probability of false positives, but less than 100% to not fall in a case of performance antipatterns deterministic detection. Therefore, we decided to use those three fuzziness values to analyze the uncertainty of a fuzzy performance antipatterns detection.

With regard to parameters related to refactoring actions, we ran the experiment twice, one by excluding BRF, and one by including it. For the latter, we set BRF of each refactoring action as reported in Table 3.2. As we said in Section 3.2, we did not employ a complex cost model for baseline refactoring factor values. However, we remark that we are interested in the ratio between BRF values rather than in their specific values, and we will deeply investigate the impact of other values on future work.

Our experimental settings on `TTBS` and `CoCoME` have generated *70,000* model alternatives and have taken *200* hours of computation. We performed our experiments on a server equipped with two Intel Xeon E5-2650 v3 CPUs at 2.30GHz, 40 cores and 80GB of RAM.

3.5 Results and discussion

Results presented in this section are aimed at answering the aforementioned three research questions.

3.5.1 RQ1

RQ1: To what extent do experimental configurations affect quality of Pareto frontiers?

BRF	*maxeval*	*probpas*	q_indicator	value
yes	72	95	HV	0.329645
yes	82	95	HV	0.304931
yes	82	95	HV	0.267898
yes	72	80	HV	0.266588
yes	82	55	HV	0.254973
yes	72	95	IGD$^+$	0.135226
yes	82	95	IGD$^+$	0.149903
yes	72	55	IGD$^+$	0.157150
yes	82	95	IGD$^+$	0.167142
yes	72	80	IGD$^+$	0.173162
yes	72	95	EP	0.295681
yes	82	95	EP	0.296014
yes	82	95	EP	0.316964
yes	72	55	EP	0.316964
yes	82	55	EP	0.323661
yes	102	55	GSPREAD	0.125487
yes	102	95	GSPREAD	0.127085
yes	102	80	GSPREAD	0.144666
yes	102	55	GSPREAD	0.148802
yes	72	55	GSPREAD	0.203504

Table 3.5: Best five of each quality indicator for Train Ticket Booking Service while varying the performance antipattern fuzziness and the genetic algorithm evolutions.

RQ1 focuses on the contribution of experimental configurations to the quality of the computed Pareto frontiers (PF^c).

In Table 3.5 and Table 3.6 it is possible to observe the configurations that result in better Pareto frontiers. Generally, quality indicators are obtained with respect to the optimal reference Pareto frontier (PF^{ref}), and each one has its ideal value (*e.g.,* $HV = 1$, $IDG^+ = 0$). Moreover, values in tables have been sorted in ascending order when the best quality indicator is the lowest one, and in descending order otherwise. Since we did not have the optimal PF^{ref} for our illustrative examples, we computed, for each example, the quality indicators with respect to a PF^{ref} that contains every non-dominated solution across all PF^c. Once quality indicators have been obtained and sorted, we identify which *maxeval* and *probpas* have generated better indicators. Finally, we also report data about *BRF*.

At a glance, we can see that in most cases for both illustrative examples, *maxeval* = 72 and lower fuzziness generates better quality indicators, whereas *BRF* has a different impact on the two illustrative examples.

In the following, we split *RQ1* into three sub-questions, each one related to a specific experimental configuration attribute. *RQ1.1* analyzes the influence of performance antipatterns on PF^c. *RQ1.2* investigates whether the fuzziness of performance antipattern detection helps to find better PF^c. *RQ1.3* studies the contribution of *BRF* to the quality of PF^c.

RQ1.1

RQ1.1: How antipattern detection contribute to find better solutions compared to the case where antipatterns are not considered at all?

In order to answer this research question, we have conducted an additional experimentation for

BRF	*maxeval*	*probpas*	q_indicator	value
no	72	95	HV	0.360432
no	82	95	HV	0.359415
no	102	95	HV	0.342563
no	72	55	HV	0.326384
no	82	95	HV	0.305201
no	72	95	IGD+	0.091767
no	82	95	IGD+	0.105173
no	102	95	IGD+	0.106406
no	82	95	IGD+	0.132800
no	72	55	IGD+	0.135904
no	82	95	EP	0.250000
no	72	55	EP	0.250000
no	72	95	EP	0.250000
no	82	95	EP	0.313857
yes	72	95	EP	0.333333
no	82	55	GSPREAD	0.145989
yes	102	55	GSPREAD	0.193488
yes	102	95	GSPREAD	0.196790
no	102	55	GSPREAD	0.200320
no	102	80	GSPREAD	0.203431

Table 3.6: Best five of each quality indicator for CoCoME while varying the performance antipattern fuzziness and the genetic algorithm evolutions.

every problem configuration, where we have removed performance antipattern occurrences from the fitness function, thus reducing the optimization to the remaining three objectives.

Train Ticket Booking Service Figure 3.6 depicts the Pareto frontiers of 72 genetic evolutions while considering the lowest fuzziness (*i.e., probpas* $= 0.95$) and no performance antipatterns (*i.e., probpas* $= 0$). We can see that frontiers with performance antipatterns are generally more densely populated than the case where *probpas* $= 0$. Also, performance antipatterns help finding model alternatives showing lower *#changes* than the ones found when they have been ignored. Although *probpas* $= 0$ generates the highest value of *perfQ* (*i.e., perfQ* $= 0.24$), there are more solutions in the topmost part of the plot when performance antipatterns drive the search process. From our analysis, it emerges that *probpas* $= 0.95$ produces better frontiers among those with performance antipatterns. Therefore, we can state that, for `TTBS`, the lower fuzziness the better the quality of frontiers in terms of *perfQ*, *reliability*, and *#changes*.

CoCoME Figure 3.7 depicts the Pareto frontiers with 72 genetic evolutions while considering the lowest fuzziness (*i.e., probpas* $= 0.95$) and no performance antipatterns (*i.e., probpas* $= 0$). Most of the solutions lay in the topmost part of the plot, thus meaning that PF^c shows better *perfQ* and *reliability* of the initial solution (see the black cross in the figure). Frontiers generated by performance antipatterns are more densely populated than those without performance antipatterns. Thus, the reduction of the number of performance antipatterns occurrences, if it is included among the objectives, helps the process finding more alternative models showing higher *perfQ* and *reliability* with lower *#changes*.

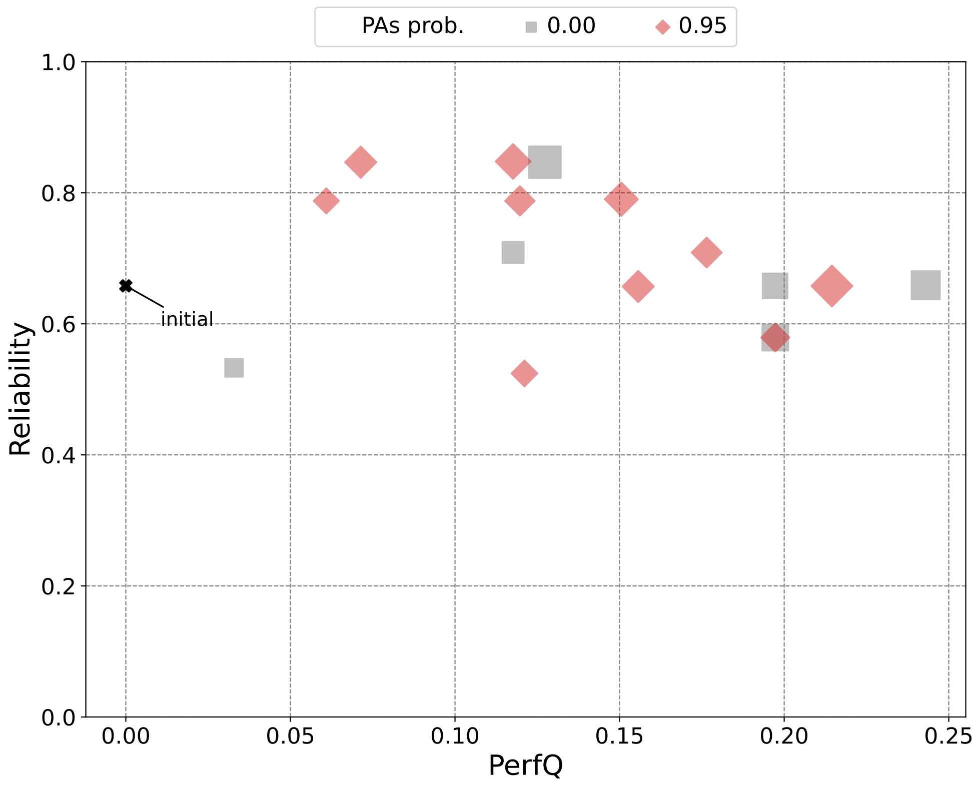


Figure 3.6: The scatter plot of PF^c of `TTBS` with 72 genetic evolutions while considering, and excluding performance antipatterns in the optimization process (*i.e.*, $probpas = 0.95$, and $probpas = 0$)

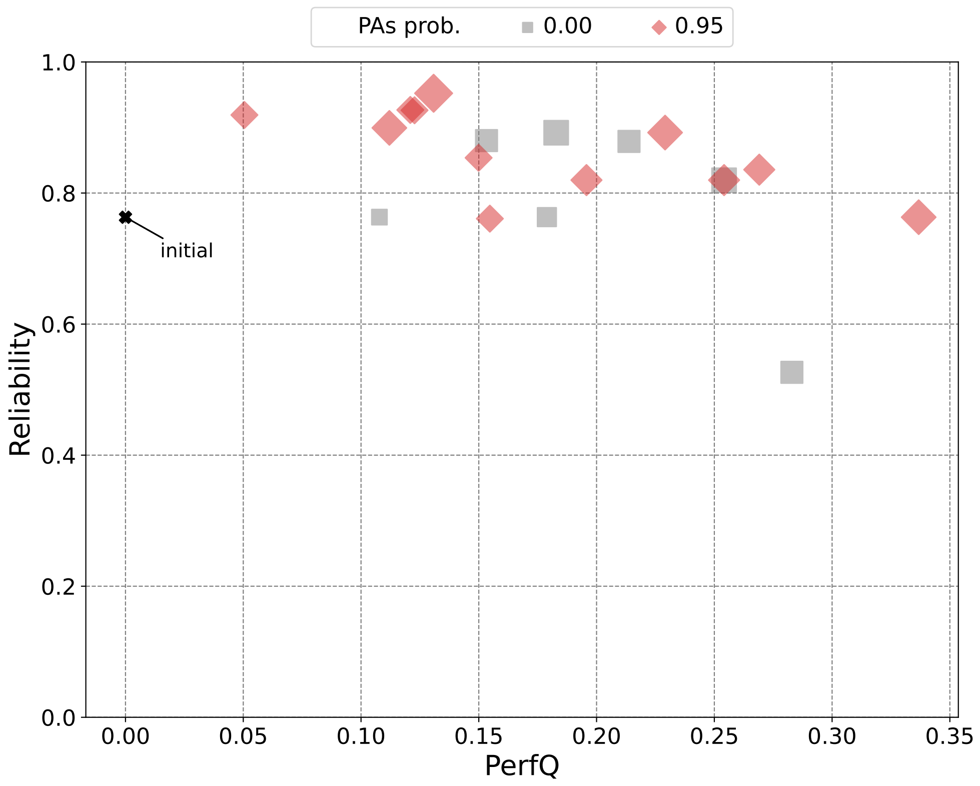


Figure 3.7: The scatter plot of PF^c of `CoCoME` 72 genetic evolutions while considering, and excluding performance antipatterns in the optimization process (*i.e.*, $probpas = 0.95$, and $probpas = 0$).

Discussion Based on our analysis, the reduction of performance antipatterns helps the optimization problem to generate alternatives showing better performance and reliability in most of the cases. CoCoME has mainly shown a light search for better reliability, likely due to the high reliability value of the initial model.

On the basis of our experimentation, we can state that the consideration of performance antipattern occurrences in the optimization process leads to better solutions than the ones found when ignoring them.

RQ1.2

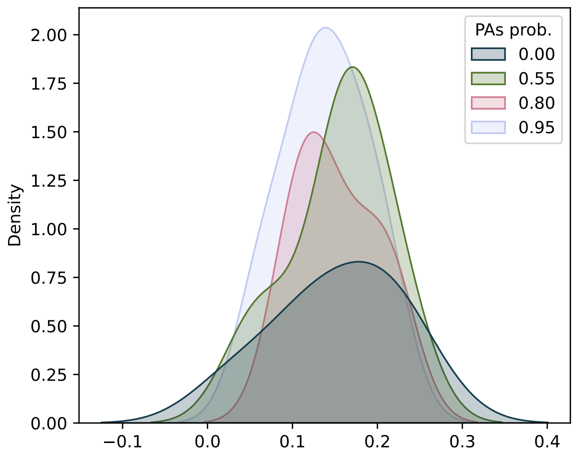
RQ1.2: How the probabilistic nature of fuzzy antipatterns detection help to include higher quality solutions in Pareto frontiers with respect to the deterministic one?

In order to answer this research question, we varied the values of the fuzziness threshold of the performance antipatterns detection within {0.50, 0.80, 0.95} for the two illustrative examples. Figure 3.8 and Figure 3.9 depict the kernel density estimate (KDE) plots showing each possible combination among objectives for TTBS and CoCoME respectively. Each plot depicts the KDE of the relative objectives, *e.g.,* Figure 3.8a shows the *perfQ* KDE for TTBS.

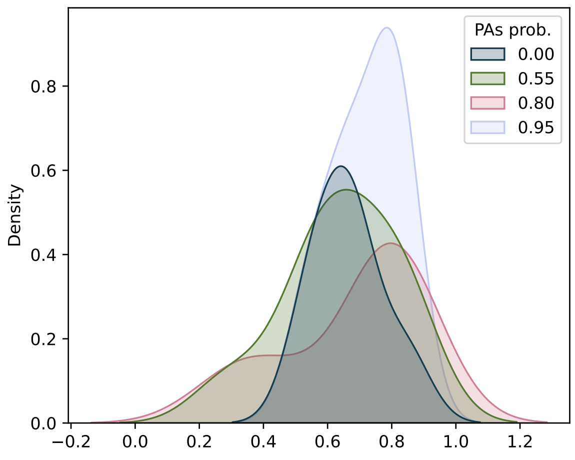
Train Ticket Booking Service For TTBS, we have noticed larger variability of *perfQ* when performance antipatterns are ignored, see the flattest curve in Figure 3.8a. In addition, *perfQ* is narrower to the mean (≈ 0.2) when performance antipatterns are involved in the fitness function, which means less variability in terms of performance in the model alternatives. With regard to the *reliability* (Figure 3.8b), it seems to be more stable without performance antipattern detection. Moreover, the performance antipattern detection helps including solutions with higher reliability values than the case without them. Figure 3.8c shows that the lower the fuzziness the more stable the *#changes* values, which means less variability in the model alternatives discovered by the search. Finally, the 0.95 fuzziness reduces the variability of the *performance antipatterns* objective (Figure 3.8d). Thus, the more deterministic, the higher the probability of discovering true positive performance antipatterns.

CoCoME We notice that Pareto frontiers obtained while ignoring performance antipatterns in the fitness function showed larger variability than the ones obtained while considering them. This is depicted in Figure 3.9a where *perfQ* shows negative values and the curve is flatter than the other cases. For CoCoME we notice that the higher the performance antipattern *probpas*, the higher *perfQ*, which becomes similar to a normal distribution with mean falling on 0.3 for a $probpas = 0.95$ of performance fuzziness. In the case of the lowest fuzziness value, *perfQ* assumed the highest value in our experiments. With regard to *#changes* (Figure 3.9c), it increases when *performance antipatterns* are ignored. Moreover, the higher the *probpas*, the more stable *#changes*, which means less variability in the model alternatives. Again, due to the high value of *reliability* for the initial model, CoCoME shows most of the *reliability* values around 0.9 (Figure 3.9b).

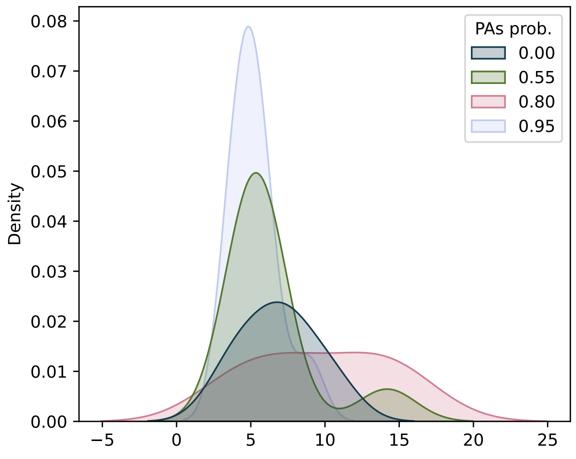
Discussion Our analysis shows that in most of the cases the higher *probpas*, the closer to the mean is the distribution of *perfQ*, which means less variability for *perfQ*. Therefore, it seems better to use



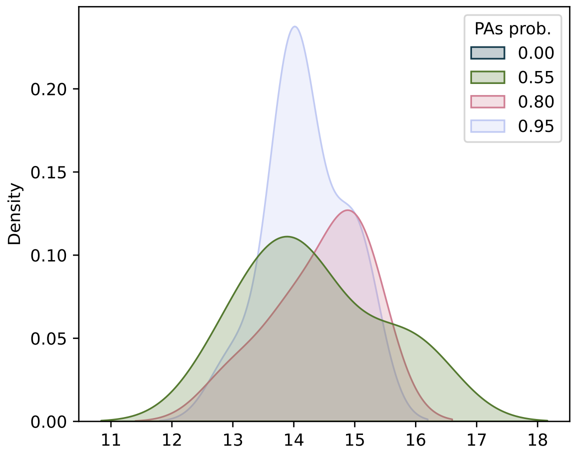
(a) *perfQ*



(b) *reliability*

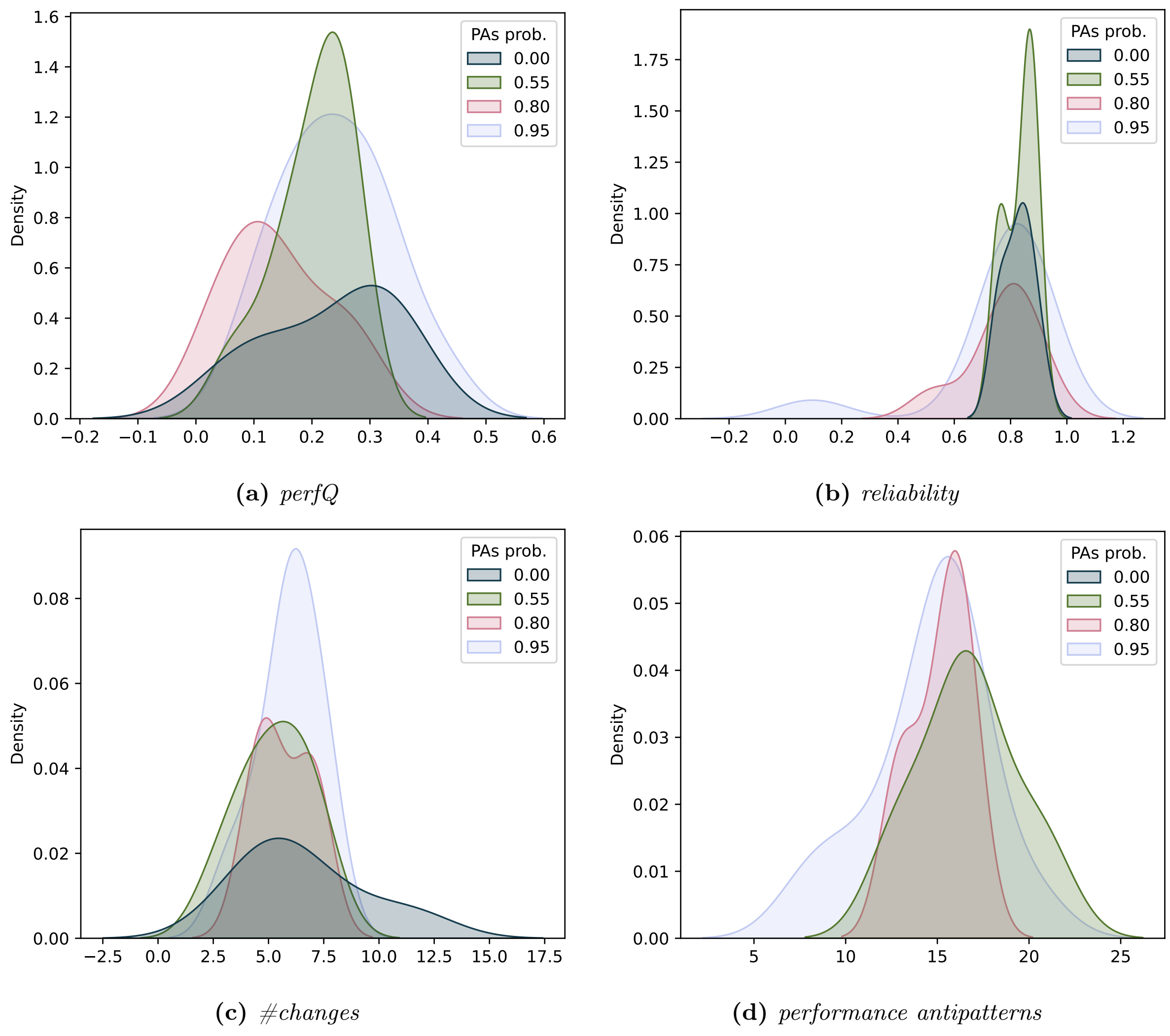


(c) *#changes*



(d) *performance antipatterns*

Figure 3.8: The KDE plots of Train Ticket Booking Service while varying the Performance Antipattern fuzziness probabilities. The *probpas* = 0.00 means performance antipatterns were ignored as objectives. Each plot is referring to the objective in the label.



(a) *perfQ*

(b) *reliability*

(c) *#changes*

(d) *performance antipatterns*

Figure 3.9: The KDE plots of `CoCoME` while varying the Performance Antipattern fuzziness probabilities. The *probpas* = 0.00 means performance antipatterns were ignored as objectives. Each plot is referring to the objective in the label.

a more deterministic antipattern detection (*i.e.,* higher values of *probpas*). However, a deterministic detection has the drawback of relying on fixed thresholds that must be computed in advance for each model alternative. The trade-off between better quality solution and the effort to bind thresholds is likely domain-dependent and worth to be more investigated.

On the basis of our experimentation, we can state that performance antipattern fuzzy detection does not help to improve the quality of Pareto frontiers.

RQ1.3

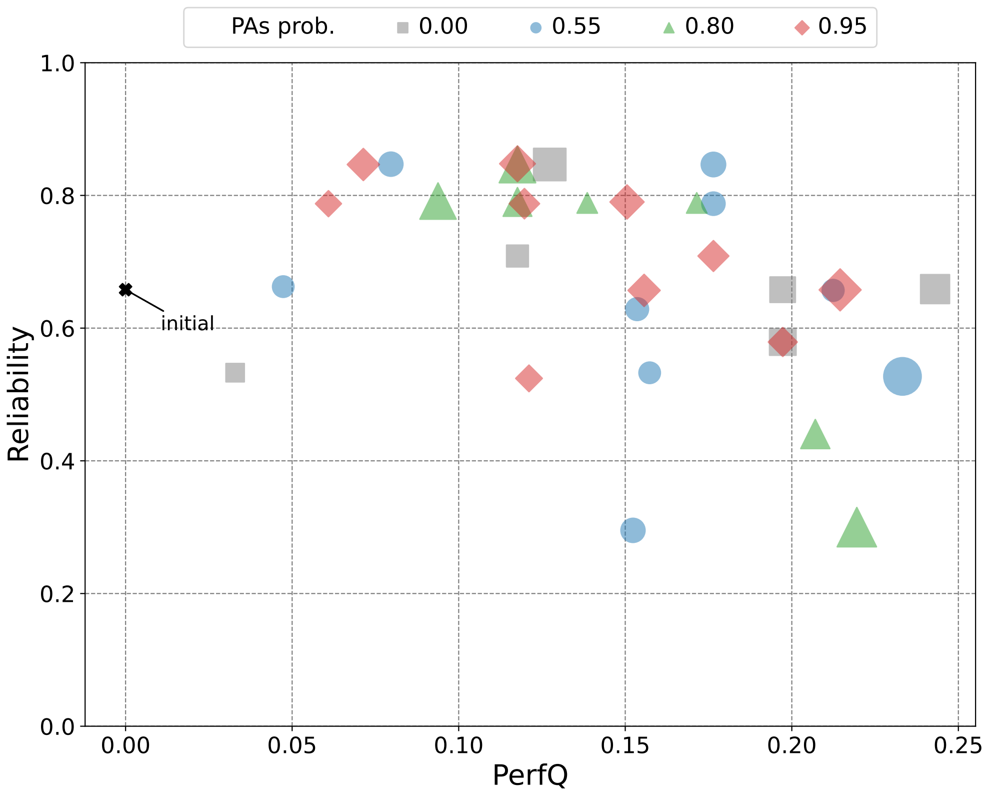
RQ1.3: To what extent does the architectural distance contribute to find better alternatives?

In order to answer this research question, we run the same problem configurations by varying the baseline refactoring factor value. In particular, we decided to activate (*BRF*) and deactivate (*noBRF*) the baseline refactoring factor to study how it contributes to the generation of Pareto frontiers.

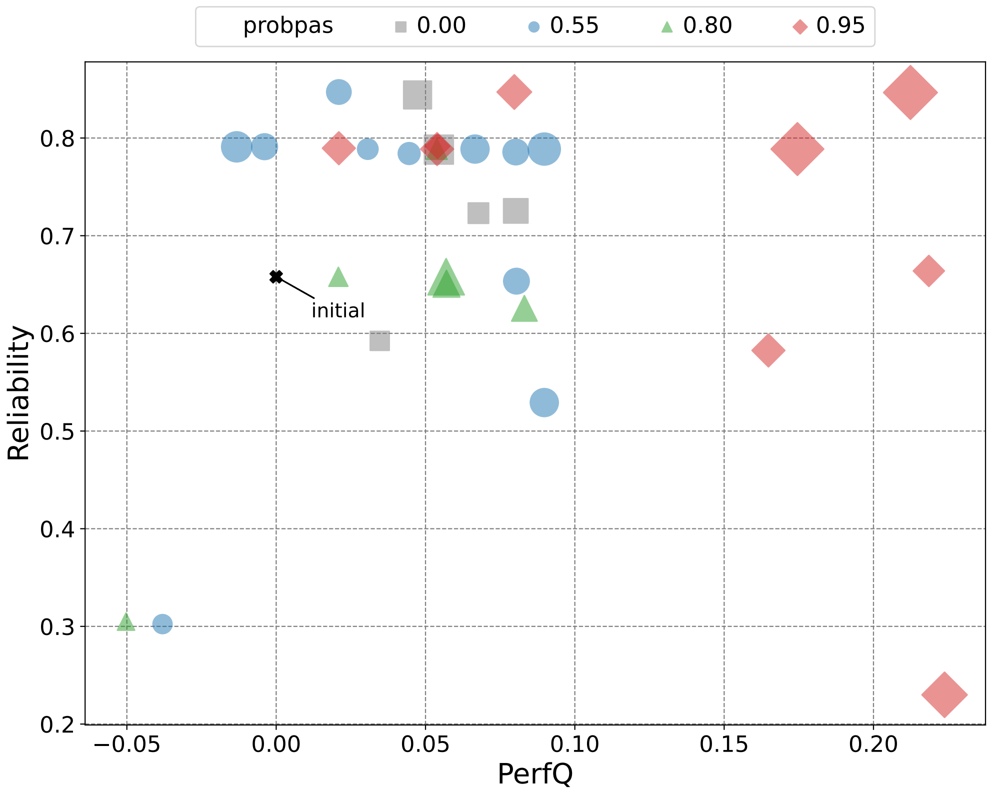
Train Ticket Booking Service Figure 3.10a and Figure 3.10b show Pareto frontiers obtained with *BRF* and *noBRF* configurations, respectively. We can see that results with *noBRF* are narrower to the initial solution (*i.e.,* the black marker in figure) than the case where *BRF* is activated. *noBRF* seems to penalize performance antipatterns with higher fuzziness, in fact $probpas = 0.95$ generates the best alternatives in terms of *perfQ* and *reliability* (see the topmost right corner in Figure 3.10b). However, the highest *perfQ* in the case of *noBRF* is lower than the one in the case of *BRF*. Hence, *BRF* helps the search finding better solutions in terms of *perfQ* for the `TTBS` case. Also, the *noBRF* configuration shows, in a few cases, a detriment of the initial performance and reliability (see the left bottom-most corner) that it never happened when the *BRF* is active.

CoCoME Figure 3.11b shows Pareto frontiers obtained with *noBRF* configuration. By comparing this plot with the one shown in Figure 3.11a, we can see that the *BRF* exclusion generates more densely populated frontiers than the other case. Furthermore, no extreme differences arise between the executions with *BRF* and *noBRF* configurations. In both cases *perfQ* and *reliability* fall within the same region of the plot, where alternatives with *BRF* reached better *perfQ* (see $perfQ > 0.4$ in Figure 3.11a). With regard to the *reliability*, we can see that *noBRF* configuration found few model alternatives showing lower values.

Discussion Based on our analysis, the *baseline refactoring factor* helps generating better alternatives in terms of objectives. We noticed that the *reliability* is penalized with *noBRF* configurations. Also, the *BRF* deactivation penalized *perfQ* in few cases. A deeper investigation is required on how *BRF* might affect the computed Pareto frontiers quality. For example, we can introduce more complex cost models, *e.g.,* COCOMO [45], to improve its estimation. However, we preferred having a more straightforward cost estimation to avoid burdening the search algorithm with additional computational costs.

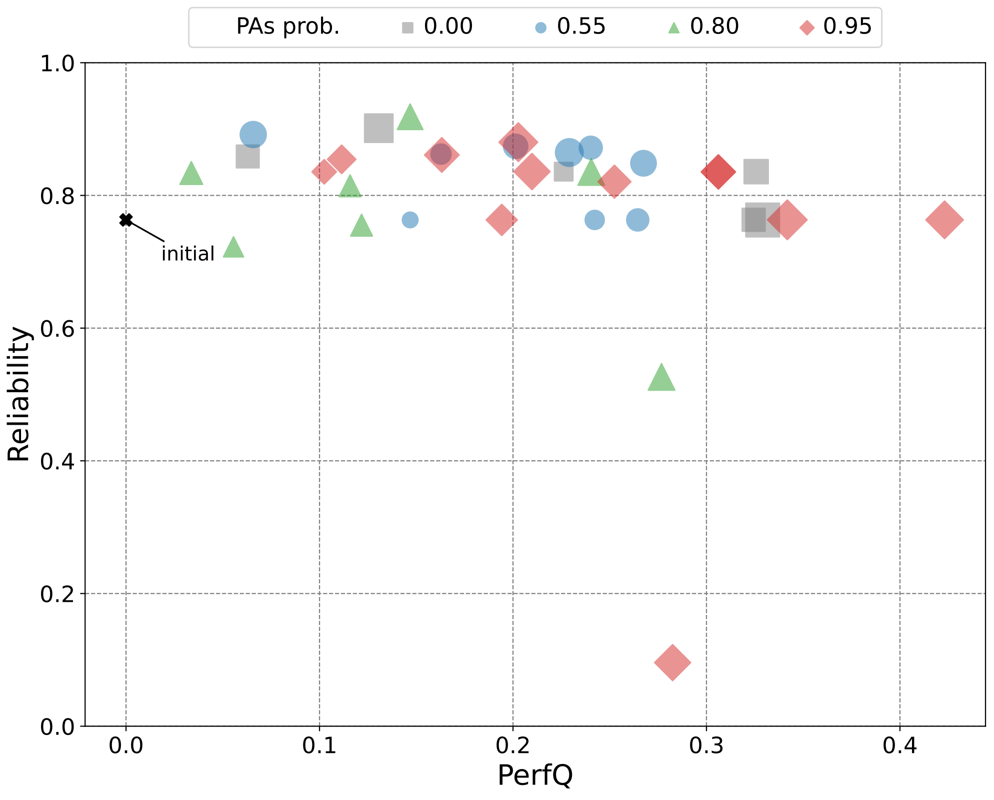


(a) *BRF*.

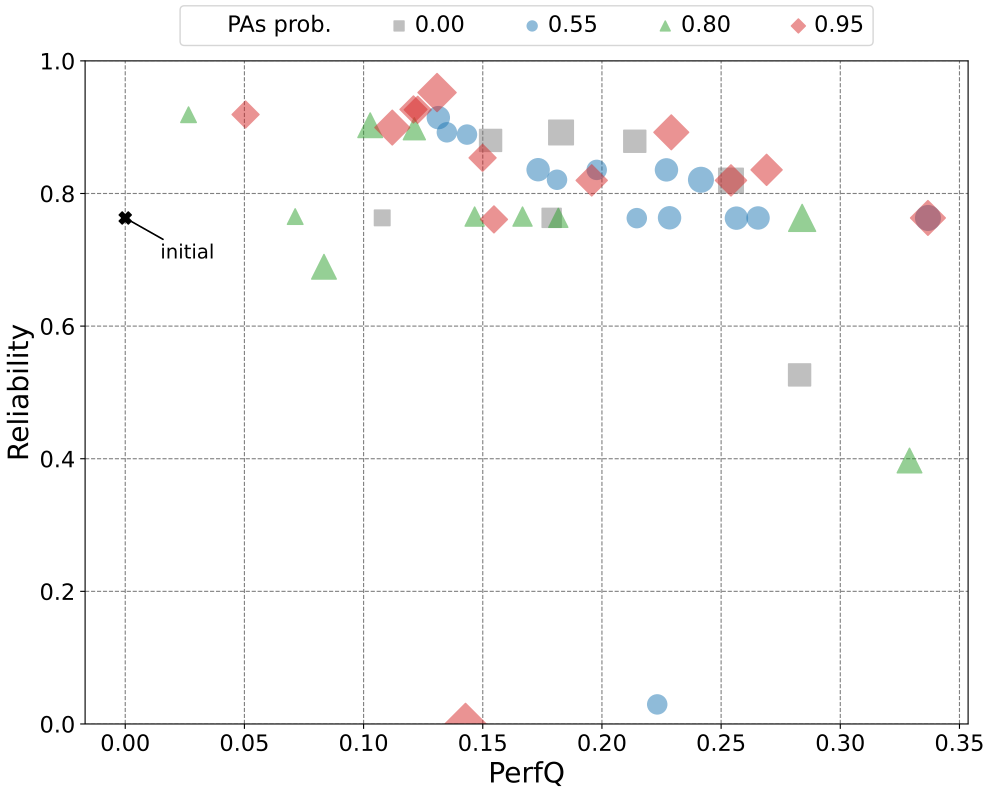


(b) *noBRF*.

Figure 3.10: The scatter plot of Train Ticket Booking Service Pareto frontiers while varying the fuzziness after 72 genetic evolutions with *BRF*, and *noBRF* configurations.



(a) *BRF*



(b) *noBRF*

Figure 3.11: The scatter plot of `CoCoME` Pareto frontiers while varying the fuzziness after 72 genetic evolutions with *BRF* and *noBRF* configurations.

Based on our results, we can state that BRF helps better estimating #changes of refactoring actions, which generates Pareto frontiers showing higher quality (or at least it does not worsen the Pareto frontier quality).

3.5.2 RQ2

RQ2: Is it possible to increase reliability without performance degradation?

We answer RQ2 by looking for model alternatives, within the computed Pareto frontiers (PF^c), that improve both initial reliability and performance.

Figure 3.12 shows the results obtained on the PF^c of `TTBS` and `CoCoME`. The dark dots represent the alternatives we are looking for, *i.e.,* those improving both *reliability* and *perfQ*. Instead, the bright dots represent the model alternatives that improve one of the two non-functional aspects.

Train Ticket Booking Service Figure 3.12a shows that in `TTBS` we obtained 54% of the model alternatives improving *reliability* and *perfQ*. Thus, there is a portion (*i.e.,* 46%) presenting a detriment of the *reliability* but an improvement in terms of performance. This is confirmed by looking at the model alternatives within the PF^{ref}: 18 over 26 alternatives are those taken from the examined Pareto. In this case, model alternatives that guarantee an improvement can be very important for a designer, as we find a performance upgrade of up to 27% and a reliability increase of up to 32%.

`CoCoME` The case of `CoCoME`, in Figure 3.12b, strengthens the observations made for `TTBS`. In this case, the majority (*i.e.,* 74%) of the model alternatives improve both *perfQ* and *reliability* of the initial model. This is confirmed by the number of improving alternatives in the PF^{ref}: 38 out of 48. We got an improvement of the reliability up to 24%, which is smaller than `TTBS` but likely affected by the fact that, in this case, the starting model has higher initial reliability (*i.e.,* 0.75). Instead, the performance improvement is higher, *i.e.,* up to 42%.

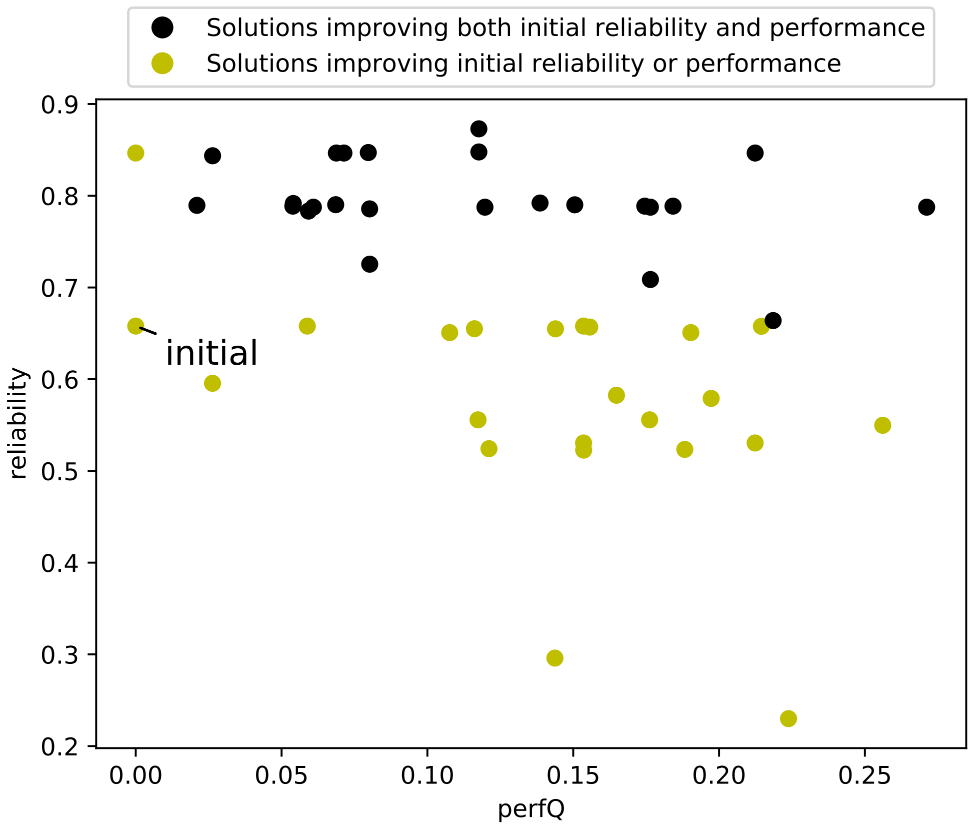
Discussion The set of model alternatives, which have been found while answering to RQ1, are characterized by a neat improvement of two quality attributes: *reliability* and *perfQ*. This result could be fundamental for designers, as they could do further analysis or use the model as a starting point in subsequent stages of the development process.

Our experimentation shows that, our approach can find design alternatives characterized by a significant improvement of both reliability and performance.

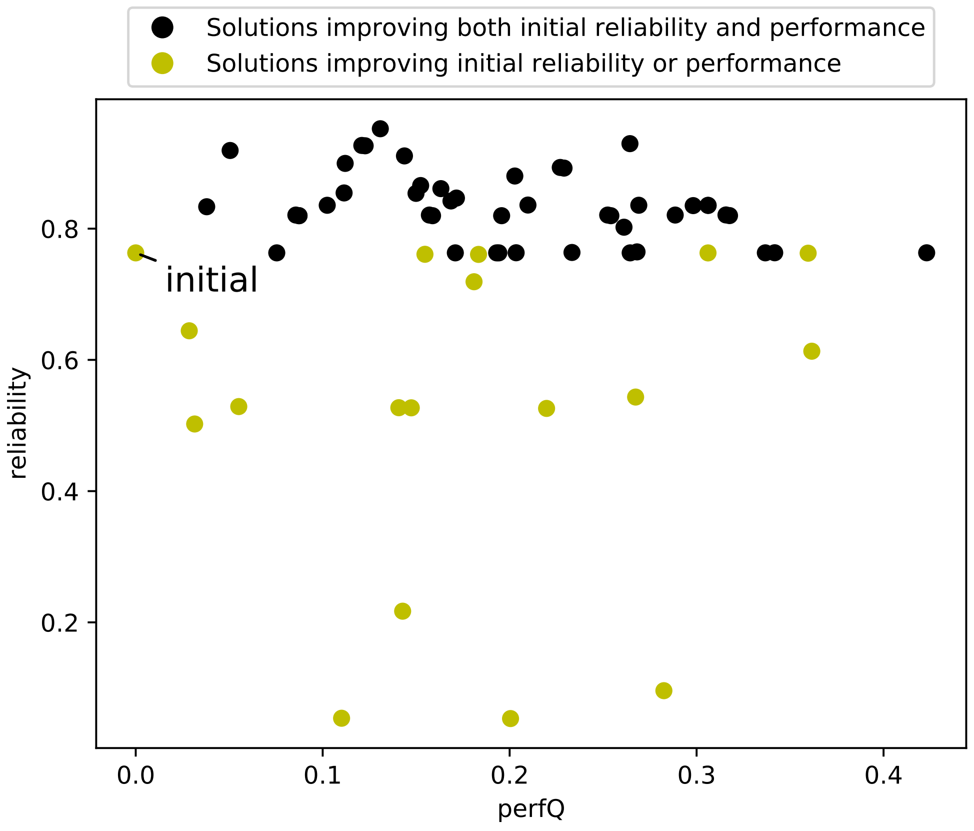
3.5.3 RQ3

RQ3: What type of refactoring actions are more likely to lead to better solutions?

With this research question, we investigate whether some refactoring actions are more likely to be selected than others in the Pareto optimal front during the optimization process. This could potentially lead to more general insights on the effectiveness of specific types of refactoring actions to improve the considered objectives.



(a) Train Ticket Booking Service



(b) `CoCoME`

Figure 3.12: Solutions of the Pareto frontiers displayed according to their reliability and performance.

Train Ticket Booking Service Table 3.7 reports the share of refactoring types for Train Ticket Booking Service. Each row represents a configuration (*i.e.,* an experiment) with a different combination of *BRF*, *maxeval*, and *probpas*. The rightmost four columns represent the refactoring action types that we have considered in our approach. The last row shows the percentages computed over all the configurations.

It is evident that the genetic algorithms prefer to select certain types of refactorings. *MO2C* and *Clon* are clearly more likely to be selected, with a slight preference for *Clon* in most configurations and, consequently, on average across all configurations. These refactorings are inherently very beneficial for the performance: cloning a component will frequently split the utilization in half, and moving an operation to a new component will not only reserve a node for a single operation, but will also relieve the original component of the load related to that operation. Also, they are unlikely to disrupt the reliability objective, since the new nodes will have the same probability of failure as the ones they are cloned from. Conversely, the *ReDe* refactoring may be advantageous for performance and reliability only when the component to be redeployed is sharing the current node with many other components, and this is not the case in the initial model. This is most probably the reason why the *ReDe* refactoring is considerably less likely to be selected, and there is even a configuration in which it was not selected in any Pareto solution (*BRF*: yes, *maxeval*: 72, *probpas*: 0.95).

brf	maxeval	probpas	Clon	MO2N	MO2C	ReDe
no	72	0.00	31.77	42.71	12.50	13.02
no	72	0.55	39.58	47.40	2.60	10.42
no	72	0.80	31.25	53.12	9.90	5.73
no	72	0.95	34.38	28.12	18.23	19.27
no	82	0.00	56.25	27.08	2.60	14.06
no	82	0.55	36.98	39.06	17.71	6.25
no	82	0.80	23.96	51.04	11.98	13.02
no	82	0.95	42.71	30.73	23.44	3.12
no	102	0.00	42.71	30.73	16.15	10.42
no	102	0.55	35.94	27.08	17.71	19.27
no	102	0.80	40.10	30.21	14.58	15.10
no	102	0.95	25.00	58.85	10.94	5.21
yes	72	0.00	40.10	30.21	16.15	13.54
yes	72	0.55	37.50	36.98	14.06	11.46
yes	72	0.80	42.71	19.27	16.15	21.88
yes	72	0.95	49.48	37.50	13.02	0.00
yes	82	0.00	19.79	57.81	10.42	11.98
yes	82	0.55	39.06	36.98	22.40	1.56
yes	82	0.80	27.60	40.62	13.54	18.23
yes	82	0.95	43.75	34.90	20.31	1.04
yes	102	0.00	43.75	35.94	16.67	3.65
yes	102	0.55	41.15	25.00	9.38	24.48
yes	102	0.80	35.42	40.10	10.42	14.06
yes	102	0.95	54.17	22.92	16.67	6.25
Total			38.13	36.85	14.06	10.96

Table 3.7: Share of refactoring types in Train Ticket.

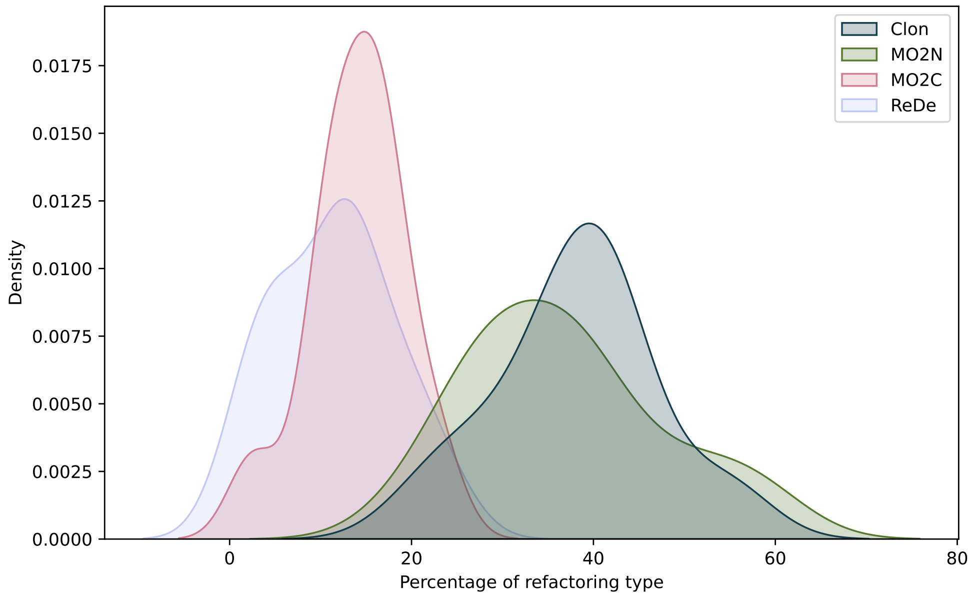
`CoCoME` Analogously, we report the share of refactoring actions for `CoCoME` in Table 3.8. The overall preferences in the selection of refactorings seem to be similar to the Train Ticket Booking Service case. However, we can notice an even stronger preference for the *Clon* refactoring. Since this refactoring largely decreases the utilization of nodes, it may be reasonable to conclude that, in the initial CoCoME model, some nodes with high utilization are preventing the performance to improve. While the *ReDe* refactoring is still the less selected one, there are no configurations in which at least one refactoring of this type does not contribute to Pareto solutions. However, in 13 configurations over a total of 24, the *ReDe* refactoring has a share below 10%.

brf	maxeval	probpas	Clon	MO2N	MO2C	ReDe
no	72	0.00	30.21	37.50	19.79	12.50
no	72	0.55	54.69	24.48	12.50	8.33
no	72	0.80	42.19	32.81	18.75	6.25
no	72	0.95	45.83	37.50	9.90	6.77
no	82	0.00	43.23	25.52	17.19	14.06
no	82	0.55	48.96	27.60	12.50	10.94
no	82	0.80	37.50	41.67	10.42	10.42
no	82	0.95	53.12	28.12	5.73	13.02
no	102	0.00	20.83	36.46	17.71	25.00
no	102	0.55	44.27	28.12	20.31	7.29
no	102	0.80	55.73	27.08	1.04	16.15
no	102	0.95	56.25	28.65	13.54	1.56
yes	72	0.00	41.15	29.17	23.96	5.73
yes	72	0.55	38.02	32.81	20.83	8.33
yes	72	0.80	35.94	42.71	13.02	8.33
yes	72	0.95	61.98	23.44	10.94	3.65
yes	82	0.00	51.56	30.73	14.06	3.65
yes	82	0.55	41.67	33.85	18.75	5.73
yes	82	0.80	38.54	40.62	12.50	8.33
yes	82	0.95	44.27	26.56	16.67	12.50
yes	102	0.00	43.75	20.31	17.19	18.75
yes	102	0.55	66.67	6.25	20.31	6.77
yes	102	0.80	59.90	19.27	8.33	12.50
yes	102	0.95	61.46	16.67	8.33	13.54
Total			46.57	29.08	14.34	10.00

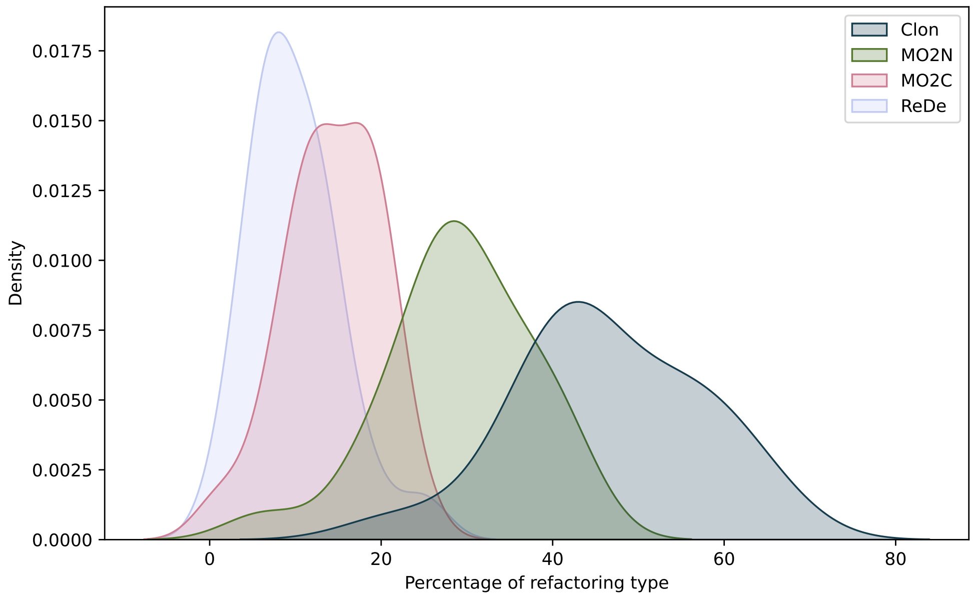
Table 3.8: Share of refactoring types in CoCoME.

Discussion In both illustrative examples, we can observe a common trend on preferring some refactoring types over other ones. In order to confirm that the trend is consistent, we show in Figure 3.13 the density distributions of the shares of refactoring types across the different configurations. The order in which the distributions are shifted along the x-axis is the same in both cases, and their overlapping is somehow similar. This indicates that, on average, the refactoring types are selected with the same order of preference. We can also notice that, while in CoCoME the variability decreases together with the average percentage, in Train Ticket Booking Service the situation is less clear. A greater variability indicates that there are more chances that a change in the configuration will lead to a change in the selection preference of refactoring types, as it can be observed for *Clon* and *MO2N*. On the other hand, a narrow distribution means that configuration changes have little effect on the selection choice, as it happens for *MO2C* and *ReDe*. However, the refactorings that are more likely to be selected (*i.e.,* *Clon* and *MO2N*) exhibit larger variability in both illustrative examples, thus meaning that these refactorings are also the most variable ones from one configuration to another. This may indicate that, even if these two refactorings dominate, on average, the composition of solutions, the Pareto frontiers obtained by different configurations tend to be quite diverse.

Another aspect to consider is the influence of *BRF* on the choice of refactoring actions. While



(a) Train Ticket



(b) `CoCoME`

Figure 3.13: Distributions of refactoring types among different configurations.

BRF clearly has a direct impact on the *#changes* objective, it looks like its presence is not enough to impose a different order of preference among the refactoring types. On the one hand, it could be expected that the *Clon* refactoring will be the most preferred because of its low *BRF* (1.23), but on the other hand the *MO2N* refactoring, that is consistently in the second place, has the highest value of *BRF*.

In an attempt to understand if there is a stronger relation between refactoring types and the objectives, we have also performed a multiple regression analysis. We tried to predict *perfQ*, *reliability*, and *#changes* using the refactoring types as predictors. The coefficients of determination (r^2) we obtained for each objective and for both examples are very low. This means that the refactoring types are not suitable to explain most of the variability we observe in the objectives. Such a result might be the indication that, at least for the two examples we considered, we are not able to derive general refactoring strategies to improve the objectives without going through the optimization process.

From our experimentation, we were able to establish an order of preference among refactoring types that is consistent in both illustrative examples.

3.6 Threats to validity

The validity of our study can be affected by different threats described by the Wohlin *et al.* classification [194]. In the following, we detail each category by discussing the causes and motivations for each threat.

Construct validity The way we have designed our problem and our experimentation might be affected by *Construct validity* threats. In particular, the role played by the architectural distance objective on the combination of refactoring actions might affect the selection of refactoring actions. However, we have studied the influence of our *BRF* in building PF^c in two different illustrative examples, and it has coherently shown the ability to improve the overall quality of the non-dominated solutions in both cases. We will further investigate to what extent *BRF* could improve the overall quality with more accurate cost estimation, such as COCOMO [45], which might have as drawback the increase of the execution time for *BRF* estimation.

Another important aspect that might threaten our experimentation concerns the parameters of the initial UML model. For example, `CoCoME` showed higher initial reliability that might affect the search. However, in our experiments, it seems that `TTBS` and `CoCoME` initial configurations did not threaten the optimization process. We will further investigate how different initial UML model parameters could change the optimization results. We remark that changing a single model parameter means starting the optimization process on a different point of the solution space that might produce completely different results.

Internal validity Our optimization approach might be affected by *internal validity* threats. There are high degrees of freedom on our settings. For example, the variations of genetic configurations, such as the $P_{crossover}$ probability, may produce PF^c with different quality solutions. Also, the problem configuration variations may also change our results. The degrees of freedom in our experimentation generate unfeasible brute force investigation of each suitable combination. For this reason, we limit the variability to subsets of problem configurations, as shown in Table 3.3. We also mitigate this threat by involving two different illustrative examples derived from the literature, thus reducing biases in their construction.

A fruitful investigation will be on the length of the sequence of refactoring actions. At this stage, we fixed the length to four actions. It will be interesting to investigate how the length of the sequence affects results. At a glance, the longer the sequence, the farther the solutions can go from the initial one, and it means that having a long sequence of refactoring actions might be unfeasible because it generates different model alternatives.

External validity Our results might be affected by *external validity* threats, as their generalization might be limited to some of the assumptions behind our approach.

In the first place, a threat might be represented by the use of a single modeling notation. We cannot generalize our results to other modeling notations, which could imply using a different portfolio of refactoring actions. The syntax and semantics of the modeling notation determine the amount and nature of refactoring actions that can be performed. However, we have adopted UML, which is the de facto standard in the modeling domain. In general terms, this threat can be mitigated by porting the whole approach on a different modeling notation, but this is out of this thesis scope.

Another threat might be found in the fact that we have validated our approach on two illustrative examples. While the two illustrative examples were selected from the available literature, they might not represent all the possible challenges that our approach could face in practice. Nonetheless, our results could presumably hold in all the cases in which the modeling assumptions described in Section 3.2 are met. Specifically, the performance antipattern detection and the refactoring actions are designed to rely on information coming from static, dynamic, and deployment views of the system. Without such information, even if in most cases the refactoring actions would still be applicable, they would not be as effective.

Finally, this study is limited to the use of a single algorithm. Therefore, our results are influenced by the ability of *NSGA-II* of exploring the solution space, given the objectives of our approach. While comparing the effectiveness of genetic algorithms in this context is out of the scope of this thesis, we started investigating this issue [63, 79], and we will continue in future work.

Conclusion validity Our results might be affected by *Conclusion validity* threats, since our considerations might change with deeply-tuned parameters for the *NSGA-II*. Also, parameter configurations might threaten our conclusion. We did not perform an extensive tuning phase for the latter due to the long duration of each run, while we used common parameters for the *NSGA-II*, which should mitigate these threats. We can also soften this threat by employing other generic algorithms to generalize our results. Each algorithm will require its tuning phase, which is a clear drawback in execution time.

Another aspect that might affect our results is the estimation of the reference Pareto frontier (PF^{ref}). PF^{ref} is used for extracting the quality indicators as described in Section 3.5. We soften this threat by building the PF^{ref} overall our PF^{c} for each illustrative example. Therefore, the reference Pareto should optimistically contain all non-dominated solutions across all configurations.

Takeaways Model-based multi-objective refactoring optimization presents a variety of challenges that may jeopardize the validity of results.

Genetic algorithms contain a number of configuration options, to start. Every parameter assignment may have an effect on the outcomes quality. Indeed, there is opportunity for research direction here, since it would be impractical to evaluate every parameter combination. There have been studies on determining the (almost) ideal configuration of genetic algorithms in diverse contexts. We employed the standard genetic algorithm setup, such as the crossover probability [32]. However, it would be interesting to see which study applies to our situation as well. We plan to examine how different configurations affect the outcomes quality in future work.

The initial model setup is another factor taken into account. Studies that mix running data (such as traces) and model artifacts already exist to address this problem. There are plenty of shortcomings with these studies. We recently investigated the potential of model-based performance predictions when models are fed with running application data [64]. We discovered that if models take into account the confounding factors affecting application performance, such as network latency, they can anticipate the performance of the running application.

Moreover, the modeling notation affects how expressive the technique is. For instance, the use of a domain specific language to speed up design time could impair models expressiveness. Therefore, we chose to utilize UML, even though its broad general-purpose character is one of its disadvantages. With regard to the modeling and annotation practices in industry, the effort dedicated to these activities can largely vary depending on the field where industries work. As an example, automotive industries have adopted (since many decades) model-driven engineering approaches for designing their embedded software systems. For instance, [24] provide an interesting study on the adoption in industrial contexts of modeling for sake of non-functional analysis.

Finally, regarding the applicability of the approach, it is difficult to establish a category of systems for which our approach would be better suited. Indeed, the only constraint that we require for its applicability is the usage of UML with the DAM [43] and MARTE [97] profiles. Obviously, such approach should be applied in systems where performance and reliability requirements have high priority. For example: distributed systems where reliable connections and timely response are main critical issues; embedded domains (e.g., automotive) where resources with limited hardware capability must guarantee high reliability. Centralized systems represent a further category of systems that may be subject to stringent performance requirement because, for example, a single host machine and its hardware resources must manage a complex software system.

3.7 Related Work

In the last decade, multi-objective optimization studies have been introduced to optimize various quality attributes (e.g., reliability, and energy [116, 129, 130, 132]) with different degrees of freedom in the model modification (*e.g.,* service selection [57, 166]). A systematic literature review on model optimization can be found in [19]. We consider here, as related work, those approaches that directly involve multi-objective evolutionary algorithms, and the ones that exploit LQN as performance modelling notation [21, 111, 115, 197].

We split this section in two subsections, namely *Software Architecture optimization* and *Layered Queueing Network approaches*. The partition is not strict, as it might happen that some studies fall in both conceptual areas. In order to prevent duplication, we chose to describe these studies in only one specific area.

3.7.1 Software Architecture optimization

Menasce *et al.* have presented a framework for architectural design and quality optimization [133], where architectural patterns are used to support the search process (e.g., load balancing, fault tolerance). Two limitations affects the approach: the architecture has to be designed in a tool-related notation and not in a standard modelling language (as we do in this chaper), and it uses equation-based analytical models for performance indices that could be too simple to capture architectural details and resource contention.

Aleti *et al.* [18] have presented an approach for modeling and analyzing AADL architectures [88]. They have also introduced a tool aimed at optimizing different quality attributes while varying the architecture deployment and the component redundancy. Our work relies on UML models and considers more complex refactoring actions, as well as different target attributes for the fitness function. Besides, we investigate the role of performance antipatterns in the context of many-objective model refactoring optimization.

A recent work compares the ability of two different multi-objective optimization approaches to improve non-functional attributes [149], where randomized search rules have been applied to improve the model. The study of Ni *et al.* is based on a specific modelling notation (*i.e.,* Palladio Component Model) and it has implicitly shown that the multi-objective optimization problem at model level is still an open challenge. They applied architectural tactics, which in general do not represent structured refactoring actions, to find optimal solutions. Conversely, we applied refactoring actions that change the structure of the initial model by preserving the original behavior. Another difference is the modelling notation, as we use UML with the goal of experimenting on a standard notation instead of a custom Domain Specific Language (DSL).

Some authors of this chapter have previously studied the sensitivity of multi-objective model refactoring to configuration characteristics [63], where models are defined in Æmilia, which is a performance-oriented ADL. They compared two genetic algorithms in terms of Pareto frontiers quality. In this chapter, we change the modelling notation from Æmilia to UML, and we add the reliability as a new objective. Both approaches provide a refactoring engine, however, in this chapter, the refactoring engine offers more complex refactoring actions since UML is more expressive than Æmilia.

[87] presented an approach aimed at improving architecture quality attributes through genetic algorithms. The multi-objective optimization considers component-based architectures described through a DSL, *i.e.,* AQOSA IR [117]. The architecture evaluations can be obtained by means of several notation, such as Queueing Network and Fault Tree. The genetic algorithm consider variation of designs (*e.g.,* number of hardware nodes) as objectives of the fitness function. The main difference between our approach and the one of Etemaadi et al. [87] is based on the types of the fitness function objectives. Yet, we used UML as the modeling notation instead of a DSL, and the LQN as the performance model.

3.7.2 Layered Queueing Network approaches

Koziolek et al. have presented PerOpteryx [111], *i.e.,* a performance-oriented multi-objective optimization problem. In PerOpteryx the optimization process is guided by tactics referring to component reallocation, faster hardware, and more hardware. The latter ones do not represent structured refactoring actions, as we intend in this chapter. PerOpteryx supports architectures specified in Palladio Component Model [39] and produces, through model transformation, a LQN model for performance analysis.

Rago et al. have presented SQuAT [162], which is an extensible platform aimed at including flexibility in the definition of an architecture optimization problem. SQuAT supports models conforming to Palladio Component Model language, exploits LQN for performance evaluation, and PerOpteryx tactics for architectural changes. A main difference of our approach with PerOpteryx and SQuAt is that we use the UML modelling notation. We moved a step ahead with respect PerOpteryx and SQuAT. Beyond the modeling notation, we introduced more complex refactoring actions, and we use different objectives, *e.g.,* performance antipatterns.

Model-to-model (M2M) transformations from UML to LQN notations have been presented in [21, 22, 115, 197]. For example, [115] presented a tool, namely Tulsa, aimed at enabling performance analysis of data intensive applications. Li et al. [115] augmented UML models with the DICE profile, which allows expressing data intensive application domain specification. Also, they introduced a model-to-model transformation aimed at allowing a performance analysis through Layered Queueing Network. In contrast with these approaches, we present a novel M2M transformation mapping that employs UML Sequence Diagrams as the behavioral view of the logical structure, instead of UML Activity Diagrams. UML Sequence Diagrams have two benefits: they are adopted more frequently than UML Activity Diagrams for design [85], and they explicitly define method calls, while UML Activity Diagrams usually focus on workflows and processes. Therefore, our approach supports a more detailed behavioral representation in terms of time intervals between method calls.

3.8 Conclusions

In this chapter, we have used *NSGA-II* to optimize UML models with respect to performance and reliability properties, as well as the number of detected performance antipatterns and the architectural distance. We focused our study on the impact that performance antipatterns may have on the quality of optimal refactoring solutions. We studied the composition of refactoring actions, and how the architectural distance metric can help the approach to compute Pareto frontiers.

From our experimentation, we gathered interesting insights about the quality of the generated solutions and the role of performance antipatterns as an objective of the algorithm. In this regard, we showed that, by including the detection of performance antipatterns in the optimization process, we are able to obtain better solutions in terms of performance and reliability. Moreover, we also showed that, the more we increase the probability of detecting a performance antipattern using the fuzziness threshold, the better the quality of the refactoring solutions. In addition, we noticed that the *baseline refactoring factor* generally helps discovering better model alternatives. Another important aspect of our study was to ensure that our approach did not worsen the reliability of the initial model. In this respect, our experiments showed that we were in fact able to increase the reliability of model alternatives, with respect to the initial model, in the majority of cases.

As future work, we intend to tackle the threats to validity discussed before. In particular, we intend to investigate the influence of settings (*i.e.,* experiment and algorithm configurations) on the quality of Pareto frontiers. For example, we will investigate the impact of more dense populations in our analysis, in terms of computational time and quality of the computed Pareto frontiers (PF^c). Also, we are interested in the role played by *#changes*, and specifically in studying the effect of estimating the *baseline refactoring factor* through more complex cost model, such as COCOMO-II [45], on the combination of refactoring actions. A fruitful investigation will be on the length of the sequence of refactoring actions, which is currently fixed to four refactoring actions, and we intend to extend the refactoring actions portfolio, for example, by including fault tolerance refactoring actions [67]. We also intend to extend the reliability model to also take into account error propagation [68]. We will involve other genetic algorithms in our process to study the contribution of different optimization techniques within the model refactoring.

We also planned to study how modeling outcomes could be verified and estimated on real-systems. As a first step to address this long-term study, we combined runtime traces (*i.e.,* traces from a running system) and modeling outcomes [64] and we found out that models can help improve performance of HW/SW systems.

Another interesting aspect to investigate could be whether the refactoring actions proposed in the Pareto frontiers make sense form the point of view of the designer and within the established development practices. Therefore, we plan on using visualization techniques to conduct a detailed analysis of the solutions resulting from the optimization process. Visualizing refactoring solutions also opens to a human-in-the-loop process, in which the designer could interactively drive the optimization towards acceptable solutions.

Part II

Platform-Specific Techniques

Chapter 4

Statement-Level Timing Estimation for Embedded System Design Using Machine Learning Techniques

In the last thirty years, there has been an exponential increase in the spread and evolution of information technology. In this regard, it undoubtedly underlined the spiraling of embedded systems [124]. Their diffusion has led practitioners in researching and implementing techniques to improve the quality of embedded systems. Evaluating the attributes of an embedded system is a task with an effort that should not be underestimated, especially during early design phases. Indeed, working on a higher abstraction level (i.e., Electronic System-Level (ESL)) is needed to early estimate HW/SW performance [78]. Furthermore, in the era of Big Data, the use of Machine Learning (ML) methods [123, 174] in this context can be a valid alternative to the classic methods to estimate metrics for embedded system performance [23, 33, 52]. Performance estimation techniques can be valuable during Design Space Exploration (DSE) for helping designers to compare different HW/SW implementations. This part of the thesis focuses on DSE, which, as mentioned in the introduction, includes techniques to compare different HW/SW implementations. In the V&V-based HW/SW Co-Design, these techniques are used to refine the platform-agnostic model and select a platform for the candidate design. In this chapter, we provide an ML model that can be exploited during DSE to produce accurate performance values of a software executed on a given microprocessor. This work answers the following research questions:

1. RQ_1: How to extrapolate a meaningful timing performance metric for embedded platforms without Instruction Set Simulator (ISS) or the target device, which is time consuming?
2. RQ_2: How to reduce estimation time and increase accuracy for embedded timing software performance predictions using Machine Learning techniques?

RQ_1 relies on the considered timing performance metric extraction [119], and the use of ISS instead of the final HW/SW embedded platforms can reduce the time needed for estimation step, depending on the simulator accuracy. RQ_2 considers to replace ISS or selected HW/SW platforms with ML algorithms to reduce design time and increase accuracy w.r.t. the traditional approaches.

Thus, this section presents an approach for timing performance estimation. The aim is to extract information about the main practice used in system-level design flows in order to reduce the time needed for the initial activities, reducing also estimation errors without an extensive and deep analysis of the final hardware/software platforms. A timing performance metric related to high-level C programming language statements will be exploited, as presented in [145]. In addition, a framework that helps to calculate this metric for a given function will be presented. Additionally, this framework can automatically generate large amounts of constrained random inputs and evaluate statistics on the metric itself. The framework exploits ISSs to evaluate several features from the source code, while static code analysis is used to enrich the dataset. The number of clock cycles needed to execute the program, the number of executed C statements with profiling on the host architecture, the Cyclomatic and Healsted number are a subset of the considered features selected for the following ML activity. These big data have been analyzed through several statistical methods, and useful results encourage exploiting this approach inside an ESL design flow.

4.1 Preliminaries

In the context of Embedded System Design [124], some of the main challenges are related to accuracy and simulation/estimation time associated with estimation methods at different abstraction level [189]. The higher the abstraction level, the lower the accuracy and the estimation time, depending on adopted estimation models and simulators [120]. Different abstraction levels can be considered to perform timing estimation, and several approaches have been proposed in the literature. Malik et Al. [126] explored different performance metrics that need to be considered in embedded software performance analysis and examined a wide range of techniques (e.g., path analysis techniques, system utilization analysis techniques). The section in [35] presents two approaches to solve the performance estimation problem: (1) a source-based approach that uses compilation onto a virtual instruction set, and (2) an object-based approach that translates the assembly generated by the target compiler to "assembly-level". Brandolese et Al. [52] introduce a methodology for software execution time estimation. The procedure is supported by mathematical models of C statements and statistical analysis. Alterbernd et Al. [23] introduce an approach to predict the execution time of software through an early, source-level timing analysis. The estimation is done directly from the source code, without compiling and running it, using a set of virtual instructions, and defining an abstract machine able to execute the source code. Finally, [91] introduces an approach to source-level timing estimation through elementary operations. Most of these works aim to predict timing performance behaviors using time-consuming but quite accurate simulators, mathematical prediction models at the system-level, or virtual platform emulators.

In contrast, timing prediction approaches based on ML techniques are the newest methods exploiting big training data-set to make predictions without knowledge of the considered HW platform. Oyamada et Al. [156] presents a methodology for system design and performance analysis. They use an analytic approach based on neural networks for high-level software performance estimation. The analytical estimation results in errors only up to 29.75%, with an estimated time of about 17 seconds. The section in [157] extracts performances of a small set of computers. It uses this information to develop linear regression and neural network models to predict any different considered computer's performance. They can predict the performance of regarded platforms within 3.4% average error rate. Huang et Al. [106] propose the SPORE (Sparse POlynomial REgression) methodology to build prediction models of program performance using feature data collected from

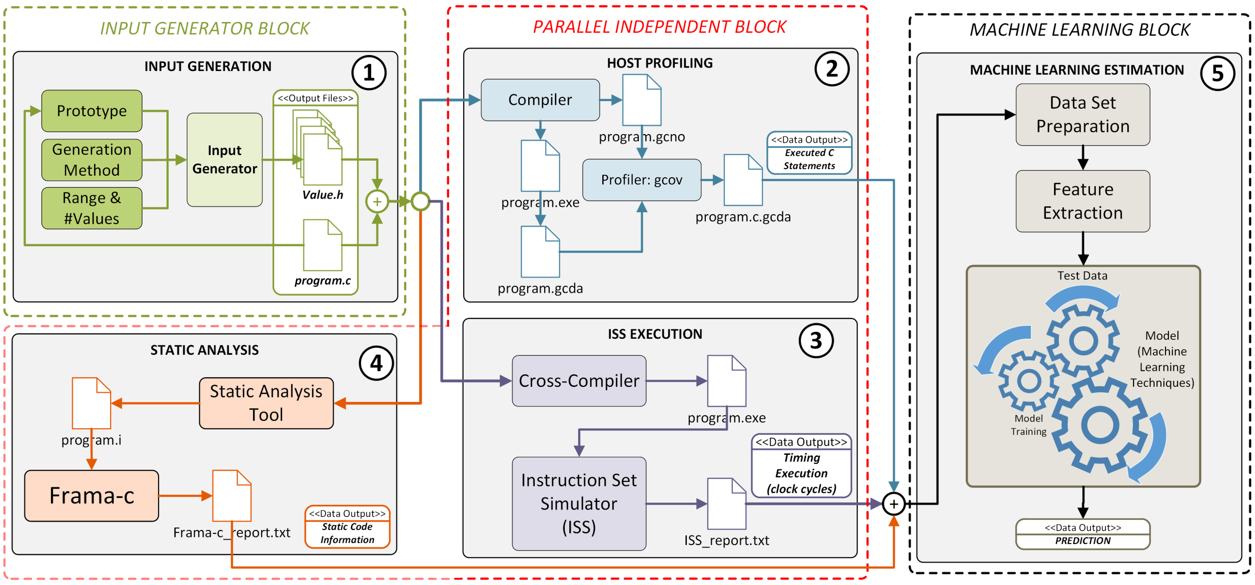


Figure 4.1: Proposed Approach.

program execution on sample inputs, with relative error less than 7%. Finally, the paper in [135] proposes a method through performing data mining on historical data. The authors suggest performing timing prediction using three ML techniques (i.e., Naïve Bayes, Logistic Regression, and Random Forests). These works confirm the validity of ML and data mining in general and the use of these techniques in particular for software estimation. Table 4.1 compares the different approaches, w.r.t. ML techniques and results. Furthermore, to the best of our knowledge, very few research works exploit these techniques for embedded system timing performance estimation, mostly for the low accuracy of these techniques. This section tries to solve this open research problem.

Table 4.1: ML timing performance estimation comparison (N.A. means Not Available).

Work	Domain	Prediction Models	Data Preparation	Benchmark	Platform	Accuracy	Training Time	Estimation Time
[156]	Embedded	Neural Network	N.A.	Custom Benchmark	ARM946	up to 29.75%	Slow	up to 17s
[157]	General Purpose Computer	Linear Regression (LR) Neural Network (NN)	Clementine Software	SPEC2000	AMD Opteron Intel Pentium D and 4 Intel Xeon	LR: 1.5% ... 3.5% NN: 1.16% ... 5.94% (Best Case)	N.A.	N.A.
[106]	General Purpose Computer	Sparse Polynomial Regression	SPORE LASSO Method	Lucene Find Maxima Segmentation	Generic CPUs	up to 7%	N.A.	N.A.
[135]	General Purpose Computer	Naïve Bayes (NB) Logistic Regression (LR) Random Forest (RF)	Wrapper Feature Selection	NASA software Benchmark	Generic CPUs	NB: 17% LR: 19% RF: 14%	N.A.	N.A.
This Work	Embedded	BAT, BOT, FT, LR, SVM	Average Score Value	Custom Benchmark	8051 LEON3 ATmega328/P	up to 2.58% (Best Case)	$<$ 10s	$\ll$ 1ms

4.2 Proposed Approach

The approach proposed in this section performs statement-level execution time estimation using statistical analysis and approximate predictions, as shown in Fig. 4.1. The idea behind this approach is to train the ML model, using (micro-)benchmarks, so that it can predict the performance for any given input C-code function for a given target processor. Compiler and program analysis tools are exploited to extract data for this goal on different real embedded devices. The ML module uses this data to solve the timing performance prediction problem. The whole framework is divided into three macro-blocks: (a) *Input Generator Block*, where the selected functions take several inputs randomly generated inside module ①. (b) *Parallel Independent Block*, where three sub-modules can be executed independently from each other. Module ② uses Gcov program to extract dynamic information about executed C statements; module ③ performs Instruction Set Simulator (ISS) execution, with the collection of clock cycles needed to execute the benchmark function set; module ④ implements static analysis, where Frama-C [13] is used to enlarge the dataset with static function parameters useful for the ML algorithm. (c) *Machine Learning Block*, the core of the estimation method.

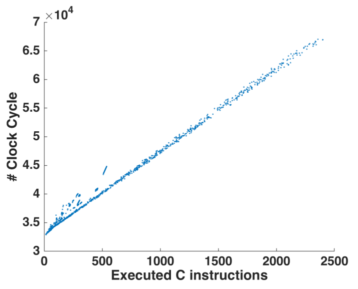
A module that (semi)automatically generates inputs for the benchmark functions has been created in ①. In particular, for each function they are randomly generated 1000 input data sets. Moreover, for each function, different data types have been considered (i.e. *int8, int16, int32,* and *float*) to analyze the results w.r.t. the internal architecture of the considered processor. Each input data set is stored in a header file to be included in the function at compile time. After the inputs generation phase, a procedure to count the number of executed C statements is evaluated in ②. This value is obtained by performing a profiling of the benchmark functions employing the *gcov* profiler for each generated input. To obtain the total number of executed C statements for each function, a sum of the single profiling numbers is performed. The number of clock cycles needed by the target processor technology to execute each function (for each generated input) in the benchmark (and the number of executed assembly instructions, useful for energy/power analysis) is extracted in ③. Depending on the target processor technology there is the need for an Instruction Set Simulator (ISS) or an High Level Synthesis (HLS) for Special Purpose Processors (SPP). The latter is not part of this section. The last data are evaluated using Frama-C tool and other useful script in ④, integrated inside the whole framework instance. Finally, all the generated data output artifacts are sent to ⑤ where the information is organized, selected and exploited for model training, test and validation, as described below.

4.3 Dataset Preparation and Feature Extraction

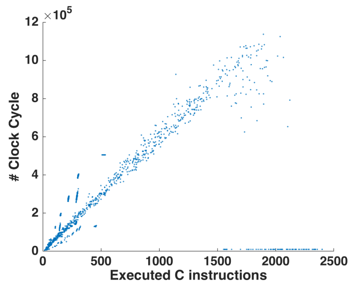
In this study, several features have been chosen to provide a statistical analysis of collected data (i.e., distribution parameters). Several Python scripts have been created to automate all CSV file's statistical analysis, integrated into the last framework module ⑤. To guarantee unbiased data and correct ML training activities, feature analysis is applied to the output framework dataset (i.e., 33 features) that can be clustered w.r.t. the number of Source Lines of Code (SLOC), input data type, Halstead Complexity, Cyclomatic Complexity, functions reachability and program profiling, as described in [11]. Once the dataset is created, four regressor models are considered in order to perform the feature analysis, taking inspiration from [114]. In particular, we considered the *Random Forest Regressor*, *Extra Trees Regressor*, *Gradient Boosting Regressor* and *Ada Boost Regressor*, exploiting the python *scikit-learn* package. The average value of the previous results [114], reduced by the confidence interval value of 99%, has been used as the lower bound to select the most important features. After the feature analysis process, the most significant features will be taken into consideration. ML algorithms will be trained to predict the timing performance metric, or better the *Effective Clock Cycles* feature.

4.4 Machine Learning Techniques

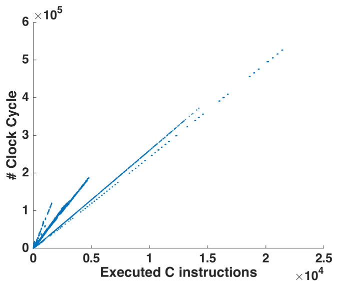
The ML techniques considered, among the most used in literature [56, 102], range from regression trees to SVM, and linear regressors, with the aim to identify which of these can better predict the timing performance metric. At this stage of the work, the Matlab app Regressor Learner [14] has been used. In particular, we considered five ML methodology, according to those available: Linear Regression (LR), Fine Trees (FT), Boosted Trees (BOT), Bagged Trees (BAT), and Support Vector Machine (SVM). We excluded Neural Network Algorithm's usage since we plan to dedicate a specific work in future time so as not to burden the discussion in this study [95]. Furthermore, this analysis involves the usage of a different Matlab package.



(a) LEON3 Corr. Scatter Plot.



(b) 8051 Corr. Scatter Plot.



(c) ATmega328/P Corr. Scatter Plot.

Figure 4.2: Correlation plot w.r.t. different processors and float data type

Linear regression models [103] have predictors linear in the model parameters and are easy to interpret and fast for making predictions. However, the highly constrained form of these models means that they often have low predictive accuracy compared to the others. Regression Learner App uses the *fitlm* function to train *Linear* model option. Regression trees [103] are easy to interpret, fast for fitting and prediction, and low in memory usage. The Matlab App Regression Learner gives the possibility to choose among different kinds of regression trees. In this section, the *Fine Tree* option was selected, which means high flexibility, with many small leaves for a highly flexible response function (the minimum leaf size is 4). The Regression Learner App uses the *fitrtree* function to train regression trees, with the parameter *Minimum leaf size* equal to 4 and no splitting criteria for surrogate nodes. Ensemble Boosted Tree model combine results from many weak learners into one high-quality ensemble model. The approach involves a least-squares boosting methodology with regression tree learners [103]. Regression Learner App uses the *fitrensemble* function to train ensemble models and gives the possibility to set three different parameters: the *Minimum leaf size*, the *Number of learners* and the *Learning rate*, which are respectively 8, 30 and 0.1. Another kind of ensemble model with regression tree learners is the Bagged Trees [103]. Regression Learner App uses the *fitrensemble* function to train ensemble models. Here the parameter to set are two: the *Minimum leaf size* and the *Number of learners*, which are respectively 8 and 30. Support vector machines are supervised learning methods used both for classification and regression [103]. Among the advantages of the support vector machines approach, it is effective in high dimensional spaces and in cases where the number of dimensions is greater than the number of samples. Regression Learner App uses the *fitrsvm* function to train SVM regression models and, in this case, the parameter *Kernel function* is set to *Linear*.

4.5 Experimental Results

The proposed approach has been evaluated first in the SW domain by considering General Purpose Processor (GPP) technology. In this work, three GPPs, and their related ISSs, have been analyzed: ***LEON3***, a 32-bit synthesizable RISC soft-microprocessor with a Harvard architecture, and a simulated system clock of 75 MHz. Cobham Gaisler offers TSIM System Emulator as an accurate ISS of LEON3 processors, with gcc compiler (in this work, the default optimization flag $-O0$ has been used); ***Intel 8051 micro-controller***, an 8-bit micro-controller with MCS-51 CISC instruction set and Harvard Architecture; The Dalton Instruction Set Simulator (ISS) [12] has been used to simulate programs written for the 8051 and to collect statistics, with Small Device C Compiler (SDCC) compiler; ***picoPower CMOS 8-bit AVR ATmega328/P***, with an 8-bit RISC-based processor core and Harvard architecture. The SimulAVR [38] program has been used as a simulator for the Atmel AVR family of microcontrollers, with gcc compiler (in this work, the default optimization flag $-O0$ has been used).

The benchmark execution has been done with a microprocessor software simulation using each processor's ISS, as presented above.

4.6 Dataset Preparation and Feature Extraction Results

A benchmark composed of control-dominated and data-dominated applications was used to train and test the estimator [156] and finally validate the approach. The benchmark is composed of 15 different algorithms taken from [140] for training and test, with a total amount of $6 * 10^4$ samples, and 12 different algorithms taken from [169] for validation, with a total amount of $48 * 10^3$ samples. Table 4.2 describes the functions used in the experiments. For each function, different data types have been considered (*int8, int16, int32,* and *float*). Indeed, timing performance metric changes w.r.t. the dimension of data [120].

Table 4.2: Functions Set.

Sort and Search	binarysearch, bubblesort, insertionsort, mergesort, quicksort, selectionsort
Numerical	matrixmul, gcd, bankeralgorithm
Networking	astar, bellmanford, bfsdfs, djikstra, floydwarshall, kruskal
Data Processing	allpass, bitrev, can, conv, dir Viterbialgorithm, tap, wave, wrap

Fig. 4.2 shows the scatter plot related to the Pearson correlation for different processors. The

three figures show a strict correlation between C statements and clock cycles. Regarding the other points (the ones under the main linear regression line), the deviation depends on function implementations that introduce different behaviors compared to the significant distribution. These points introduce errors inside the timing estimation activity and are related to the internal processor micro-architecture (i.e., 8/16/32 bit architecture, pipeline, number of registers, etc.).

Table 4.3 shows the Pearson correlation and slope values between clock cycles and executed C statements. The high correlation values reported in Table 4.3 between the Clock Cycles and the number of C statements prompts us to explore further the processor characteristics and to exploit ML algorithms for timing performance estimation.

Table 4.3: Correlation and Slope Analysis results (p-value $\ll 0.001$ for every processor, so that we can reject the null hypothesis and the statistical analysis is highly significant).

Function	**int8**	**int16**	**int32**	**float**	**Tot.**
LEON3 Corr.	0.9939	0.9872	0.9277	0.7465	0.9631
MPU8051 Corr.	0.9939	0.9871	0.9276	0.7465	0.9631
ATmega328/P Corr.	0.7465	0.9939	0.9871	0.9277	0.9631
LEON3 Slope	10.8838	9.4241	10.9702	15.1550	88.9492
MPU8051 Slope	88.2752	110.5197	183.8507	389.2319	88.9492
ATmega328/P Slope	24.7353	11.197	18.1595	14.0614	88.9492

In Fig. 4.3 is reported the result of the feature analysis. The figure (green line) shows the arithmetic mean behavior between the four algorithms results [114]. Some features show prominent behavior compared to others. To select the most important, a selection criterion is introduced, as described in Section 3.1. From Fig. 4.3 it is worth noting that two feature dominates the other ones. These two features are related to the dynamic execution of the code. To introduce features connected to static code analysis, we decided to remove the "Executed Assembly Instructions" and "Executed C instructions" features and to apply the selection criteria to the remaining features again. The solid red line represents the average value calculated without the two dominant features. Since some features are close to the solid red line in Fig. 4.3, we included features with a mean score value holds within the confidence interval of 99% (i.e., the dashed lines), eliminating all those features with a value less than the lower confidence interval value.

Respect to each regressor algorithms the variance is 0.0210 for *Random Forest Regressor*, 0.0169 for *Extra Trees Regressor*, 0.0202 for *Gradient Boosting Regressor* and 0.0188 for *Ada Boost Regressor*. The most prominent value is for *Random Forest Regressor*, while the feature *GLOBAL VARIABLES* has the larger variance than others, considering all the regressor.

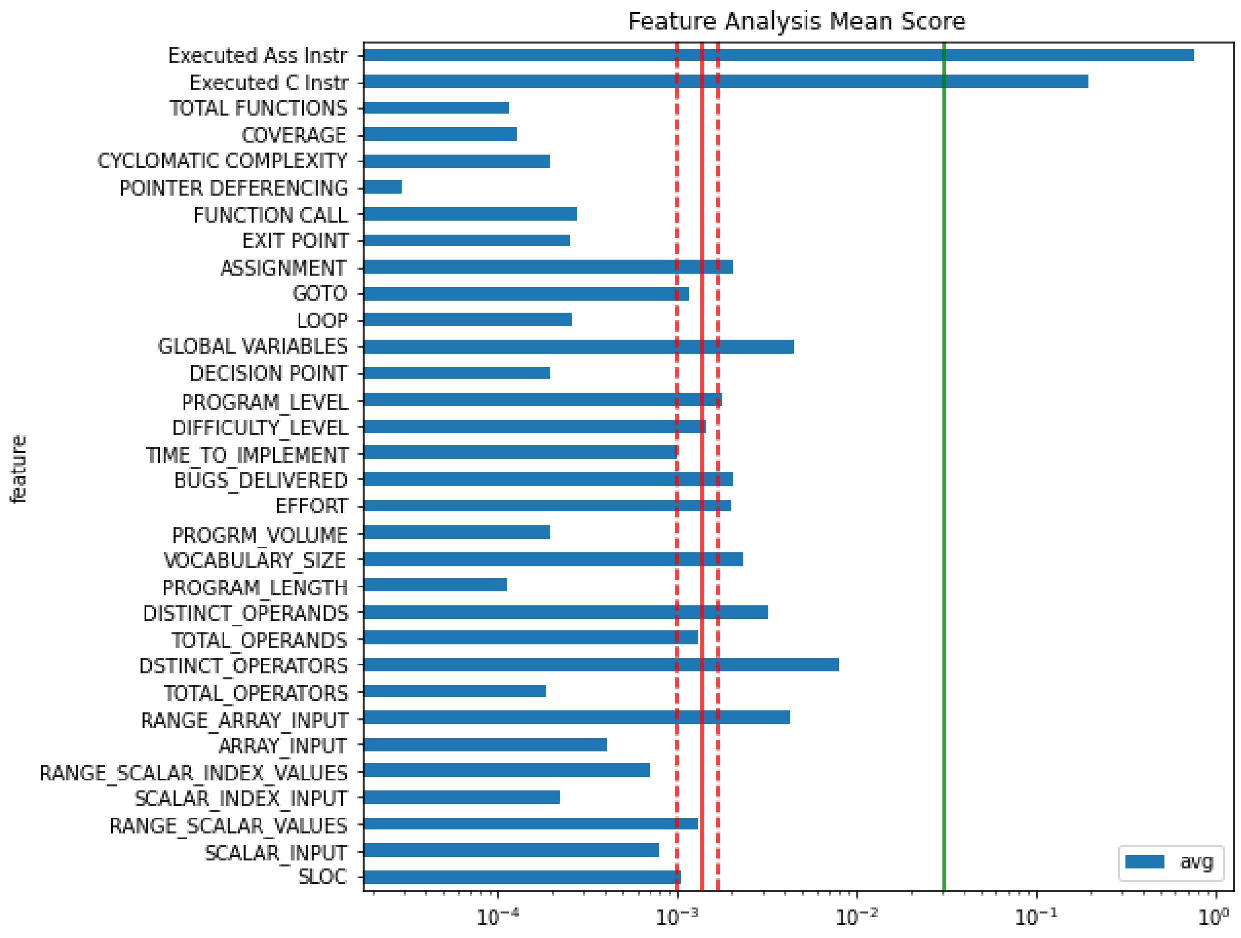
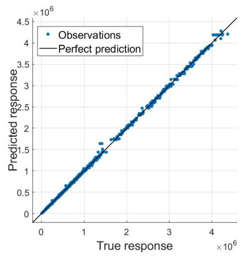


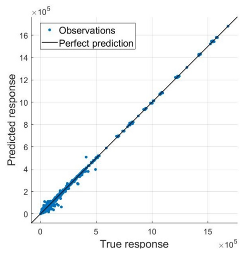
Figure 4.3: Feature analysis (X-axes represent the percentage score from 0 to 1).

4.7 Machine Learning Training and Test Results

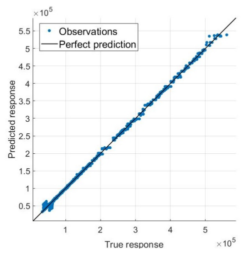
Once the less significant features were eliminated from the dataset for each processor, the ML algorithms listed in the previous paragraph were trained to define a model capable of predicting the output performance variable, i.e., the Clock Cycles. The dataset generated from the 15 functions inside the training and test benchmark is divided into 80% for training and 20% for testing for each processor. After the training process, in Fig. 4.4 are reported the prediction values Vs. the real values for a subset of devices and ML models trained. The good agreement between prediction and real values for *FT* in each device is evident. Also, the *LR* and *SVM* behave well, at least for 8051 and LEON3. The *BAT*, on the other hand, provides bad results for each device. The ATmega328/P device badly behaves for every predictor except for the FT.



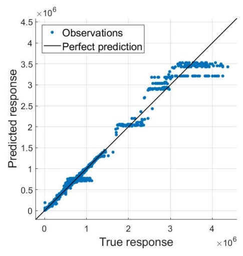
(a) 8051 FT.



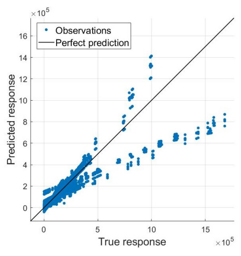
(b) ATmega328/P FT.



(c) LEON3 FT.

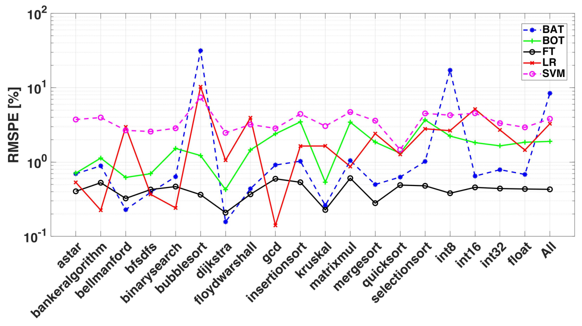


(d) 8051 BAT.

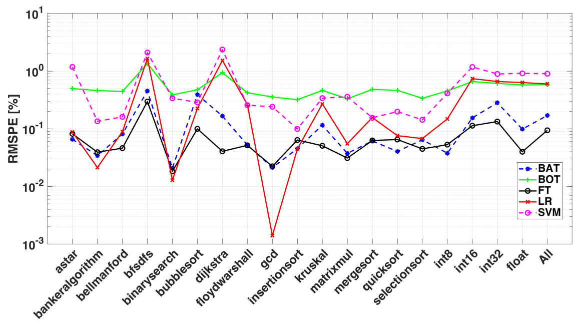


(e) ATmega328/P SVM.

Figure 4.4: Predictions Vs Real Values w.r.t. different prediction models and processors

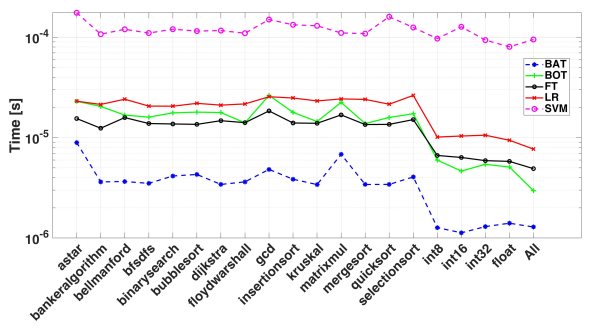


(a) 8051 RMSPE (%).

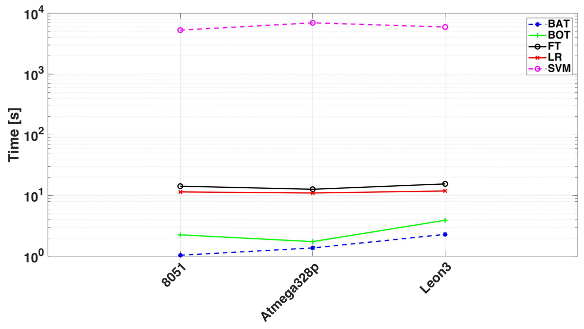


(b) LEON3 RMSPE (%).

Figure 4.5: Test Errors w.r.t. different processors.



(a) 8051 Prediction Time.



(b) Training Time.

Figure 4.6: Training and Prediction Time w.r.t. different processor technologies.

Testing is performed on 20% of the dataset. Fig. 4.5 shows the results of the Root Mean Square Percentage Error (RMSPE) for a subset of the processors. In each plot, it is possible to distinguish each ML predictor model's behavior for various types of data aggregation: by function, data type, total dataset test (All). As mentioned for the prediction vs. real values plots, the *Fine Tree* is the ML model that performs best since its RMSPE values are globally lower than all other ML models. It is worth noting that in the LEON3 and 8051 scenarios, the predictor's RMSPE is always under the 10%, while the FT and BAT for the same devices are generally under the 1%. There is just an exception w.r.t. 8051, where the int8 scenario is badly driven by bubblesort function (maybe related to some dataset noise). The ATmega328/P results are the worst scenario because the error is always $\geq 10^0$ and $\leq 10^1$ for BAT and FT, while is $\geq 10^1$ for BOT, LR, and SVM. Concerning the RMSE, the error variance ranges from 10^2 and 10^4. For the ATmega328/P, the error range is shifted to 10^3 and 10^5.

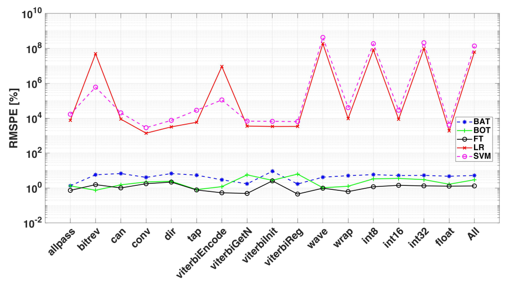
Overall, from the following analysis, it can be stated that the *SVM* does not fit well the problem and owns the worst overall RMSPE, probably since there is a lot of data w.r.t. the feature. *Boosted Trees* and *Bagged Trees* can be considered acceptable. Still, they need an optimization of the model's input parameters, especially for the BAT, if we look at Fig. 4.4. The *Linear Model* is good for prediction but swinging for the RMSE, as expected. The best ML model is the *Fine Tree*, which performs best for predictions and RMSPE. Also, the latter can be improved by further modifying the input parameters of the model.

As a final analysis, in Fig. 4.6 are reported the average prediction time considering 8051 processor (Figure a) and the average training time for each processor of the five ML models (Figure b). The average prediction time behaviour is equal for each selected processor. The best performance belongs to the *Bagged Trees*, while the *SVM* persist with worse execution timing behavior. Knowing each model's execution times is essential to understand how quickly the model is in its training and prediction phase, considering the ESL scenario where it is necessary to execute the model predictions several times. Regarding the training time, BAT and BOT are the quicker ones, under 5 s, LR and FT approximately 10 s, while the SVM takes the slowest execution time, more than 2 h.

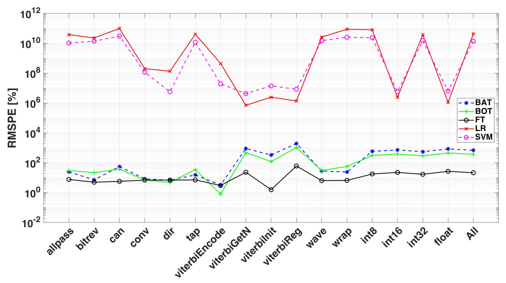
4.8 Validation

In this paragraph, the validation of the five ML models is presented. Fig. 4.7 report the RMSPE for 8051 and ATmega328/P. Still, it is possible to see a good behavior of the decision tree (FT) and ensemble models (BAT and BOT), unlike the *Linear* and *SVM*, which have extremely high percentage values. Regarding FT, BOT, and BAT, it is possible to notice that FT is the best considering the different processors. For the 8051, all the three best models are under the 10% of RMSPE. FT is under the 3% (with values below 1%), BOT is under 7% (with an error range greater than the FT scenario), while BAT is between 1% and 10%. For the LEON3, FT, BOT, and BAT are similar, varying between 9% and 70%. Finally, the unique model that is always under 60% is the FT for the ATmeag328/P, ranging from 1% to 60%, with an average error of about 20%.

The average time prediction analysis shows the same behavior w.r.t. the original training dataset, placing the *BAT* as the best and *SVM* the worst from a timing performance point of view. The prediction times for the validation dataset are similar to those considered in previous paragraphs, while variance is in terms of $ms - \mu s$, as shown in Fig. 4.8.

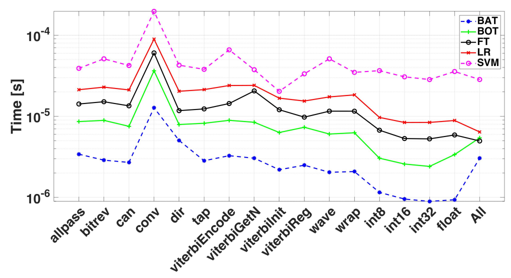


(a) 8051 RMSPE (%).

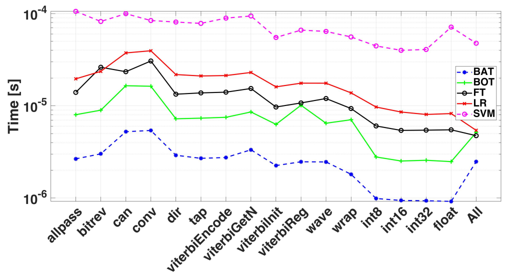


(b) ATmega328/P RMSPE (%).

Figure 4.7: Validation Errors w.r.t. different processors.



(a) 8051 Prediction Time.



(b) LEON3 Prediction Time.

Figure 4.8: Prediction Time w.r.t. different processor technologies and validation data set.

4.9 Discussion

Answer to RQ_1: The considered "meaningful" timing performance metric is the embedded software execution time (i.e., Clock Cycles) [120]. It is possible to use the approach presented in this section to predict this metric. The idea is to use ML techniques instead of simulators/emulators. This idea is reasonable because we have noted a quite high observed correlation among the Clock Cycles, C Statements, and Assembly Instructions Executed, as shown in Fig. 4.2 and Table 4.3. Despite the high correlation values shown in the table, LEON3 and ATmega328/P present lower correlation values, respectively, to float and int8, which can lead to wrong predictions. For 8051, the lowest correlation value of float data type depends on internal microcontroller architecture (i.e., due to the absence of the Floating-Point Unit). The use of the proposed ML approach leads to feature selection problems (i.e., to find the most essential features considering only the software code). From Fig. 4.3, we can state that the most important features that contribute to the prediction model are the Executed Ass. Instr. (Assembly Instructions) and Executed C Instr. (C Instructions). The only Cyclomatic Complexity parameters considered are ASSIGNMENT and GLOBAL VARIABLES, which depends on the input data type. The Halsted Complexity parameters, the Input Data Type, and SLOC are included almost totally, while the Function Reachability parameters (COVERAGE and TOTAL FUNCTIONS) do not reach the considered threshold. The minimum score feature is the POINTER DEFERENCING since there are not many dereferencing pointer operators inside the considered benchmark.

Answer to RQ_2: It is necessary to exploit several ML algorithms and analyze them to reduce estimation time and increase accuracy using ML. Regarding the Fig. 4.4, we can notice that the models based on the regression trees approach behaves better than the other ones. The same behaviour is also present in the plots reported in Fig. 4.5 (training and test) and Fig. 4.7 (validation).

8051 shows the best prediction behavior in terms of RMSPE. For training and test FT, BOT and BAT are under 1%, while validation of the 8051 timing performance prediction errors ranges from 1% to 10%. These low errors depend on 8051 microcontroller architecture (i.e., 8 bit), instructions set (i.e., MCS-51 CISC), and limited register size (i.e., 8 bit). For ATmega328/P, despite the architecture and register size being the same as 8051, the different instructions set (i.e., RISC) leads us to have high errors because of the reduced number of micro-code instructions that deal with a vast amount of assembly line after the compilation step. LEON3 presents good results w.r.t. training and test (i.e., all the prediction models have errors lower than 1%), but worst for validation (i.e., higher than 60%), compared to ATmega328/P (i.e., errors lower than 1% for the FT scenario). This error depends on LEON3's more complex processor architecture (i.e., 32 bit with 7 stage pipeline). Fig. 4.6 and Fig. 4.8 present the training, test, and validation prediction time, where all the results are aligned in terms of time granularity. The fast prediction model is the BOT, followed by BAT and FT, and it shows that these three models are the best in estimation and training time, and they can be improved in future works. This approach seems to be good enough to answer to RQ_2 as presented in [156]. Concerning the results in [156], Table 4.4 present a comparison between the two different approaches (Neural Network for [156], FT for our work). The FT estimation error is lower than the other test scenario approach, while LEON3 and ATmega328/P behave worst in the validation scenario. Also the estimation time is slower than [156], as shown in Fig. 4.6. Future works will continue to refine this strategy working on ML parameters and algorithms.

Table 4.4: Estimation Error Comparison.

Processor	**Max Error**	**Mean Error**	**Std Deviation**
ARM946 Test [156]	29.75%	9.05%	8.90%
8051 Test	0.61%	0.42%	0.12%
ATmega328/P Test	6.03%	2.24%	1.48%
LEON3 Test	0.29%	0.07%	0.06%
ARM946 Validation [156]	N.A.	N.A.	N.A.
8051 Validation	2.58%	1.19%	0.69%
ATmega328/P Validation	65.39%	12.58%	17.79%
LEON3 Validation	73.74%	52.02%	21.36%

4.10 Conclusion

This chapter presented an ML-based approach to predict the timing performance of embedded processors. The modular framework helps the designer extracting static and dynamic code information, used by the next ML module to guess selected embedded processors' timing performance. The results show that the proposed approach is good enough to predict timing behaviors for a CISC microcontroller. Simultaneously, the use of this method for more complex processors introduces estimation errors that can be reduced with more advanced ML techniques. Future works will investigate (1) the possibility to increase the total number of features introducing metrics and features able to characterize not only the input data types but also the input specific values (2) the model prediction parameters optimization (i.e., minimum leaf size and the number of learners for ensembles of trees and regression trees, kernel function for SVM, etc.); (3) overfitting problems (e.g., the LEON3 scenario) and the use of cross-validation to solve them, (4) the possibility to analyze several regression models, different from the ones considered in this section, also considering the use of Neural Networks as a concrete and useful ML algorithm alternative, (5) the usage of the proposed approach to characterize the impact of SW-monitoring systems on given processors [190], (6) the possibility to apply this approach to other domains (i.e., Cyber-Physical Systems, SW Computer performance, etc.) [158].

Chapter 5

An Early-Stage Statement-Level Metric for Energy Characterization of Embedded Processors

Energy consumption is one of the most critical design issues in the embedded systems domain. In particular, the need to guarantee even longer life for all battery-powered devices is one of the main problems that affect the design activities. Indeed, the choices made by designers at the system-level of abstraction, can drastically influence the final system energy consumption, since different optimizations can be considered in the whole *Electronic System Level* (ESL) design flow [98]. Therefore, different energy consumption models can be taken into account in order to estimate the energy consumption of the final system implementation. Such models can be related to processors, Application Specific Integrated Circuit (ASIC), memories, and the interconnections among them. Moreover, the models can be at different levels of abstraction and granularity, mainly depending on the required estimation accuracy.

Since this chapter focuses on embedded processors, Figure 5.1 shows the typical abstraction levels involved in a classical ESL design flow for embedded processors [159]. The first abstraction level, called *Functional*, catches very few non-functional static processor features, as average Clock cycles Per Instruction (CPI), static power dissipation, etc. The *Architectural/ISS* abstraction level involves the knowledge of the Instruction Set Architecture (ISA) and it is normally supported by a so-called *Instruction Set Simulator* (ISS) to perform several kinds of dynamic analysis. The *Pipeline-accurate Architectural/ISS* abstraction level adds details about the behavior of the pipeline into the simulator, providing a more refined processor model. Finally, the *Cycle-accurate Micro-Architectural* abstraction level introduces further details about the processor architecture in terms of *Control Unit* and *Data Path*, allowing a cycle-accurate analysis of the final implementation. The following work is located between the first two levels of the design flow. Here, a *statement-level* metric called Joule for C Statement (J4CS) is proposed, in order to estimate the average energy consumption associated to the execution of a generic C statement by means of a given target processor (i.e., a hybrid functional/instruction level approach). However, two main issues must be firstly clarified. The first one is related to the definition of "generic C statement". This work exploits the same approach presented in [143], where it has been defined, by adopting an empirical approach. In particular, it refers to the way a common profiling tool as *GCov* [3] performs C

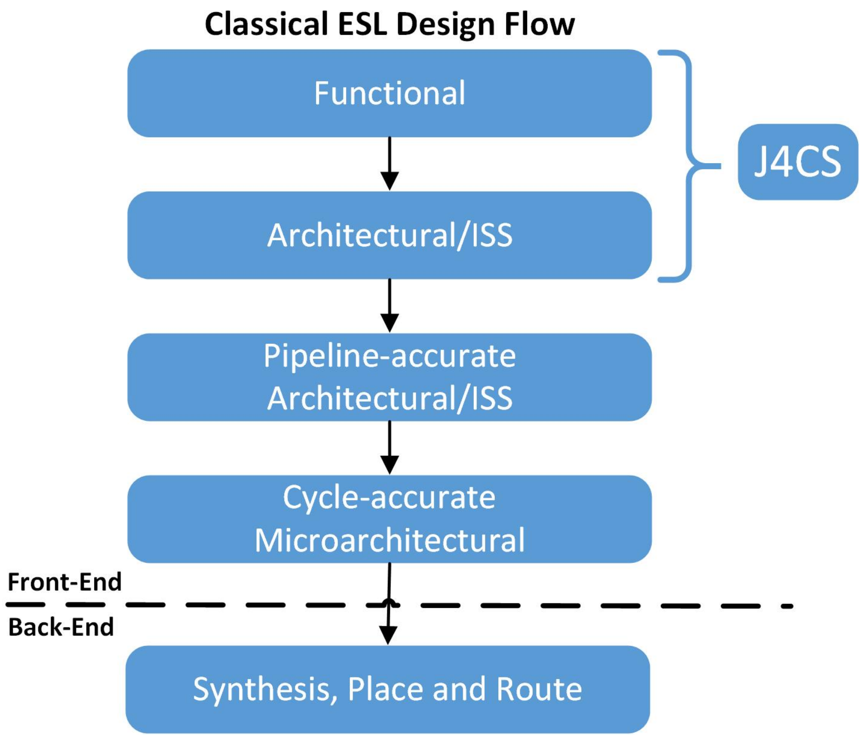

Classical ESL Design Flow
Functional
Architectural/ISS
J4CS
Pipeline-accurate Architectural/ISS
Cycle-accurate Microarchitectural
Front-End
Back-End
Synthesis, Place and Route

Figure 5.1: Classical ESL design flow for embedded processors.

statements identification and counting, when profiling their execution. The second issue is related to the fact that J4CS is influenced also by the used C compiler (and the adopted optimization options) since the optimizations performed by a compiler can lead to different energy consumption value. Furthermore, there are several ways to manage such an influence. One is to introduce the used compiler, possibly giving rise to a J4CS for each processor/compiler pairs. Another one is to consider the average of the results obtained by using the most diffused C compilers. In such a case, the issue can be managed through a statistical characterization of J4CS, considering both different compilers and different compiler optimization options. This can be obtained by evaluating a set of values related to *Min*, *Max*, *Average*, *Standard Deviation* and by also trying to identify an associated statistical distribution. In such a context, this work intends to explore the statistical characterization approach by providing a metric useful to estimate, during Design Space Exploration (DSE), the energy consumption related to the execution of SW on a target embedded processor, so characterizing the processor itself. J4CS is suitable for very fast estimation of energy consumption, as well as comparison and selection of different HW/SW solutions. This chapter enriches the set of platform-specific techniques that can be used during the DSE of our V&V-based HW/SW Co-Design methodology.

J4CS evaluation considers the assembly-level analysis presented in [59] [59], as explained in Section 5.1, and exploits the framework developed in [143]. The obtained value can be assigned to each statement of a given C function and exploited, with a host-based source-level profiling, in order to estimate the total amount of energy consumed when the C function with specific inputs is executed on the target embedded processor. According to this scenario, this work intends to extend the ideas addressed in [138] by providing in addition:

- a comparison of related works, with a focus on platforms and accuracy;

- more detailed aspects about the proposed J4CS evaluation framework;
- the addition of the Intel CISC 8051 micro-controller and the introduction of new target boards;
- a statistical analysis considering different data types and the correlation between assembly instructions and C statements;
- error estimation associated with a validation benchmark, and the exploitation of the *Affinity* [50] metric value to increase J4CS accuracy;

The remainder of the paper is organized as follows: Section 5.1 formally defines the proposed J4CS metric. Section 5.3 presents the J4CS metric evaluation framework applied to two reference embedded processors. Section 5.5 describes how we validated the metric, thus the comparison between estimations and measurements. Finally, Section 5.6 closes the paper with conclusions and future works description.

5.1 Metric Definition

The transistor-level power consumption of a microprocessor [173] during the execution of a given program can be evaluated as:

$$\begin{aligned} P_{tot} &= P_{dyn} + P_{stat} = P_{switching} + P_{short-circuit} + P_{stat} = \\ &= \alpha \cdot C_L \cdot V_{dd}^2 \cdot f + I_{sc} \cdot V_{dd} + V_{dd} \cdot I_{leak} \end{aligned} \tag{5.1}$$

where P_{tot} is the total power consumption made up of dynamic and static power contributions, α is the node transition activity factor (normally, $\alpha = \frac{1}{2}$), C_L is the average switched capacitance per clock cycle during the execution of the program, V_{dd} is the supply voltage, and f is the clock frequency. $P_{short-circuit}$ is related to the direct-path short circuit current I_{sc}, which arises when both the NMOS and PMOS transistors are simultaneously active, conducting current directly from supply to ground. P_{static} is related to the leakage current I_{leak}, i.e., the current that flows through the circuit to ground. The works [167][107] show that the switching activity represents 90% of the total power consumption, so estimations mainly focus on it.

Considering the execution time associated to a given SW program (Δt), it is possible to evaluate the total energy consumption as:

$$E_{tot} = P_{tot} \cdot \Delta t = \alpha \cdot C_{tot} \cdot V_{dd}^2 + I_{sc} \cdot V_{dd} \cdot \Delta t + V_{dd} \cdot I_{leak} \cdot \Delta t \tag{5.2}$$

where C_{tot} is the total switched capacitance. Changing the clock frequency (and so decreasing/increasing the program execution time) doesn't change C_{tot} [173] and so the energy consumption decrease/increase linearly with the scaled frequency with the slope proportional to the amount of leakage. Equation 5.1 and Equation 5.2 define the power processor model at circuit level.

Then, by considering the assembly-level software model presented in [185], it is possible to define the energy consumption associated with a given program also as:

$$\bar{E} = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k (E_k) \tag{5.3}$$

where B_i is the base cost (i.e., the energy cost associated to the execution of a specific assembly instruction), N_i is the number of times a specific assembly instruction has been executed, $O_{i,j}$ represents the circuit state overhead (i.e., the energy overhead due to the execution of two separated sequential instructions), and E_k is the energy contribution related to other inter-instruction effects (i.e. stalls or cache misses) [184].

The higher the abstraction level, the lower the estimation accuracy, but also the lower the timing simulation activities needed for power estimation. Furthermore, at instruction level, considering the average power consumed by a microprocessor while running a program, it is possible to simplify Equation 5.2 and Equation 5.3 by considering $\bar{P} = \bar{I} \times V_{dd}$, where $\bar{I}$ is the average current and V_{dd} the voltage supply. So, the average energy consumed by a program can be expressed by: $\bar{E} = \bar{P} \times N \times \tau$, where N is the number of program clock cycles and τ is the clock period [185].

Thus, while taking into account the formulas described above, the approach proposed in this work exploits some benchmark activities on a specific set of C functions in order to evaluate a metric related to the average energy consumption per C statement, as described below, to estimate a statistical interval of energy consumption.

5.2 Main Assumptions

As described in [59], many embedded microprocessors have a statistical property of constant energy consumption for each executed assembly instruction. So, the proposed idea is to apply the same approach to a higher abstraction level (i.e., statement-level) by characterizing the energy cost (e.g., probabilistic distribution in terms of distribution parameters) associated to the execution of a C statement. This is achieved by performing several simulations in order to consider a meaningful number of execution paths depending on different inputs.

In order to perform statistical analysis, some assumptions must be made:

- if the program has a huge amount of Source Lines of Code (LOC), then the energy consumed for each instruction can be considered constant without great loss of accuracy [173];
- each statement contributes with the same weight for the evaluation of the total energy consumption, since the analysis is given from a statistical point of view, while the evaluation does not consider the influence of operators and variables on the total energy cost;
- an average number of assembly instructions per C statement is considered, since the number of instructions can vary depending on several factors (i.e., memory accesses, addressing modes, register file size, branch mis-predictions, etc.);
- an average number of clock cycles for a pipeline stage divided by the processor efficiency Φ is considered in order to evaluate the mean energy consumed by each executed assembly instruction, as presented in [59].

By these assumptions, it is possible to define the following energy model.

5.2.1 Proposed Energy Model

The approach proposed in this work performs statement-level energy estimation using statistical analysis and approximate predictions.

Definition 5.2.1. $Z = \{z_i \mid i = 1, 2, \cdots, n\}$ is a set of n software functions, $B = \{b_{i,k} \mid i = 1, 2, \cdots, n \ \wedge \ k = 1, 2, \cdots, t\ \}$ is a set of t randomly generated inputs for each function z_i, and $P = \{p_j \mid j = 1, 2, \cdots, m\}$ is a set of m processing units (i.e., processors that are able to execute the considered software functions).

Definition 5.2.2. *Total Energy Consumption for a Generic Software Function*:
Let $CS(z_i, b_{i,k})$ the number of statements executed for a generic software function z_i with input $b_{i,k}$ (evaluated by considering a statement-level execution trace representing the sequence of the executed statements), then the total energy consumed $E_T(p_j, z_i, b_{i,k})$ to execute the whole function z_i on processor p_j with input $b_{i,k}$ is:

$$E_T(p_j, z_i, b_{i,k}) = \sum_{s=1}^{CS(z_i,b_{i,k})} E_s(p_j, z_i, b_{i,k}) \tag{5.4}$$

where $E_s(p_j, z_i, b_{i,k})$ is the amount of energy consumption related to the statement s in the execution trace.

The energy consumption is different each time a generic statement s inside the function z_i is executed, because, depending on involved data location (i.e., memory, cache or register), data type size (i.e., 8, 16, 32, 64 bit), and statement complexity, the number of needed assembly instructions is different.

If $I_s(p_j, z_i, b_{i,k})$ is the number of assembly instruction required to execute statement s of function z_i on processor p_j with input $b_{i,k}$ (evaluated by considering an assembly-level execution trace representing the sequence of the executed instructions), and $E'_{l,s}(p_j, z_i, b_{i,k})$ is the corresponding energy consumed by each assembly instruction l of statement s, then:

$$E_s(p_j, z_i, b_{i,k}) = \sum_{l=1}^{I_s(p_j,z_i,b_{i,k})} E'_{l,s}(p_j, z_i, b_{i,k}) \tag{5.5}$$

$$E_T(p_j, z_i, b_{i,k}) = \sum_{s=1}^{CS(z_i,b_{i,k})} \sum_{l=1}^{I_s(p_j,z_i,b_{i,k})} E'_{l,s}(p_j, z_i, b_{i,k}) \tag{5.6}$$

Simplify Eqn. 5.6, we can consider the mean energy consumption value $\overline{E}'_s(p_j, z_i, b_{i,k})$ for assembly instruction in the execution trace belonging to statement s inside the function z_i executed on processor p_j with input $b_{i,k}$ in this manner:

$$\overline{E}'_s(p_j, z_i, b_{i,k}) = \frac{1}{I_s(p_j, z_i, b_{i,k})} \cdot \sum_{l=1}^{I_s(p_j,z_i,b_{i,k})} E'_{l,s}(p_j, z_i, b_{i,k}) \tag{5.7}$$

$$E_s(p_j, z_i, b_{i,k}) = \sum_{l=1}^{I_s(p_j,z_i,b_{i,k})} E'_{l,s}(p_j, z_i, b_{i,k}) = I_s(p_j, z_i, b_{i,k}) \cdot \overline{E}'_s(p_j, z_i, b_{i,k}) \tag{5.8}$$

We can re-write Eqn. 5.6 in this way:

$$E_T(p_j, z_i, b_{i,k}) = \sum_{s=1}^{CS(z_i,b_{i,k})} I_s(p_j, z_i, b_{i,k}) \cdot \overline{E}'_s(p_j, z_i, b_{i,k}) \tag{5.9}$$

When the executed code of function z_i is longer enough, the mean energy consumption $\overline{E}'_s(p_j, z_i, b_{i,k})$ per assembly instruction on statement s can be considered constant among different statements without great loss of accuracy [173]. Under this assumption, at system-level we have:

$$\left\{ \begin{array}{l} \forall l \in \{1, 2, \cdots, I_s(p_j, z_i, b_{i,k})\} \\ \forall s \in \{1, 2, \cdots, CS(z_i, b_{i,k})\} \end{array} \right\} \Rightarrow \overline{E}'_s(p_j, z_i, b_{i,k}) \cong \overline{E}'(p_j) \tag{5.10}$$

$$E_T(p_j, z_i, b_{i,k}) = \sum_{s=1}^{CS(z_i, b_{i,k})} I_s(p_j, z_i, b_{i,k}) \cdot \overline{E}'(p_j) \tag{5.11}$$

where $\overline{E}'(p_j)$ is the approximate average assembly instruction energy consumption on processor p_j defined as follow [59]:

Definition 5.2.3. *Average Assembly Instruction Energy*:
The average energy consumption associated to a generic assembly instruction executed on a processor p_j can be defined as:

$$\bar{E}'(p_j) = \frac{\overline{MPC}(p_j)}{\phi(p_j) \cdot f(p_j)} \tag{5.12}$$

where $\overline{MPC}(p_j)$ is the *Mean Power Consumption*, $f(p_j)$ is the *Frequency*, and $\phi(p_j)$ is the *Power Efficiency* associated to processor p_j, while $\phi(p_j)$ is related to the MIPS parameter, normally provided on processors data-sheets [150].

Simplify Eqn. 5.11 using the average number of assembly instruction executed, we have:

$$\overline{I}'(p_j, z_i, b_{i,k}) = \frac{1}{CS(z_i, b_{i,k})} \cdot \sum_{i=1}^{CS(z_i, b_{i,k})} I_s(p_j, z_i, b_{i,k}) \tag{5.13}$$

$$I(p_j, z_i, b_{i,k}) = \sum_{s=1}^{CS(z_i, b_{i,k})} I_s(p_j, z_i, b_{i,k}) = \overline{I}'(p_j, z_i, b_{i,k}) \cdot CS(z_i, b_{i,k}) \tag{5.14}$$

where $\overline{I}'(p_j, z_i, b_{i,k})$ is the mean number of assembly instructions executed for each generic statement C belonging on software function z_i on the processor p_j with input $b_{i,k}$, while $I(p_j, z_i, b_{i,k})$ is the total number of assembly instruction executed. Finally, we can define the total energy consumption of a software function z_i running on processor p_j with input $b_{i,k}$ as follow:

$$E_T(p_j, z_i, b_{i,k}) = \overline{E}'(p_j) \cdot I(p_j, z_i, b_{i,k}) \tag{5.15}$$

$$E_T(p_j, z_i, b_{i,k}) = \overline{E}'(p_j) \cdot \overline{I}'(p_j, z_i, b_{i,k}) \cdot CS(z_i, b_{i,k}) \tag{5.16}$$

The evaluation of $I(p_j, z_i, b_{i,k})$ in Eqn. 5.15 is time consuming, since it is needed to evaluate this value each time using an ISS execution, while $CS(z_i, b_{i,k})$ in Eqn. 5.16 considers the use of static profiling tools on host environment and does not need to have the real or emulated target processor. Furthermore, the $\overline{I}'(p_j, z_i, b_{i,k})$ value can be evaluated using a statistical approach, as presented in the next section.

5.2.2 J4CS Metric

Starting from Eqn. 5.16, we need information about the average number of assembly instruction executed $\overline{I}'(p_j, z_i, b_{i,k})$ for each generic statement C belonging to function z_i executed on a processor p_j with input $b_{i,k}$. This feature can be considered as a fixed distribution evaluated on a selected function set (as much as possible characterizing all the possible functionalities used to realize embedded software application) and re-used to evaluate energy consumption associated to a generic C function executed on a processor p_j. For this, it is possible to exploit two profiling activities on a specific selected benchmark:

- by means of an ISS to find the number of executed assembly instructions $I(p_j, z_i, b_{i,k})$;
- by means of *GCov* [143] on a host-based compilation it is possible to find the number of executed C statements $CS(z_i, b_{i,k})$;

Then, it is possible to define a statement-level energy consumption metric as follows below.

Definition 5.2.4. *J4CS (Joule for C Statements)*:
Let $Z = \{z_i \mid i = 1, 2, \cdots, n\}$ a benchmark set of n reference leaf C functions (i.e., no other internal nested function calls). The J4CS metric is the ratio between the number of assembly instructions executed by the target processor p_j running the functions z_i, and the number of executed C statements multiplied by the average energy of a assembly instruction execution $\overline{E}'(p_j)$, i.e.:

$$J4CS(p_j) = \left\{ \overline{E}'(p_j) \cdot \overline{I}'(p_j, z_i, b_{i,k}) = \overline{E}'(p_j) \cdot \frac{I(p_j, z_i, b_{i,k})}{CS(z_i, b_{i,k})}, \forall z_i \in Z, b_{i,k} \in B \right\} \tag{5.17}$$

where $I(p_j, z_i, b_{i,k})$ is the number of assembly instructions executed by the target processor p_j to execute the function z_i with inputs $b_{i,k}$, and the $CS(z_i, b_{i,k})$ is the number of executed C statements for the function z_i with inputs $b_{i,k}$, evaluated with static host profiling. The $J4CS(p_j)$ is a frequency distribution of assembly instructions, C statements and average power consumption, and it is represented by meaningful statistical values (e.g., min, arithmetic mean, standard deviation, median, max, quartiles, percentiles).

The reference approach to evaluate J4CS metric is shown in Fig. 5.2. The first step is the function selection, where all the reference functions, one at a time, are extracted from the benchmark. The next step involves the input generation, where a fixed random number of inputs for each considered function have been generated. Two parallel path evaluates the assembly instructions needed to execute the functions and the "host-based" executed C statements (i.e., they depends only on inputs and functions, not on target processing unit). Finally, it is possible to evaluate J4CS using Eqn. 5.17 with an iterative approach.

As said in the introduction, a first clarification is due with respect to the concept of “generic C statement”. It could be generally intended as “something that ends with a semicolon” (other views are possible too, e.g. Table 6.1 in [171]) but, to avoid ambiguity, this work adopts an empirical approach: it refers to the way a common profiling tool as *GCov* [3] performs the C statements identification when profiling their execution. Another clarification is related to the fact that such a metric will be for sure influenced by the used compiler. Some ways to manage this issue could be: (1) to specify also the used compiler (possibly giving rise to a set of J4CS for each processor/compiler pair); (2) to report the average of the results obtained by using the most diffused compilers; (3)

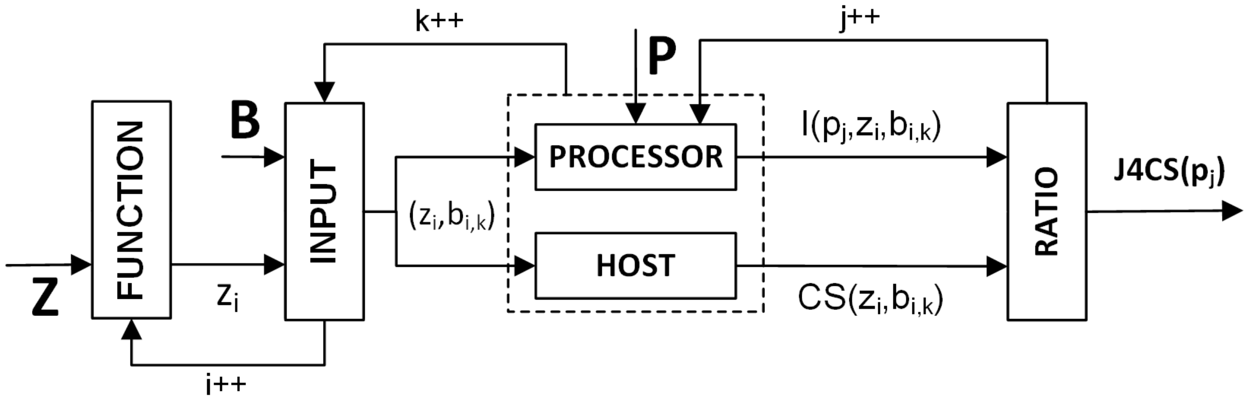


Figure 5.2: J4CS Evaluation Approach.

to report only the results related to the most diffused one. At this point, it is quite clear that J4CS, as defined above, will be influenced by several factors and that a J4CS-based estimation will be probably affected by relevant errors. However, these can be still acceptable by keeping in mind the following aspects: it is a straightforward way to have an off-the-shelf metric (i.e., by defining a standard benchmark it would possible to report its value on a processor data-sheet like normally done for the MIPS metric); it can be applied to several SW processor technologies; it is intended to be used for very early performance analysis in SW domain. Anyway, as described in the next section, J4CS can be also characterized by a set of values related to *Min*, *Max*, *Average*, and *Standard Deviation* (i.e., by statistical distribution parameters). In this way, it is possible to perform different analysis depending on the final goal.

5.3 Evaluation of J4CS

5.3.1 Generic Framework

In order to evaluate the metric for a given target processor, it is needed, at least, to: define a set of relevant C functions to be used as benchmark (ideally a standard benchmark to be used for all the processors) [143]; identify a way to stimulate (i.e., to execute) it by means of relevant input data sets; identify a tool to perform statement-level profiling in order to count the number of executed C statements for each input; identify tools to compile the C function for the target processor and to simulate its execution in order to obtain total number of executed assembly instructions, as shown in Fig. 5.3.

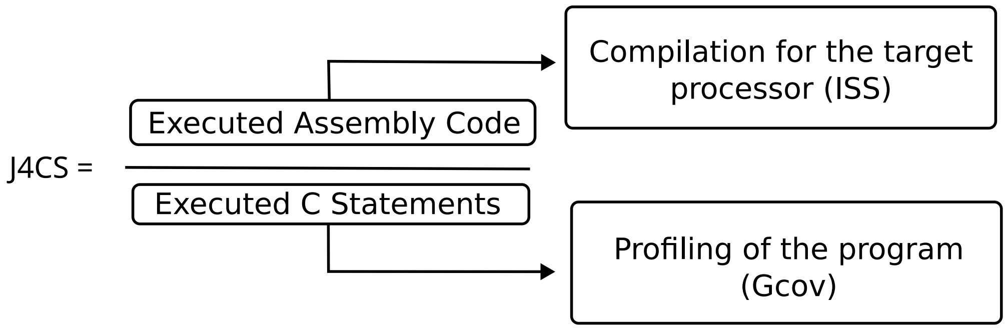

Compilation for the target processor (ISS)
Executed Assembly Code
J4CS =
Executed C Statements
Profiling of the program (Gcov)

Figure 5.3: J4CS Evaluation Methodology.

It is worth noting that only the last step must be applied for each different processors that have to be characterized. Indeed, the others are independent. Moreover, J4CS is an one-time effort since this metric, once evaluated, is available ”for free” for any estimation activity. So, to support J4CS evaluation, a proper framework [144] has been adapted. Additionally, such a framework is also able to evaluate statistics on the metric itself.

The overall flow of the J4CS evaluation can be summarized into three phases.

- Energy Data Acquisition: collect energy information useful to evaluate J4CS metrics (i.e., assembly line executed, C statement executed).
- Energy Model Characterization: calculate the parameter related to energy consumption [59]
- Energy Estimation: execute profiling and substitute this results into the energy model to calculate the total energy consumption.

The advantage of this approach relies on a statistical analysis able to extract as-much-as-possible energy/power pattern behavior, without an exhaustive instruction-level energy estimation for software that often need to perform deep architectural software analysis (i.e., control-flow analysis, loop bound analysis, path analysis).

It is possible, for instance, to consider the real number of executed assembly line, and correlate them with the number of executed C statement. The higher the correlation between these two parameters, the higher the accuracy in power consumption estimation inside a specific fixed interval. The whole framework is shown in Fig. 5.4. The following paragraphs describe the main features of this generic framework, meanwhile processor specific features are described later.

5.3.2 Reference Benchmark

A simple benchmark composed of 14 well-known functions (i.e., C leaf functions z_i in Eqn. 5.17 has been realized. The functions of the benchmark are the following ones:

- *Quicksort*: the sorting algorithm that follows the divide et impera approach. The algorithm recursively divides the input array until many small 1-length arrays was obtained. An array and its length have been passed as parameters.
- *Mergesort*: a sorting algorithm that follows the divide et impera approach. The array is recursively split until the sub-lists are composed by a single element. Then, two adjacent sub-lists are compared and merged sorting the inner elements.
- *Matrix Multiplication*: an algorithm that multiplies rows by columns of two-dimensional array.
- *Kruskal*: it is used to find the minimum spanning tree of a non-oriented graph that does not contains negative edges. It is a greedy algorithm and, in this case, the greedy choice consists in taking always the edge with minimum cost between among those available.
- *Floyd-Warshall*: it calculates the distances between all pairs of vertices of a weighed graph with no negative loops. The costs of the edges may be negative values as long as these are not part of a negative loop.

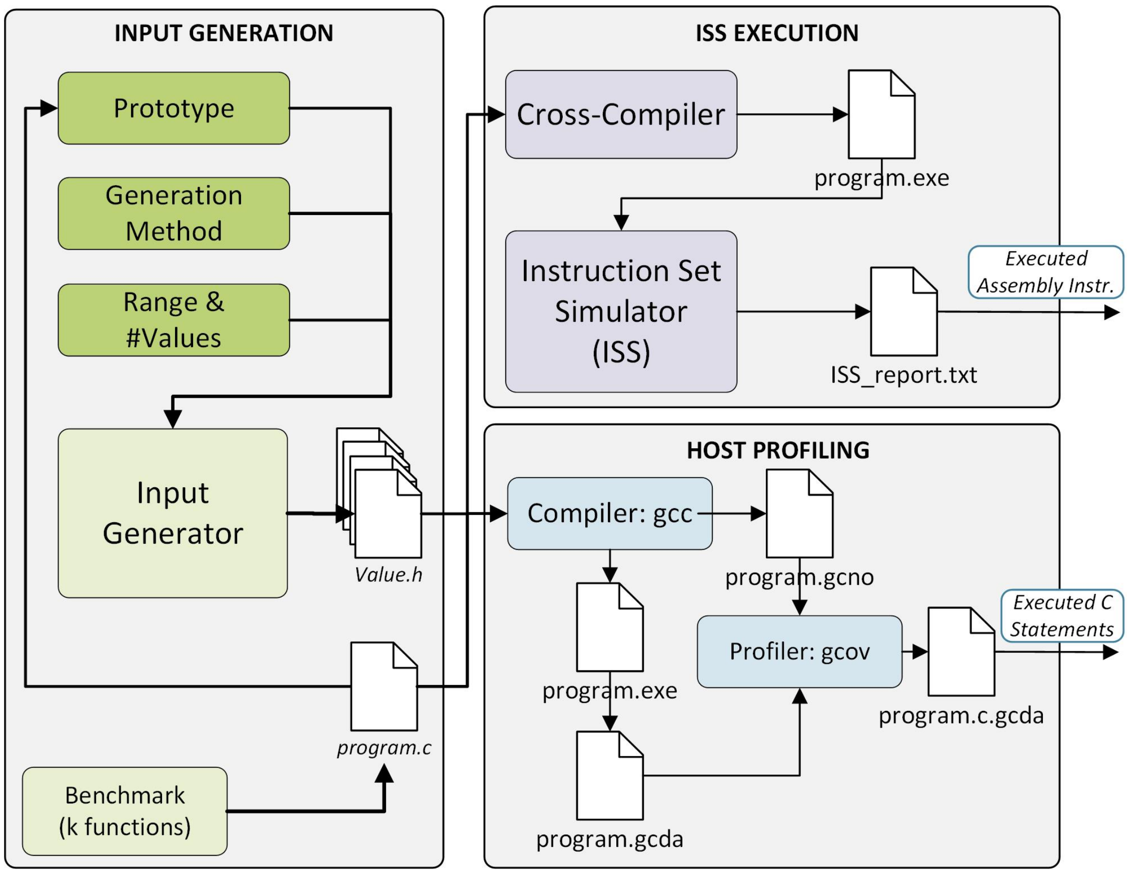


Figure 5.4: J4CS Evaluation Framework.

- *Dijkstra*: it calculates the minimum paths from a starting node x towards all nodes accessible by x. The graph must be oriented, can contain loops and must have edges with positive costs. This algorithm uses the concept of relaxation in order to obtain distances.
- *Breadth First Search and Depth First Search*: two algorithms for traversing a graph. In the first function, the nodes that must be visited are inserted in a queue, while in the second one in a stack.
- *Banker's Algorithm*: it is used in the operating systems to avoid deadlock situations during the allocation of resources to a process.
- *A**: a graph-searching algorithm that identifies a path from an initial node x to a final node y. It is similar to the Dijkstra algorithm that for each node takes into account all possible directions and then chooses the one with lower cost. Instead, A* avoids to visit all edges connected to a node using a heuristic function that estimates the cost to the destination node.
- *Bubble Sort*: Sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.
- *Selection Sort*: it divides the input list into two parts, the subset of items already sorted, and the subset of items remaining to be sorted that occupy the rest of the array.
- *Insertion Sort*: it builds the final sorted array one item at a time.

- *Greatest Common Divisor (GCD)*: the classical greatest common divisor algorithm.
- *Binary Search*: it finds the position of a target value within a sorted array.
- *Bellman Ford*: it computes shortest paths from a single source vertex to all of the other vertices in a weighted graph.

Table 5.1: Benchmark Functions Characterization

Functions	Decision Point	Global Variables	Loop	GOTO	Assignment	Exit Point	Total Operators	Distinct Operators	Total Operands	Distinct Operands
A*	19	13	7	3	39	10	372	41	205	34
Banker's Algorithm	6	12	4	1	20	4	247	33	119	44
Bellman Ford	14	7	7	2	28	5	352	34	168	52
Binary Search	6	6	3	1	19	4	195	35	75	25
Bubble Sort	4	5	2	0	17	4	243	29	105	57
Dijkstra	15	12	6	2	35	6	331	39	161	35
Floyd-Warshall	4	5	3	0	11	3	266	26	140	88
GCD	6	6	2	1	13	4	164	25	55	21
Insertion sort	3	7	2	0	12	3	164	30	59	20
Kruskal	23	14	12	1	46	11	394	34	194	38
Matrix Multiplication	4	6	3	0	16	3	207	33	80	35
Mergesort	10	6	6	0	41	5	271	36	133	45
Quicksort	6	5	4	0	27	5	238	36	118	33
Selection Sort	3	7	2	0	14	4	163	28	62	22

The source-code is available on [2]. Table 5.1 and Table 5.2 summarizes the main features of the whole benchmark. Functions are characterized by decision points (i.e., control flow statements), loops, GOTO (i.e., unconditional jumps), variables assignments, functions exit points, operators and operands (total and distinct). Moreover, all the functions are leaf ones. Functions source-code is characterized by source lines of code (SLOC), data types, inputs types (scalar or vector) and their values. SLOC do not include comment lines or empty lines. The scalar values range have been chosen related to 8051 internal memory size (128 KB), to prevent buffer overflows.

5.3.3 Inputs Generation

A module that (semi)automatically generates inputs for the benchmark functions has been used in order to evaluate J4CS (i.e., $b_{i,k}$ in Eqn. 5.17). In particular, for each function they have been randomly generated 1000 input data sets. Moreover, for each function, different data types have been considered (i.e., *int8, int16, int32,* and *float*) to analyze the results with respect to the internal architecture of the considered processor. Each input data set is then stored in a header file to be included in the function at compile time.

The module needs to know what variables the function requires. For this purpose, the user must define a prototype of the implemented function. This prototype contains the function name and the name and type of each input variable. The input generator parses the prototype file to find its name and to find out proper data for the function. For each variable, the user is asked to insert a values range (as shown in Table 5.2) and then the module will generate the number of values accordingly. With a function that requires more then one variable, it performs the Cartesian product of generated values. For each combination obtained, it creates a header file that contains

Table 5.2: Source-level characteristics

Functions	**SLOC**	**Data Type**	**Scalar Inputs**	**Range Scalar Values**	**Array Inputs**	**Range Array Values**
A*	105	int8	s, g	$s \in [2,9]$, $g \in [0,8]$	a[s][s]	[-128,127]
		int		$s \in [2,6]$, $g \in [0,5]$		[-32768,32767]
		long		$s \in [2,3]$, $g \in [0,2]$		[-2147483648,2147483647]
		float		$s \in [2,3]$, $g \in [0,2]$		[-2147483648.0,2147483647.0]
Banker's Algorithm	46	int8	nr, np	$nr \in [1,5]$, $np \in [1,5]$	available[nr]	[0,255]
		int		$nr \in [1,3]$, $np \in [1,3]$	allocated[np][nr]	[0,65535]
		long		$nr \in [1,2]$, $np \in [1,2]$	max[np][nr]	[0,4294967295]
		float		$nr \in [1,2]$, $np \in [1,2]$		[0.0,4294967295.0]
Bellman Ford	75	int8	s	$s \in [2,10]$	a[s][s]	[-128,127]
		int		$s \in [2,7]$		[-32768,32767]
		long		$s \in [2,4]$		[-2147483648,2147483647]
		float		$s \in [2,4]$		[-2147483648.0,2147483647.0]
Binary search	44	int8	n	$n \in [2,116]$	a[n]	[-128,127]
		int		$n \in [2,54]$		[-32768,32767]
		long		$n \in [2,23]$		[-2147483648,2147483647]
		float		$n \in [2,23]$		[-2147483648.0,2147483647.0]
Bubble Sort	35	int8	n	$n \in [1,116]$	a[n]	[-128,127]
		int		$n \in [1,54]$		[-32768,32767]
		long		$n \in [1,23]$		[-2147483648,2147483647]
		float		$n \in [1,23]$		[-2147483648.0,2147483647.0]
Dijkstra	82	int8	s	$s \in [2,9]$	a[s][s]	[-128,127]
		int		$s \in [2,5]$		[-32768,32767]
		long		$s \in [2,3]$		[-2147483648,2147483647]
		float		$s \in [2,3]$		[-2147483648.0,2147483647.0]
Floyd-Warshall	29	int8	s	$s \in [1,10]$	a[s][s]	[0,255]
		int		$s \in [1,7]$		[0,65535]
		long		$s \in [1,5]$		[0,4294967295]
		float		$s \in [1,5]$		[0.0,4294967295.0]
GCD	32	int8	n, m	$n \in [2,120]$, $m \in [2,120]$	-	-
		int		$n \in [2,32768]$, $m \in [2,32768]$		
		long		$n \in [2,2147483647]$, $m \in [2,2147483647]$		
		float		$n \in [2,2147483647]$, $m \in [2,2147483647]$		
Insertion Sort	35	int8	n	$n \in [2,116]$	a[n]	[-128,127]
		int		$n \in [2,54]$		[-32768,32767]
		long		$n \in [2,23]$		[-2147483648,2147483647]
		float		$n \in [2,23]$		[-2147483648.0,2147483647.0]
Kruskal	129	int8	s	$s \in [2,10]$	a[s][s]	[-128,127]
		int		$s \in [2,6]$		[-32768,32767]
		long		$s \in [2,4]$		[-2147483648,2147483647]
		float		$s \in [2,3]$		[-2147483648.0,2147483647.0]
Matrix Multiplication	34	int8		[1,6]		[-128,127]
		int	rowA, colA	[1,4]	a[rowA][colA]	[-32768,32767]
		long	rowB, colB	[1,2]	b[rowB][colB]	[-2147483648,2147483647]
		float		[1,2]		[-2147483648.0,2147483647.0]
Mergesort	84	int8	n	$n \in [1,57]$	a[n]	[-128,127]
		int		$n \in [1,25]$		[-32768,32767]
		long		$n \in [1,11]$		[-2147483648,2147483647]
		float		$n \in [1,6]$		[-2147483648.0,2147483647.0]
Quicksort	55	int8	n	$n \in [1,36]$	a[n]	[-128,127]
		int		$n \in [1,17]$		[-32768,32767]
		long		$n \in [1,6]$		[-2147483648,2147483647]
		float		$n \in [1,5]$		[-2147483648.0,2147483647.0]
Selection Sort	29	int8	n	$n \in [2,116]$	a[n]	[0,255]
		int		$n \in [2,54]$		[0,65535]
		long		$n \in [2,23]$		[0,4294967295]
		float		$n \in [2,23]$		[0.0,4294967295.0]

the values of a single combination. Finally, the module creates a directory that contains every header file.

5.3.4 Profiling on the Host Architecture (Host Profiling)

After the inputs generation phase, a tool to count the number of executed C statements is needed (i.e., $CS(z_i, b_{i,k})$ in Eqn. 5.17)). This value is obtained by performing a profiling of the benchmark functions by means of the *GCov* [3] profiler for each generated input. The functions have been compiled with GCC, using *-fprofile-arcs* and *-ftest-coverage* compilation flags. These flags tell the compiler to generate other information needed by *GCov* in order to make correct profiling. The first flag allows the generation of a *.gcda* file that has more information for each branch of the program, while the second one adds information to count the number of times a statement has been executed. The compilation will trigger the creation of a *.gcno* file and generate the corresponding *.gcda* file. To complete the task, the *GCov* command has been executed. The profiling will be done correctly only if the above-described files were generated and reachable.

The total number of executed C statements for each function is simply the sum of the single profiling numbers associated with each statement. It is worth noting that such profiling is performed one-time on the host platform since it is independent of the target processor.

5.3.5 Profiling on the Target Processor (ISS Execution)

The last data needed to calculate the J4CS metric is the number of assembly instructions executed by the target processor for each function and input set in the benchmark (i.e., $I(p_j, z_i, b_{i,k})$ in Eqn. 5.17). So, for each target processor there is the need for an *Instruction Set Simulator* (ISS).

5.4 Processor Specific Framework: Two Examples

J4CS has been evaluated by considering some specific processors (i.e., p_j in Eqn. 5.17. In this work, as done in [143], two processors have been analyzed: the Cobham Gaisler LEON3 micro-processor [6], and the Intel 8051 micro-controller [8].

5.4.1 LEON3 micro-processor

LEON3 [6] is a 32-bit synthesizable soft-processor that is compatible with SPARC V8 architecture: it has a seven-stage pipeline and Harvard architecture, uses separate instruction and data caches and supports multiprocessor configurations. It represents a soft-processor for aerospace applications. Cobham Gaisler offers TSIM System Emulator [9] as an accurate emulator of LEON3 processors. A free evaluation version of TSIM/LEON3 is available on Gobham website [9], but it does not support code coverage, configuration of caches, memories and so on. Anyway, it has been chosen as the reference ISS for first analysis since it provides the information needed to evaluate J4CS. The LEON3 version has a default simulated system clock of 50 MHz. The evaluation version of TSIM/LEON3 implements 2*4 KiB caches (not removable), with 16 bytes per line with Least-Recently-Used (LRU) replacement algorithm. It has 8 register windows, a RAM size of 4096 KiB and a ROM size of 2048 KiB. By default, TSIM/LEON3 emulates the FPU. Benchmark functions have been compiled, with the Bare-C Cross-Compiler (BCC) for LEON3 processors [4]. It is based on the GNU compiler tools and the New-lib standalone C-library. BCC is composed by GNU GCC C/C++ compiler 4.4.2, GNU Binutils 2.19.51 and Newlib C-library 1.13.1. After the simulation, the framework is ready to calculate the metric and some statistics by considering all the inputs used to stimulate the functions.

5.4.2 LEON3 Statistical Analysis

Table 5.3 shows the Pearson correlation and slope values between executed assembly instructions and executed C statements. It is possible to note that the correlation is close to 1, while the slope indicates that the estimation uncertainty depends on the number of data input bits. Furthermore, some specific functions are more sensitive to data types compared to other ones. These last results depend on functions implementation (number of branches, loop, and complex arithmetic operations).

Table 5.3: LEON3 Statistical Analysis results.

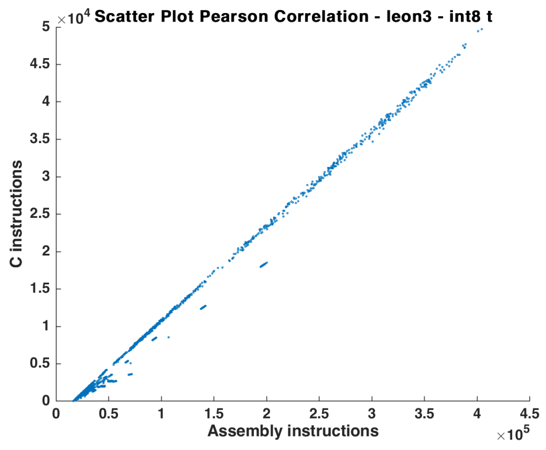
	Data Type Corr.[1]					**Data Type Slope**[3]				
Function	**int8**	**int16**	**int32**	**float**	**Corr. Tot.**[2]	**int8**	**int16**	**int32**	**float**	**Slope Tot.**[4]
A*	0.997130176	0.995528883	0.997342958	0.999199314	0.997558728	0.0890	0.0854	0.0818	0.0698	0.0878
Banker's Algorithm	0.976985991	0.971856389	0.973122344	0.983442876	0.979447788	0.0730	0.0736	0.0774	0.0729	0.0738
Bellman Ford	0.996911126	0.994174207	0.999111052	0.999758625	0.993600677	0.0694	0.0943	0.1162	0.0733	0.0697
Binary Search	0.999249964	0.998686622	0.997520361	0.993549505	0.904750109	0.1545	0.2294	0.2226	0.1692	0.1529
Bubble Sort	0.999886591	0.99998281	0.999912836	0.999845827	0.999198011	0.1266	0.1484	0.1517	0.1128	0.1270
Dijkstra	0.999239588	0.998580215	0.99865452	0.999434198	0.998348213	0.0892	0.0947	0.0879	0.0798	0.0908
Floyd Warshall	0.997867119	0.999069696	0.994619228	0.997353347	0.997684756	0.0659	0.0605	0.0693	0.0642	0.0661
GCD	0.999986897	0.997729568	0.999092395	-	0.958592379	0.1534	0.2003	0.1981	-	0.1415
Insertion Sort	0.999899431	0.999667128	0.998299365	0.99743627	0.998106672	0.1075	0.1397	0.1496	0.1177	0.1083
Kruskal	0.999692859	0.999778195	0.999659212	0.999766269	0.999700213	0.1007	0.1068	0.1151	0.0970	0.1013
Matrix Mult.	0.991026869	0.987504129	0.986854719	0.99465194	0.990567396	0.0570	0.0510	0.0714	0.1265	0.0584
Merge Sort	0.999806391	0.999832814	0.999806429	0.999840959	0.994096345	0.0957	0.1275	0.1225	0.0886	0.0979
Quick Sort	0.999577773	0.999652397	0.999680258	0.999595041	0.999137649	0.1174	0.1279	0.1321	0.1099	0.1192
Selection Sort	0.999924321	0.999882252	0.999821454	0.999667636	0.997105498	0.1151	0.1560	0.1675	0.1287	0.1161
Tot.[5]	0.99361	0.98178	0.96865	0.98326	0.992487927	0.1218	0.1391	0.1289	0.1049	0.1215

[1] Corr.: Pearson Correlation; [2] Corr. Tot.: Total Pearson Correlation considering all data types;
[3] Slope: Regression Slope Parameter; [4] Slope Tot.: Total Regression Slope considering all data types;
[4] Tot.: Total data set (considering all functions)

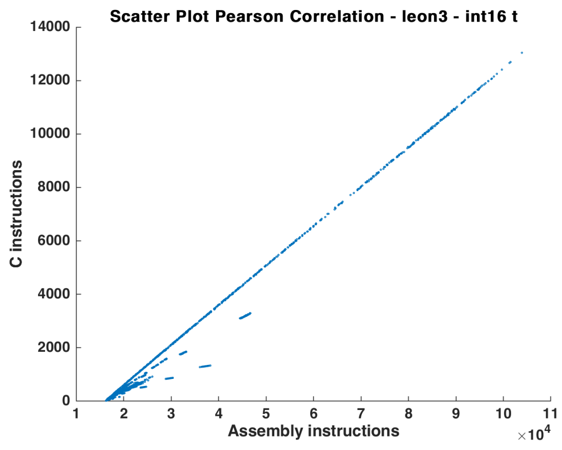
Fig. 5.5 shows the scatter plot related to the Pearson correlation. The plots show a strict correlation between C statements and Assembly instructions, due to the Sparc-V8 RISC ISA architecture, as reported in [59]. Regarding to the other points (the ones under the principal linear regression line), the deviation depends on different data types and functions implementations that introduce different behaviors compared to the prominent distribution. These points contribute to the introduction of errors inside the energy estimation activity, and are dependent on the LEON3 processor internal micro-architecture (i.e., 32 bit architecture, pipeline, number of registers).

5.4.3 8051 micro-controller

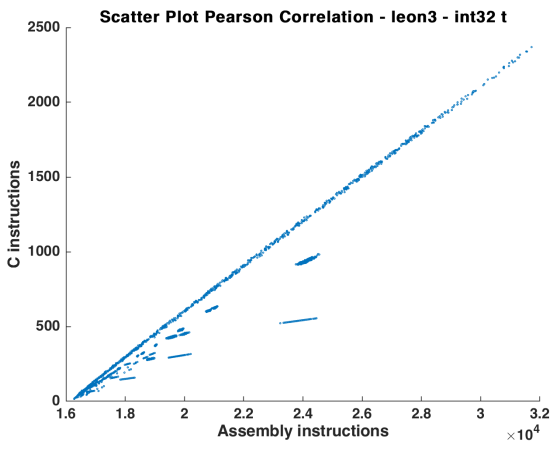
The Intel 8051 micro-controller is built around an 8-bit CPU. Architectural model used is the Harvard Architecture, and therefore it partitions data and instruction by the use of two memories and two buses; indeed 8051 presents a PROM non-volatile memory which contains program instruction and a RAM memory for data. Furthermore, it presents an 8-bit Data Bus and a 16-bit Address Bus. I8051 registers are 8-bit registers. ALU works with 8-bit words and is provided with an accumulator register and communicates with four I/O 8-bit ports. The University of California has developed a project centered on 8051 microprocessor, which provides a number of tools useful for



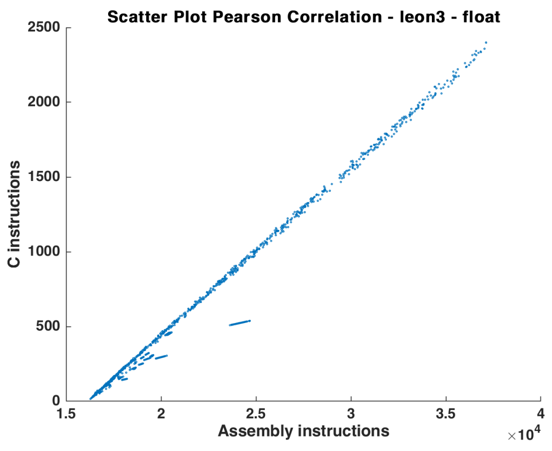
(a) LEON3 int8 Corr.



(b) LEON3 int16 Corr.



(c) LEON3 int32 Corr.



(d) LEON3 float Corr.

Figure 5.5: Pearson Correlation Plot for LEON3.

simulating C code on Intel 8051 microprocessor. The project name is Dalton and was developed by the Department of Computer Science of the University of California [12]. The Dalton Instruction Set Simulator (ISS) allows a user to simulate programs written for the 8051 and provides statistics on instructions executed, instructions executed per second, clock cycles required by the 8051, and average instructions per second for an 8051 executing the same program. For these characteristics, it has been chosen as the reference ISS for the measurement of the J4CS for 8051 microprocessor. The functions were compiled, with SDCC (Small Device C Compiler) [7]. SDCC is a free open source C compiler suite designed for 8 bit Microprocessors. The entire source-code for the compiler is distributed under GPL and has extensive language extensions suitable for utilizing various micro-controllers and underlying hardware. The Dalton ISS needs a .hex to do the simulation. This kind of file was generated with SDCC. In order to do a proper simulation, during the compilation two options was specified: *–mmcs51* and *–iram-size 128.* The first one refers to the family of the microprocessor while the second to the dimension of the internal RAM. The compilation generates an *.ihx* file that can be converted to *.hex* file using the *packihx* command. At the end, it is possible to execute the ISS. It generates a file that contains information about the simulation. After the simulation, the framework is ready to calculate the metric and some statistics on all input generated for the functions. These calculations are made with a program that returns two files containing metric values, for each input, and statistics on the sample.

5.4.4 8051 Statistical Analysis

Table 5.4 shows Pearson correlation and slope between executed assembly instructions and executed C statements in details. Differently from LEON3 results, it is possible to note that the correlation is not so close to 1 (there are also values that are under 0.9), while the slope indicates that the estimation uncertainty depends on the number of data input bits. Some specific functions are more sensitive to data types compared to other ones (behavior similar to LEON3 processor). This last results depends on different functions implementations (number of branches, loop, and complex arithmetic operations), and the fact that 8051 has an 8-bit internal architecture, and no Floating Point Unit.

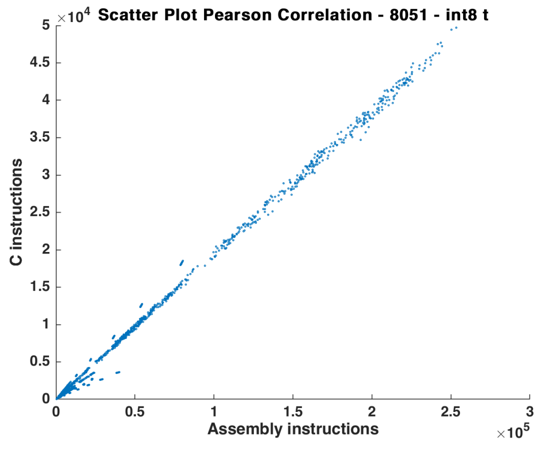
Table 5.4: 8051 Statistical Analysis results.

	Data Type Corr.[1]					**Data Type Slope**[3]				
Function	**int8**	**int16**	**int32**	**float**	**Corr. Tot.**[2]	**int8**	**int16**	**int32**	**float**	**Slope Tot.**[4]
A*	0.999018373	0.84914453	0.998332704	0.999596943	0.830460356	0.1955	0.0582	0.1048	0.0546	0.1376
Banker's Algorithm	0.966167708	0.961797134	0.950479804	0.971077972	0.340041773	0.1596	0.1060	0.0634	0.0201	0.0106
Bellman Ford	0.99794459	0.99543516	0.948538448	0.999616672	0.852702454	0.1098	0.0633	0.0363	0.0222	0.0976
Binary Search	0.999560803	0.999687367	0.999743356	0.985202744	-0.08002339	0.2128	0.1353	0.0783	0.0183	-0.002
Bubble Sort	0.999655934	0.999585159	0.998940075	0.596901958	0.974990261	0.1923	0.1405	0.0814	0.0215	0.1905
Dijkstra	0.99971141	0.9995696	0.998715493	0.998811434	0.539663795	0.1714	0.1316	0.0901	0.0295	0.0759
Floyd Warshall	0.999881657	0.999075444	0.999106673	0.99911282	0.810045772	0.0913	0.0827	0.0506	0.0171	0.0632
GCD	0.890585301	0.899892026	0.952382673	-	0.68826302	0.1734	0.1601	0.1212	-	0.1466
Insertion Sort	0.999950396	0.999902786	0.999545234	0.994771407	0.973606759	0.2592	0.1414	0.0759	0.0925	0.2503
Kruskal	0.999856993	0.999892487	0.999837653	0.999267543	0.991654051	0.2277	0.1709	0.1255	0.0533	0.2334
Matrix Mult.	0.99952095	0.995791004	0.987974866	0.956583715	0.869857949	0.1322	0.0619	0.0398	0.0206	0.1039
Merge Sort	0.999623696	0.999613733	0.999572569	0.9993614	0.71989699	0.1485	0.1001	0.0656	0.0122	0.0979
Quick Sort	0.999646522	0.999695728	0.999281295	0.997228213	0.905400216	0.1354	0.0919	0.0568	0.0242	0.1249
Selection Sort	0.999717704	0.999385492	0.998837314	0.99584793	0.966448824	0.2041	0.1098	0.0591	0.1083	0.1927
Tot.[5]	0.99293	0.98799	0.92563	0.74391	0.960083884	0.1937	0.1384	0.0732	0.0245	0.1783

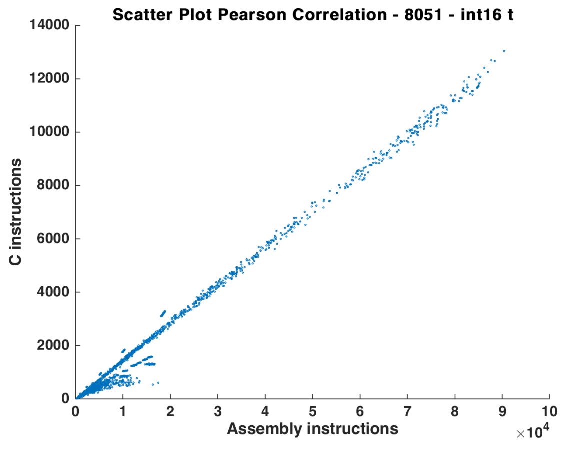
[1] Corr.: Pearson Correlation; [2] Corr. Tot.: Total Pearson Correlation considering all data types;
[3] Slope: Regression Slope Parameter; [4] Slope Tot.: Total Regression Slope considering all data types;
[4] Tot.: Total data set (considering all functions)

The different correlation and slope values in Table 5.4 (and even negative for binary search algorithm) mean that the values are very sparse in the scatter-plot. In the case of a single function, putting together all the types of data, many straight lines for each data type have been founded, with different slopes and above all different weights (the number of points from which each line is composed), while the experimental data are arranged very well on the linear regression straight line for data types (i.e., int8, int16, int32, and float). The more the correlation is low or even negative, the more the lines by data type are open to each other, with different numbers of points and also not common intercepts for the Y axis of the graph. From Table 5.4 it is possible to note also that the worst values are associated to float data types, since the 8051 did not have Floating Point Unit.

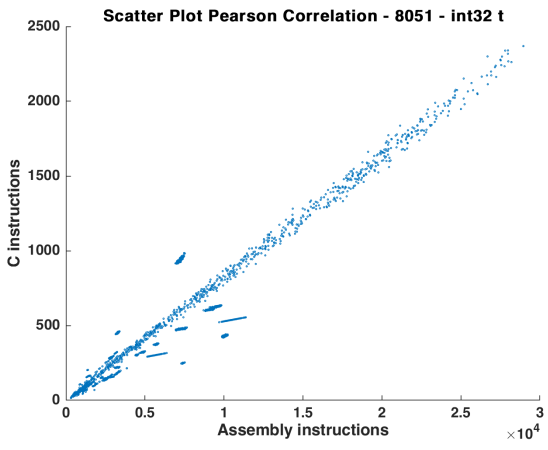
Fig. 5.6 shows the scatter plot related to the Pearson correlation. Compared to LEON3 scenarios, the correlation point distributions is not so linear as the LEON3 scenarios. This sparse points distribution depends on 8051 internal processor architecture (8-bit 8051 CISC ISA). Another difference is the internal RAM that the 8051 (and Dalton ISS [12]) has compared to the external TSIM LEON3 RAM memory. This internal 128 KB RAM limits the data input ranges, and the possible test-bench simulation activities. Meanwhile, there is a similar points placing behaviors with respect to LEON3 scenarios (i.e., the isolated points under the linear regression line) that introduce errors in the estimation activities.



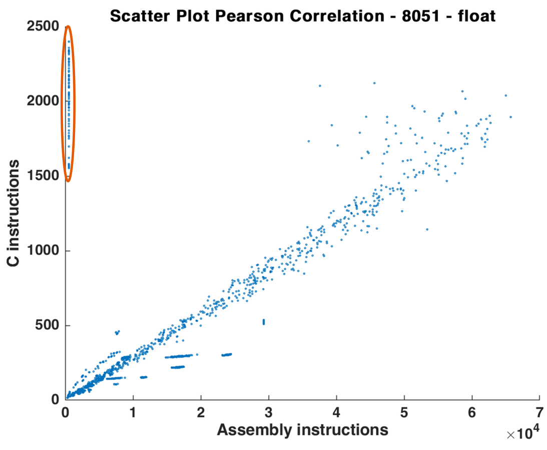
(a) 8051 int8 Corr.



(b) 8051 int16 Corr.



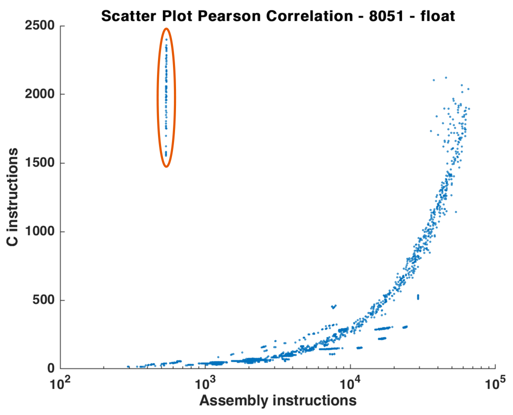
(c) 8051 int32 Corr.



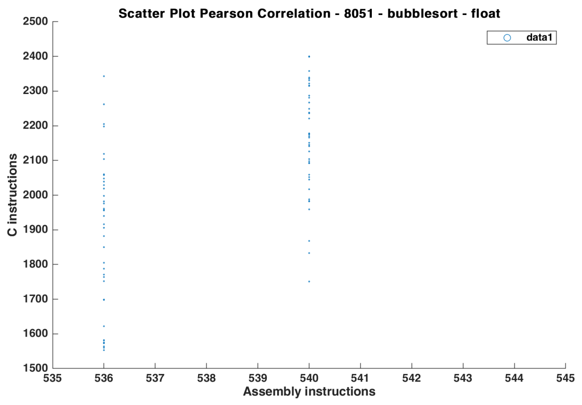
(d) 8051 float Corr.

Figure 5.6: Pearson Correlation Plot for 8051.

Furthermore, the Fig. 5.6d is the only plot that has a strange point cluster (orange circle). Fig. 5.7a shows the 8051 float correlation with x-axis in logarithmic scale. It is worth noting that there are some points outside the main regression line. These points are related to one function, the *Bubblesort*. Fig. 5.7b presents the correlation plot for the *bubblesort* function in more details. From the graph, it is possible to note that there are 2 fixed assembly instructions values corresponding to a different number of executed C statements. This behaviors is not normal, while in the other case (int8, int16 and int32) the points are close to the regression line and they do not follow a strange pattern. This problem is probably due to errors in the ISS compilation/execution so they have been deleted from the dataset and not considered.



(a) 8051 Float Corr. (Log Scale)



(b) 8051 Bubblesort Float Corr.

Figure 5.7: Pearson Correlation Plot for 8051.

5.4.5 Use Case Scenario

In order to evaluate results in a real scenario, three boards have been considered by taking voltage and frequency information (the power information can be found in the processor/board data-sheets): LEON3FT-RTAX [5], LEON3FT-UT699 Single-Core SOC [10] and AT89C51 ATMEL Development Board [1] (based on 8051 architecture). The processors parameters used to evaluate J4CS [6] are shown in Table 5.5.

Table 5.5: Board characteristics.

Parameters	RTAX	UT699	AT89C51
Clock	25 MHz	66 MHz	12 MHz
V_{dd}	3,3 V	3,6 V	6,0 V
$\bar{P}$	500 mW	178,2 mW	600 mW
MIPS	20	75	2
ϕ	0,025	0,002376	0,3

It is worth noting that LEON3-FT is a System-On-Chip design based on LEON3FT core, and it has the same ISA of the classical LEON3 processor. Therefore, the number of assembly instructions executed by the ISS is the same for both processors since they rely on the same compiler. So, considering these characteristics, the average energy consumption associated to each executed assembly instruction is: ($\bar{E}_{RTAX} = 0,8\ nJ/Instr.$, $\bar{E}_{UT699} = 1,1364\ nJ/Instr.$ and

$\bar{E}_{AT89C51} = 0,16\ nJ/Instr.$). The obtained results for the executions of the benchmark functions are summarized in Table 5.6.

Table 5.6: J4CS measured on LEON3FT-RTAX, UT699 and AT89C51 Board (in nJ)

Data Type	**Min**	Q_1^1	**Median**	Q_3^2	**Max**	**AM**[3]	**SD**[4]	**Var**[5]	**GM**[6]	**85%**[7]	**95%**[8]
LEON3FT-RTAX int8	1	17	36	148	869	100.73	137.03	18779	48.33	220	338
LEON3FT-RTAX int16	1	32	67	202	868	140.66	173.38	30061	75.33	312	523
LEON3FT-RTAX int32	4	58	129	299	868	213.11	208.60	43513	131.98	451	814
LEON3FT-RTAX float	4	86	202	452	869	270.45	240.16	57676	173.18	597	814
LEON3FT-RTAX AVG	5	48.25	108.5	241.5	869	181.24	189.79	37507	107.20	348.25	622.25
UT699 int8	1	25	51	211	1234	143.07	194.63	37882	68.549	312	481
UT699 int16	5	46	95	286	1233	199.79	246.29	60661	106.9	359	743
UT699 int32	5	83	184	424	1233	302.72	296.29	87789	187.45	564	1156
UT699 float	5	123	287	642	1235	384.22	341.13	1.1637e+05	246.1	743	1157
UT699 AVG	4	69.25	154.25	390.75	1233.75	257.45	269.58	68530.25	152.25	494.5	884.25
AT89C51 int8	1	2	2	3	14	2.2705	1.2716	1.6169	2.0124	3	5
AT89C51 int16	1	2	2	3	19	2.9666	1.413	1.9967	2.7245	4	6
AT89C51 int32	2	3	3	5	24	5.3924	5.2433	27.492	4.1399	8	23
AT89C51 float	3	5	6	11	43	9.8238	8.3489	69.704	7.7229	13	33
AT89C51 AVG	1.75	3	3.25	5.5	25	5.1133	4.0692	25.202	4.149925	7	16.75

[1]Q_1: First Quartile; [2]Q_3: Third Quartile; [3]AM: Arithmetic Mean; [4]SD: Standard Deviation; [5]Var: Variance; [6]GM: Geometric Mean; [7]85%: 85th Percentile; [8]95%: 95th Percentile;

For each function, different data types have been considered (*int8, int16, int32,* and *float*). Indeed, both timing [143] and energy, especially their average values, change with respect to the dimension of data.

Fig. 5.8, Fig. 5.9 and Fig. 5.10 show the distribution related to J4CS evaluated for RTAX, UT699 and AT89C51 boards, according to the reference benchmark. The described evaluation process of J4CS for the three boards has required a total of near 12 hours on a standard workstation (Intel i7, 1.5 GHz, 16 GB RAM). However, as highlighted before, this is a one-time effort to make available J4CS for subsequent analysis (as shown in the next section).

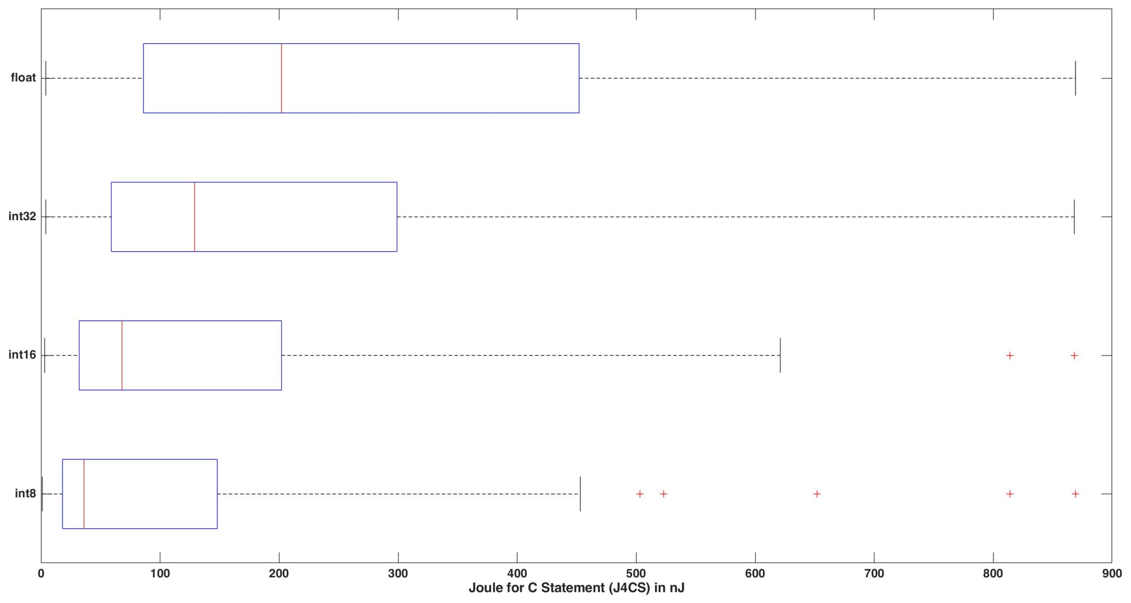


Figure 5.8: J4CS BoxPlot results for LEON3 (LEON3FT-RTAX board).

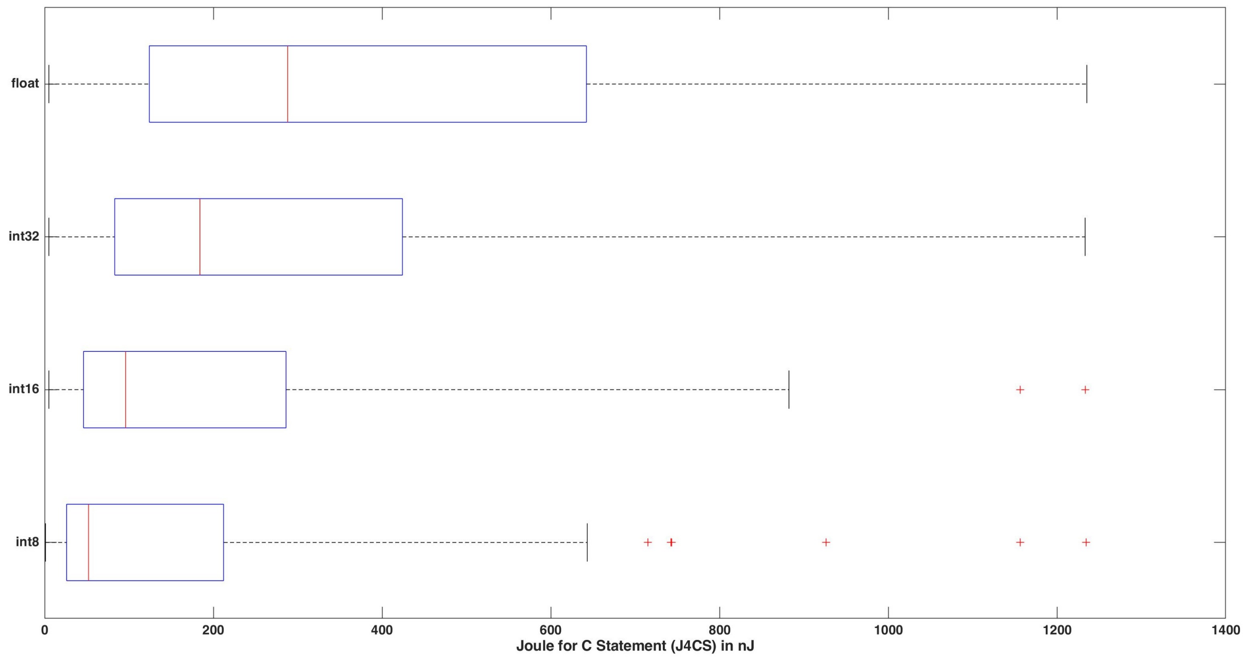


Figure 5.9: J4CS BoxPlot results for LEON3 (UT699 board).

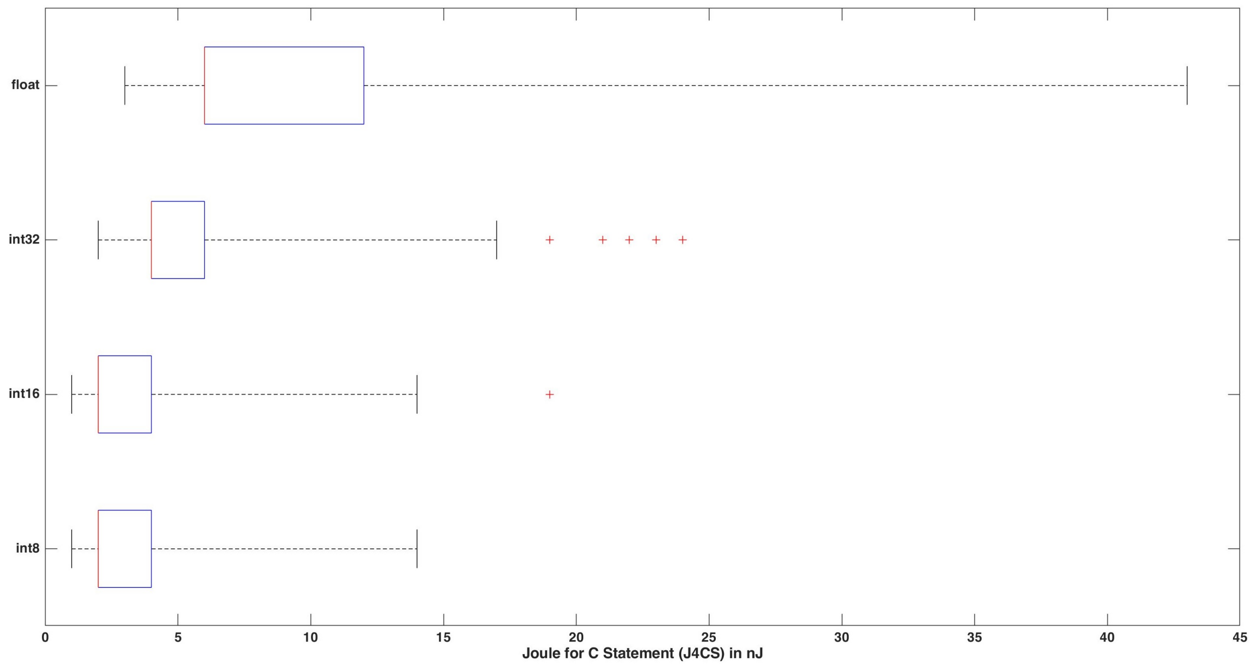


Figure 5.10: J4CS BoxPlot results for 8051 (AT89C51 board).

5.5 J4CS-based Energy Consumption Estimation

The availability of J4CS is very useful for very fast early-stage estimation, comparison and selection. Indeed, by having available J4CS for different processors, with a single host-based profiling it is possible to estimate the energy consumption of a function of interest for the whole processors set, giving very fast preliminary comparison and selection activities. As an example, strating from a target function *tf()* and considering a specific *golden input* $\boldsymbol{x}$, by means of a host-based profiling (that takes less than a second on the same workstation described in the previous section), it is possible to count the number of executed C statements during the execution of tf($\mathbf{x}$) (e.g., 100). Then, as shown in Fig. 5.11 (the x-axis is in a logarithmic scale), it is straightforward to compare the whole processors set by multiplying 100 for the related J4CS. Depending on a possible energy consumption constraint it is then possible to select a specific processor or, at least, to reduce the set to few of them in order to be considered for further analyses.

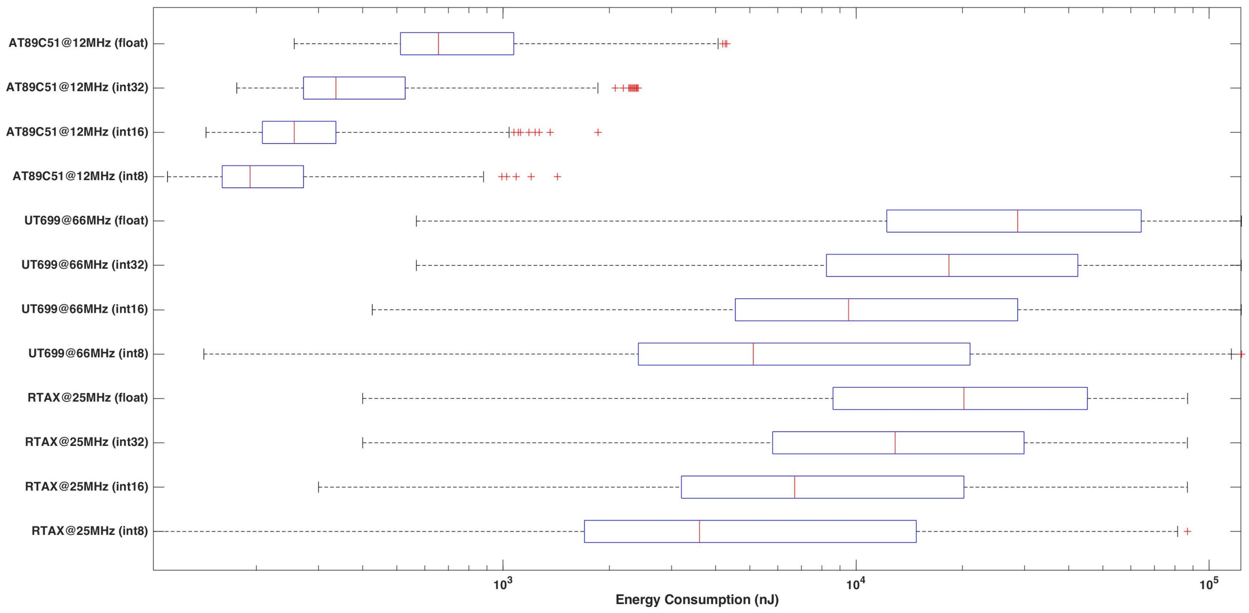


Figure 5.11: J4CS-based SW comparison.

5.5.1 Validation

In order to validate the proposed metric, an error estimation evaluation has been performed with respect to the benchmark. Table 5.7 shows some results related to AT89C51 board. It is worth noting that the error depends on the specific J4CS considered (in this example we considers the first quartile, the arithmetic mean, the median and the third quartile). Errors ranges are highly variable, where median errors are less than other ones, depending on assembly and C statements distributions. In order to reduce errors and variance associated to the estimations, a further assumption can be considered.

Fig. 5.8, Fig. 5.9 and Fig. 5.10 present the specific processor characterization w.r.t. different boards. It is possible to fix the J4CS value introducing the concept of "Affinity" defined in [51]. Since the execution time of different functions depends on some architectural features of specific executors classes, this dependency can be defined using the "Affinity" metric, which suggests the

Table 5.7: AT89C51 board Relative Error Results (in %).

	Int8				**int16**				**int32**				**float**			
Function	$^1Q_1$	**Median**	^{2}**AM**	$^3Q_3$	$^1Q_1$	**Median**	^{2}**AM**	$^3Q_3$	$^1Q_1$	**Median**	^{2}**AM**	$^3Q_3$	$^1Q_1$	**Median**	^{2}**AM**	$^3Q_3$
A*	4.99	4.99	-7.83	-42.5	46.34	46.34	22.20	19.51	21.33	21.33	-41.32	-31.10	8.31	-10.01	-79.69	-101.69
Banker's Algorithm	31.01	31.01	21.70	-3.47	35.67	35.67	6.73	3.51	43.02	43.02	-2.36	5.04	40.99	29.19	-15.64	-29.80
Bellman Ford	14.22	14.22	2.64	-28.66	25.47	25.47	-8.06	-11.78	2.97	2.97	-74.32	-61.71	18.68	2.42	-59.37	-78.89
Binary Search	30.48	30.48	21.10	-4.26	24.87	24.87	-8.92	-12.68	35.35	35.35	-16.14	-7.74	83.55	80.26	67.77	63.82
Bubble Sort	-12.27	-12.27	-27.43	-68.41	6.53	6.53	-35.51	-40.19	1.77	1.77	-76.47	-63.70	23.31	7.98	-50.29	-68.69
Dijkstra	1.67	1.67	-11.60	-47.49	10.47	10.47	-29.80	-34.28	-5.10	-5.10	-88.84	-75.18	18.15	1.78	-60.42	-80.06
Floyd Warshall	53.49	53.49	47.21	30.23	61.88	61.88	44.73	42.83	67.41	67.41	41.44	45.68	50.41	40.50	2.81	-9.07
GCD	-55.38	-55.38	-76.36	-133.07	48.96	48.96	25.99	23.44	85.59	85.59	74.11	75.98	-	-	-	-
Insertion Sort	-40.74	-40.74	-59.74	-111.11	-5.90	-5.90	-53.56	-58.86	-11.25	-11.25	-99.88	-85.41	-26.54	-51.85	-148.03	-178.40
Kruskal	-19.51	-19.51	-35.65	-79.27	-12.11	-12.11	-62.56	-68.17	-42.31	-42.31	-155.69	-137.19	-37.12	-64.55	168.76	201.67
Matrix Mult.	38.20	38.20	29.86	7.31	45.34	45.34	20.75	18.02	48.16	48.16	6.86	13.60	29.78	5.74	-37.61	-54.46
Merge Sort	-10.49	-10.49	-25.40	-65.73	14.99	14.99	-23.26	-27.51	16.84	16.84	-49.39	-38.58	57.78	49.34	17.26	17.26
Quick Sort	-0.96	-0.96	-14.59	-51.45	20.05	20.05	-15.92	-19.91	11.88	11.88	-58.31	-46.85	20.28	4.34	-56.24	-75.37
Selection Sort	-51.17	-51.17	-71.57	-126.75	-19.33	-19.33	-73.03	-79.00	-33.97	-33.97	-140.71	-123.29	-35.544	-62.65	-165.66	-198.19
Tot. Rel.[4]	1.17	1.17	-14.83	-51.76	21.65	21.65	13.58	17.50	17.16	17.16	-48.64	-38.86	19.42	2.49	-32.02	-45.51
Tot. Abs.[5]	25.90	25.90	32.33	57.12	26.99	26.99	30.78	32.83	30.49	30.49	66.13	56.96	34.64	31.58	71.50	89.02

$^1Q_1$: First Quartile; ^{2}AM: Arithmetic Mean; $^3Q_3$: Third Quartile;
[4] Tot. Rel: Total Relative Error (all functions); [5] Tot. Abs.: Total Absolute Error (all functions)

most suitable processor class for the execution of a given functionality. This value, in the range of 0 and 1, provides a quantification of the matching between the structural and functional features of the functionality implemented by the considered processor classes function and the architectural features.

Starting from the affinity value, it is possible to refine the J4CS definition using the following equations:

Definition 5.5.1. *J4CS-A (Joule for C Statements considering Affinity metric).* Considering a single C function z_i, with a specific affinity value $A_{i,j}$ evaluated with the method proposed in [51], and the $J4CS(p_j)$ distribution evaluated for a specific processor, as presented in Eqn 5.17, it is possible to chose a fixed value for $J4CS - A(p_j, A_{i,j})$ depending on an "Affinity" value and distribution parameters [{Min, Q_1, Med (Median), Q_3, Max} of $J4CS(p_j)$ from Table 5.6]. Three different scenarios are considered:

1. Best Case

$$\begin{aligned} J4CS - A(p_j, A_{i,j}) &= 2 \cdot (Med - Q_3) \cdot A_{i,j} + Q_3 && If\ A_{i,j} < 0.5 \\ J4CS - A(p_j, A_{i,j}) &= Med + 2 \cdot (A_{i,j} - 0.5) \cdot (Q_1 - Med) && If\ A_{i,j} \geq 0.5 \end{aligned} \tag{5.18}$$

2. Average Case

$$\begin{aligned} J4CS - A(p_j, A_{i,j}) &= 2 \cdot (Med - [Q_3 + \alpha \cdot IQR]) \cdot A_{i,j} + (Q_3 + \alpha \cdot IQR) && If\ A_{i,j} < 0.5 \\ J4CS - A(p_j, A_{i,j}) &= Med + (A_{i,j} - 0.5)(Q_1 - \alpha \cdot IQR) - 2 \cdot Med && If\ A_{i,j} \geq 0.5 \end{aligned} \tag{5.19}$$

3. Worst Case

$$\begin{aligned} J4CS - A(p_j, A_{i,j}) &= 2 \cdot (Med\ -\ Max) \cdot A_{i,j} + Q_4 && If\ A_{i,j} < 0.5 \\ J4CS - A(p_j, A_{i,j}) &= Med\ +\ 2 \cdot (A_{i,j} - 0.5)(Min - Med) && If\ A_{i,j} \geq 0.5 \end{aligned} \tag{5.20}$$

Eqn. 5.18-5.19-5.20 are derived considering a linear interpolation between affinity value and J4CS distribution. Fig. 5.12 shows the graphical representation of the equations above.

Table 5.8 shows the relative and absolute errors associated to the AT89C51 board where the affinity value has been introduced in order to reduce the estimation error. The total relative and absolute mean errors (considering all the functions in the benchmark and a test-bench composed of 100 inputs and executions for each functions) associated to the validation activity is in the range of $\{+/-0 \ldots 15\}$, with an highest error equal to about 34 %, and the error reduction (absolute and relative) compared to Table 5.7 is in the range of $\{50 \ldots 110\%\}$. The only worst situation is the float

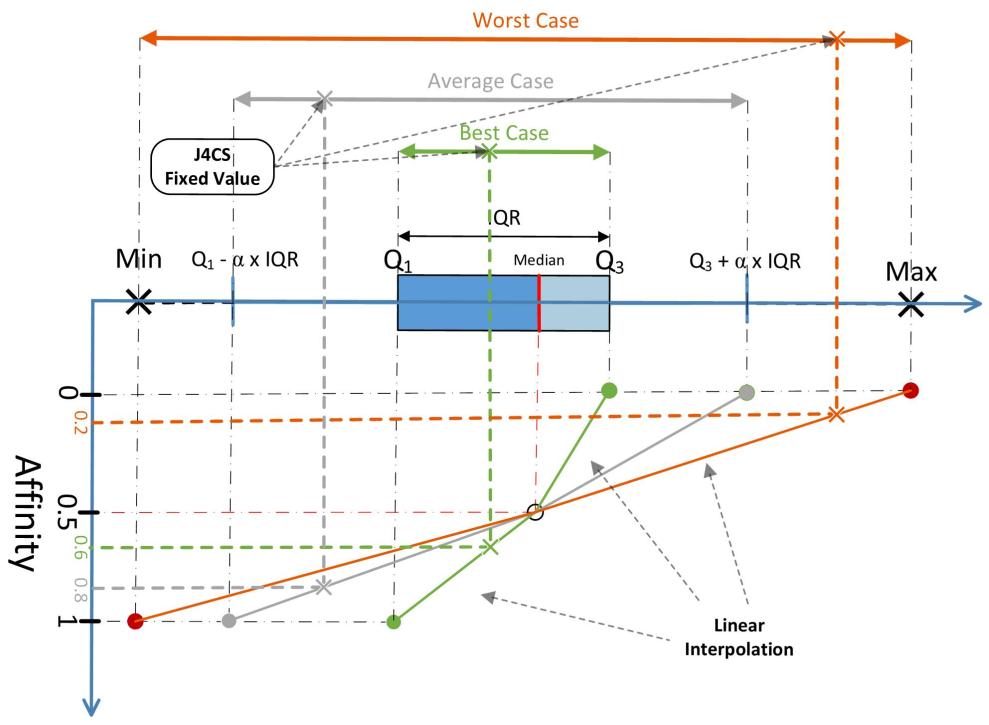


Figure 5.12: J4CS-based Refinement (considering affinity value).

case, where the error range is $\{+/-0.5\ldots18\%\}$. This is an interesting result since the estimation activity takes only few seconds (the time to profile the functions with the different inputs), without execute the specific functions on the reference target.

Finally, Fig. 5.13 shows the relative error results graph w.r.t. *Insertion Sort* function. In this case the errors are less then 10 %. The other functions have a similar behavioral pattern, with some functions that arrive to relative errors less then 15 % at most.

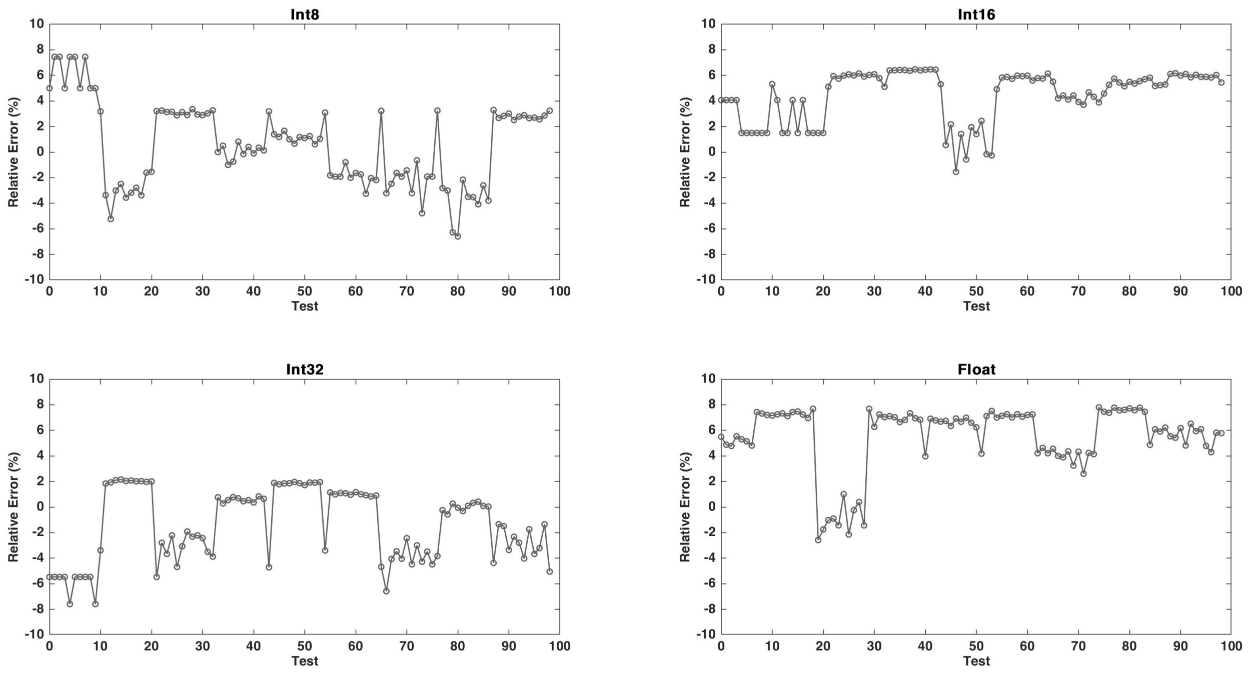


Figure 5.13: Relative Error plot compared to Insertion Sort function tests.

Table 5.8: AT89C51 board Error Results (in %) considering Affinity values.

	Int8			int16			int32			float		
Function	[1] **Aff.**	[2]**Rel.**	[3]**Abs.**	[1] **Aff.**	[2]**Rel.**	[3]**Abs.**	[1] **Aff.**	[2]**Rel.**	[3]**Abs.**	[1] **Aff.**	[2]**Rel.**	[3]**Abs.**
A*	0.5	4.9955	11.7152	0.45	6.7401	38.1508	0.48	0.6858	30.3601	0.441	-0.4464	7.4036
Banker's Algorithm	0.015	-2.4440	9.9131	0.47	2.8766	9.0229	0.445	-0.8453	9.1918	0.38	5.1252	13.2049
Bellman Ford	0.4	5.6460	14.5508	0.48	0.1357	17.6572	0.49	-10.6119	18.3243	0.425	-0.0185	24.3389
Binary Search	0.45	-11.2165	19.7001	0.48	-0.6609	5.5448	0.45	-9.8993	11.1606	0.1	34.88	40.07
Bubble Sort	0.6	-1.0513	2.5975	0.5	6.5398	7.4153	0.5	1.7746	2.9615	0.425	5.6814	7.4037
Dijkstra	0.5	1.6725	8.8011	0.5	10.4797	14.5653	0.56	-0.9043	9.2400	0.42	-0.6719	4.5762
Floyd Warshall	0.4	-2.3208	16.5335	0.4	-2.9037	17.2394	0.35	-1.0251	14.7806	0.31	-8.8814	16.6744
GCD	0.85	-1.0007	1.9384	0.45	5.5825	6.0357	0.02	-11.2415	14.7339	-	-	-
Insertion Sort	0.8	1.4812	2.5001	0.6	4.6832	4.8414	0.65	0.1265	1.9446	0.48	2.8127	6.7120
Kruskal	0.68	1.9949	14.7680	0.6	-0.9022	8.0750	0.9	-4.3662	6.8132	0.48	-5.3124	9.2739
Matrix Mult.	0.4	15.9635	21.0259	0.45	-1.1063	24.6616	0.45	11.8742	23.0171	0.4	-1.1025	15.5807
Merge Sort	0.6	0.5581	4.3580	0.5	14.9922	14.9922	0.5	16.8476	16.8476	0.3	3.7551	4.4141
Quick Sort	0.5	0.5581	4.3580	0.48	-7.1272	7.1272	0.5	11.8866	11.8866	0.4	-14.7891	14.7891
Selection Sort	0.8	-5.8196	6.1361	0.7	4.5302	4.5302	0.9	1.7502	1.7502	0.5	18.6732	19.1008
Tot. [4]	-	0.6441	9.9211	-	3.1328	12.8471	-	0.4323	12.3580	-	3.0543	14.1186
Tot. Red. AM[5]	-	-104.3430	-69.3129	-	-76.93	-58.262	-	-100.889	-81.313	-	-109.539	-80.254
Tot. Red. Med.[6]	-	-44.9518	-61.6945	-	-89.822	-52.401	-	-97.4809	-59.469	-	22.6611	-55.293

[1]Aff: Affinity; [2]Rel: Relative Error; [3] Abs: Absolute Error; [4] Tot.: Total Error (all functions);
[5] Tot. Red. AM: Total Reduced Relative and Absolute Error (reduced % respect to AM in Table 5.7);
[6] Tot. Red. Med.: Total Reduced Relative and Absolute Error (reduced % respect to Median in Table 5.7);

5.6 Related Work

Different abstraction levels can be considered in order to describe a processor and its behavior. Accordingly, several energy estimations can be performed [180], as previously stated. Starting from lower-levels of abstraction, such as gate or Register-Transfer (RT) ones [146], a lot of works consider the problem of estimate power/energy consumption by using time-consuming simulators [172][200][16]. Other works start from an accurate modeling of the target ISS [184][173], but this still requires a considerable time both for the modeling and the simulation activities.

Other studies present energy estimation approaches at (assembly) instructions level. These energy consumption estimation techniques usually consider either ISS or low-level assembly code analysis in order to obtain power characterization of the application, at the expense of a higher consumption of estimation time. Instruction-level energy estimation can be divided into two types: (1) measurement-based techniques that do not consider processors architectural details, while trying to extract an average energy cost per instruction [112][113]; (2) pipeline and accurate architectural analysis that take into account pipeline stages and complex architectural details [167][151][181]. The same problems arise in approaches that involve the introduction of some kind of *Virtual Instruction Set* (e.g., [49]), but that still require some explicit detailed knowledge of the target processors architecture.

Several works analyze the energy consumption of processor functional units, where the total energy consumption of a target processor is obtained by the sum of energy dissipation of these functional components [170]. Also for the so called *Functional-Level Power Analysis* (FLPA) methods, the energy model is derived by means of simulation or on-target measurements [125][122]. In order to bring the gap between instruction and functional level power estimation, Blume et al. [44] and Brandolese et al. [55] has presented a hybrid approach that combines these two methods to estimate energy consumption.

Other works try to rise the abstraction level, by going towards the system level one. This is often done by directly considering source-code [59], but also this kind of analysis can involve different time-consuming activities strictly related to the need of taking into account the peculiarities of the considered target processors. With respect to the source-code analysis, the works in [53][48] present a statement-level timing estimation that is used to evaluate power/energy metrics directly on the base of timing executions and profiling activities. A work that tries to fill up the gap between the reduced simulation time of high-level dynamic analysis of source-code, with the accuracy of low-level dynamic analysis, is [54], that introduces an intermediate pseudo instruction set for analyzing applications and HW architectures, using an approach still similar to [49]. Finally, a statement-level energy estimation based on GCC has been proposed in [46], where an higher absolute error was measured due to the simple and basic method used for the measurements. Table 5.9 presents a comparison of the considered works according to several features [121].

Table 5.9: Comparison of Considered Power Estimation Works.

Work	Year	Target	Simulator and/or Estimation Model	Accuracy	Abstraction Level	Benchmark
[172]	1999	ARM710a	ARMulator	5%	Cycle Level	Dhrystone
[200]	2000	VLIW processor	SimplePower	15%	Cycle Level	Custom
[16]	2002	ARM920	Armulator	N.A.	Cycle Level	MPEG4
[184]	1994	Intel 486DX2-S SPARClite 934	Custom	N.A.	Cycle Level	Custom
[173]	2001	StrongARM SA-1100 Hitachi SH-4	JouleTrack	$2\% \leq acc \leq 8\%$	Cycle Level	Custom
[112]	2001	ARM7TDMI	Regression Model	$2\% \leq acc \leq 6\%$	Instruction Level	Custom
[113]	2002	ARM7TDMI	Regression Model	$2\% \leq acc \leq 6\%$	Instruction Level	Custom
[167]	2000	VLIW Core	Math Model	$3\% \leq acc \leq 16\%$	Instruction Level	Custom
[151]	2005	ARM7TDMI	Math Model	$acc \leq 5\%$	Instruction Level	Custom
[181]	2009	LEON3	Custom Framework	N.A.	Instruction Level	Custom
[49]	2000	Motorola MC68000 ARM7TDMI	TOSCA OCCAM2	$1\% \leq acc \leq 20\%$	Instruction Level	ILC 16
[44]	2007	ARM926EJ-S C55x DSP	ARMulator XDS510PP Plus	$4\% \leq acc \leq 9\%$	Instr./Func. Level	Digital Signal Processing Tasks
[55]	2002	Intel i960JF Intel i960HD SPARClite MB86934 ARM7TDMI	Instruction Characterization	$acc \leq 9\%$	Instr./Func. Level	Custom
[170]	2002	TMS320C6201	Math Model	$acc \leq 4.2\%$	Functional Level	Digital Signal Processing Algorithms
[125]	2007	TMS320C6416	Functional Model	$acc \leq 10\%$	Functional Level	Custom
[122]	2005	TMS320C6416	Functional Model	$acc \leq 9\%$	Functional Level	FIR filter
[59]	2007	ARM9TDMI ARM TRM	SystemC Sim.	$acc \leq 11\%$	System Level	Custom
[53]	year	N.A.	Math Model	$acc \leq 11\%$	System Level	Custom
[48]	2010	ReISC III	Math Model	$acc \leq 6\%$	System Level	WCET suite
[54]	2002	Intel i486	Formal Model	$acc \leq 5\%$	System Level	Custom
[46]	2016	Tiva TM4C123G	Instruction Characterization	$acc \leq 30\%$	System Level	Custom
This Work	2019	RTAX UT699 AT89C51	TSIM, Dalton	$1\% \leq acc \leq 15\%$	System Level	Custom

In such a context, the work presented in this paper is close to the work presented in [59]. The main difference is that the purpose of this work is to reduce the time needed for estimation activities by means of a strategy that allows to quickly evaluate and select processors in an early-stage analysis. In fact, the estimation of the energy consumed during SW execution is very fast since it is based only on a host-based source-level profiling performed one-shot independently from the number of target processors considered. More detailed ISA-related analysis, if needed, can be then performed by focusing only on the selected processors.

5.7 Conclusion and Future Work

This work has presented a metric useful to estimate in an early-stage design phase, the energy consumption related to the execution of embedded SW on a target processor. Such a metric, called J4CS (Joule for C Statements) is good for very fast estimation, comparison and selection activities. Then, more accurate approaches at lower abstraction levels can be used for more precise and time-consuming estimations. Beyond the pure SW domain, this metric can be easily exploited into specific HW/SW Co-Design methodologies and tools (e.g., [142]), in order to consider energy requirements during system-level design space exploration. Indeed, it is worth noting that this metric can be evaluated also in the HW domain, by using *High-Level Synthesis* (HLS) tools and *Hardware Description Language* (HDL) simulators able to provide energy information as output. Such values can be used to substitute the $\overline{E}'(p_j) \cdot I(p_j, z_i, b_{i,k})$ numerator value in Eqn. 5.17. Moreover, J4CS can be also useful in ESL energy consumption estimation approaches that rely on the availability of an estimated energy consumption for each statement composing the ESL specification (e.g.,[42]). We chose the C language to define J4CS since it is the most popular language in the embedded systems domain. C is less abstract than most modern programming languages, e.g., Java and Python, and can be linked to assembly instructions executed by the underlying platform through its compiler. It is not the case with Java, for example, whose compiler generates Java bytecode executed by the Java Virtual Machine (JVM). The JVM makes Java portable; however, we needed to have a direct connection with the hardware platform, which influences the energy consumption of the HW/SW system. Thus, the C language helps reducing the abstraction introduced by modern programming languages and has a direct connection to the underlying platform. J4CS-type metrics can be developed by evaluating how abstractions, e.g., classes, virtual machines, libraries, and garbage collectors, impact energy consumption. This work could be replicated using the Rust programming language, which is proven to be fast and ideal for embedded systems development. Future works will concentrate on the validation of J4CS-A metric with respect to the energy consumption measured on the actual boards and choosing other functions, different from the reference ones. Other future activities will focus on reducing absolute error estimations, introducing more accurate statistical analysis and models able to better consider processor architectural features. ome interesting opportunities, still at early-stage, will be related to the use of HW profilers [137], in order to evaluate estimation errors directly on-target and to the combined exploitation firstly of the *Affinity* metric [51], so to reduce such errors by identifying a proper distribution subset, and secondly of a more detailed static analysis of source-code, in order to assign different weights to different statements.

Chapter 6

An Approach Using Performance Models for Supporting Energy Analysis of Software Systems

The ubiquity of ICT devices triggered a continuous digitalization of information, thus facilitating access, storage, and manipulation of data. ICT brought several benefits to society, such as monitoring the health conditions of people in real-time or having almost universal access to educational content. However, continuous digitization has also downsides. A considerable amount of information demands expensive resources for processing and storage, with a consequent rising need for energy to build and power ICT devices. As energy demand increases, the impact of ICT in terms of carbon dioxide (CO_2) emissions becomes significant [191]. Belkhir et al. estimate that ICT devices will produce 14% of global CO_2 emissions by 2040 [41].

As already discussed in 2018 by Georgiou et al [93] in the context of IoT systems, technology has made considerable advancements for increasing *hardware* power consumption savings, which however can be undermined by poor design decisions at the *software* level. Software energy optimization is a hard endeavor, where multiple (and frequently conflicting) design and implementation decisions can influence the energy footprint of the software [191]. Making developers aware of the impact of their decisions on the energy consumption of their software is fundamental to cutting it down [82]. However, energy optimization cannot be pursued in isolation, because it may negatively impact on other non-functional attributes of software and, in particular, on performance. Hence, software design decisions have to induce acceptable tradeoffs between the satisfaction of performance requirements and power consumption savings [86]. Energy/performance tradeoffs can be analyzed by measurement-based experiments, although they can be very time-consuming and they need contextual conditions to be taken under control (e.g., temperature of devices) for achieving reliable results. Modeling is often a valuable alternative to measurements, especially in cases where enough information about the software system and its context is known. Obviously, energy/performance models have to hold an appropriate level of abstraction that allows not to miss relevant aspects of the system and, at the same time, to keep an acceptable complexity of model evaluation process [47].

This chapter explores the combination of measurement-based experiments and modeling in the context of energy/performance analysis of software systems, in order to benefit from the advantages of both of them. In particular, we investigate how to exploit performance models (specifically, Layered Queuing Networks – LQNs) to reduce the experimentation time while keeping a high accuracy in the energy consumption estimate of a software system. Although LQNs are a well-known modeling notation for software performance analysis [90, 187], their adoption for energy consumption analysis has yet to be developed. LQNs fit our purposes as they represent system resources as time-consuming entities. As energy consumption is a time-based metric, it is possible to define a relationship between performance and energy consumption according to resource utilization (namely, the amount of time a resource is busy). We reduce the reality gap between the LQN model and the system under analysis by systematically refining the LQN model using data obtained from a small-scale measurement-based experiment. After achieving satisfactory accuracy, the LQN can be used to study the system under different workloads and get corresponding resource utilization estimates. These estimates can be multiplied by the energy consumed per second by each resource to obtain the total energy consumed while the resources are busy. We tested our approach on two different systems: Digital Camera, which we employ as a running example, and Train Ticket Booking System. The former is an image processing application that we deployed on an embedded platform, while the latter is a container-based web application for managing train bookings. For the Digital Camera, the supplied workloads correspond to batches of images of different resolutions, namely 2K, 4K, and 8K. Instead, for Train Ticket Booking System the workloads consists of bursts of 75, 150, 225, 300, 375, 450, and 500 customers. We parametrized the LQN with data measured for the batch of 2K images and the 75-customer burst, respectively, then we estimated resource utilization and energy consumption for different scenarios. Promising results emerged by comparing the measured data with the estimates. The Mean Absolute Percentage Error (MAPE) obtained for Train Ticket Booking System equals 9.24% for CPU Utilization and 8.47% for energy consumption. At the same time, we reduced experimentation time from 5 hours to 35 minutes.

The technique presented in this chapter, presumes the existence of a system prototype, i.e., a candidate HW/SW implementation of the system. The system prototype is profiled and modeled for performance and energy consumption estimation. As mentioned before, through a combination of empirical experimentation and MDE, we accelerate experimentation and energy consumption analysis. Consequently, this approach could be effective for Design Space Exploration (DSE) and can be integrated in our V&V-based HW/SW Co-Design methodology to optimize and compare different design solutions.

Hence, the main contributions of this chapter are: (i) an approach for using LQNs to make accurate energy estimations of a software system, (ii) a preliminary empirical evaluation of the proposed approach on two different software systems across different domains, (iii) a replication package for the independent verification and replication of the performed evaluation [1]. The chapter is structured as follows. Section 6.1 presents energy consumption basics and describes the Layered Queuing Networks. Section 6.2 delves into the approach and shows it through a running example: the Digital Camera. Section 6.3 outlines the experiments and the results achieved on the Train Ticket Booking System illustrative example. Threats to validity and related work are discussed, respectively, in Section 6.4 and Section 6.5. The chapter ends with conclusions in Section 6.6.

[1]https://doi.org/10.5281/zenodo.7877782

6.1 Background

6.1.1 Software Energy Measurement

The physical quantities used for expressing the energy consumed by software executions are electrical energy and electrical power. Electrical energy quantifies the amount of work needed to drive current through a circuit, while electrical power refers to the rate energy is consumed by the circuit at any instant. Energy is commonly measured in joule (J), which is defined in the International System of Units (SI). Power, unlike energy, expresses a rate. Indeed, power is defined as the total energy consumed over time measured in joule per second ($\frac{J}{s}$) or, following the standard SI, in watt (W).

Literature includes a plethora of tools for measuring software energy consumption [72]. We distinguish energy profilers and power monitors. Energy profilers are software tools which provide an estimation of the energy consumed by a running application. Compared with power monitors, energy profilers are easy to set up but are less accurate as they provide estimates. Among the most popular ones we have: `perf` and `powerstat`. Power Monitors are hardware devices wired directly to the system to profile, e.g., to the battery of the system. Therefore, they are more accurate but also more complex to set up. In this work, we exploit two power monitors for our experiments: the Monsoon [136] and the Watts up Pro? [193]. Instead of reporting total energy usage in joules, several power monitors report the power consumption in watts. They read, at each instant `t`, the current intensity (I) and voltage (V) and calculate the power as $P = I \times V$. The total energy consumption (E) can be derived by the power consumption. When the power consumption is constant, the total energy spent is proportional to the observation interval Δt, that is $E = P \times \Delta t$. As previously mentioned, some power monitors calculate power values querying the system every instant `t` over an observation period Δt. This process results in a dataset, where each row is formed by the power value in watts and the timestamp of the reading. This dataset describes the distribution of power values consumed during Δt. Since power corresponds to the rate at which energy is consumed over time, it is possible to retrieve the total amount of energy spent between two instants t_0 and t_n, by calculating the area bounded by these two instants underneath the distribution. Formally, this area can be calculated by integrating the power consumption values over t_0 and t_n.

6.1.2 Layered Queuing Network

Layered Queuing Networks (LQNs) are used to describe and analyze the performance of a system [198]. A LQN captures the behavior of a system as a set of interacting entities sending and servicing requests. Incoming requests generate workload that is handled by system resources such as CPU or Disk. If a request comes to a resource that is already busy, the request is queued. Such model of computation is peculiar of ordinary queuing models. LQNs extend ordinary queuing models by implementing simultaneous resource possession. Simultaneous resource possession occurs when a resource is blocked waiting for another to finish serving a request. Figure 6.3 shows the LQN used for analyzing a Train Ticket Booking System. The root task may be used to specify the workload which the system will undergo or receive requests from the environment. In the former case, the task is named reference task and represents the number of users in the system, while in the second case requests arrive following a rate specified with the λ parameter. System entities are shown as parallelograms and are called tasks. A task provides service through one or multiple entries and has a single host processor. Processors are represented with circles connected to a task and embody system resources. Thus, processors handle the workload generated by the entries. Entries are represented as rectangles within tasks and specify a demand corresponding to the mean time the processor is busy serving the entry. Communication among tasks is described by arrows connecting the entries. These arrows are labeled with the mean number of requests the client task sends to the server task. Carleton University provides a suite of tools including modeling languages and an analytic solver to create and retrieve performance metrics [58]

6.2 Our approach

As mentioned in the introduction, we exploit performance models to assist designers in making energy-related decisions. For this purpose, we have conceived a modeling process to reach a good trade-off between abstraction and accuracy of estimates. Developing models of existing systems may help to understand them better, which includes identifying flaws and finding opportunities for improvement. At design time, models are used to describe design choices or to verify conformance to the requirements. Considering that implementation details highly influence the energy consumption of the software, we chose to proceed bottom-up and exploit models for studying existing software systems. This implies the use of models that can be simulated or solved analytically to study "what if" cases and gain fast insights into the system under study. In addition, we see modeling as an opportunity to reduce the complexity and time of measurements. Our approach combines both strengths of measurements and abstraction.

Our modeling approach is schematically represented in Figure 6.1. Each rounded box represents an activity, while labels on the edges embody the exchanged artifacts. Initially, an existing system is modeled and profiled. Profiling means measuring the characteristics of the system that we planned to evaluate using the models, in this case performance and energy consumption. Modeling and profiling activities can be conducted in parallel. After the end of these two activities, the measured data are used to parametrize the model. Taking performance as an example, we can look at the rate at which requests arrive at the system, or how long it takes software components to handle a request. Parameters such as arrival rate, or service time are the input parameters of the model. Once the model is parametrized, it is validated and possibly refined. In the validation step, the correctness of the model and the accuracy of the estimations are considered. Moreover, during this phase, designers refine the model removing unnecessary details and simplifying it. The process stops when consensus about model accuracy and abstraction level is reached. We envisage that, under a set of assumptions, it is possible to exploit the flexibility of performance models for estimating energy consumption. Briefly speaking, since energy consumption is a metric based on time, we can relate energy consumption and resource utilization over a fixed observation time. Our approach, at the moment, considers only the cases in which *energy consumption grows linearly with execution time*. However, this is not always true. As reported by Cruz et al [72], some mobile architectures have fast but power-hungry CPUs for processing heavy tasks and slower but more efficient CPUs for less time-consuming tasks. Despite this, the approach has significant benefits since performance models can be utilized to scale workload and retrieve energy consumption values along with resource utilization estimates.

As a result of the modeling process depicted in Figure 6.1, we obtain a model approximating the behavior of the system with a certain degree of accuracy. This approximation stems from the

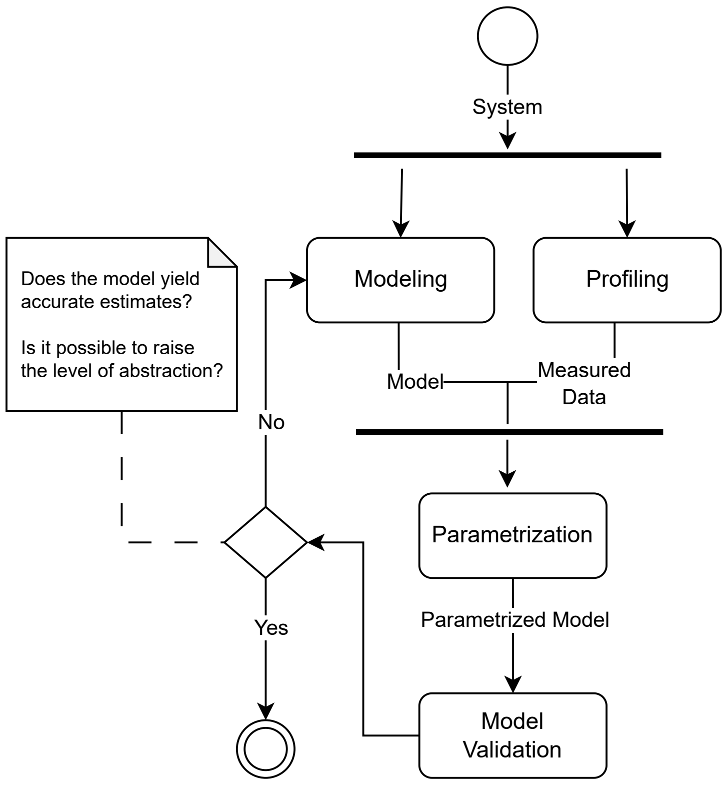


Figure 6.1: The process underlying our approach.

behavior observed while profiling the system. So, the model reproduces the behavior observed under the experimental setup of the profiling phase. For example, profiling the system under a given workload will produce a model representing resource usage under that specific workload. This aspect becomes even more important when it comes to power consumption. Indeed, behavior and power consumption are heavily connected. Different experimental settings will result in different system behaviors and thus different power distributions. On the other hand, we can exploit the relationship between behavior and power distribution to infer that the modeled behavior will induce a power distribution similar to the system one. Indeed, from the performance model solution we know when resources will be busy handling requests. For this reason, we can map the time interval resources are busy onto the power distribution measured during the profiling phase. Figure 6.2 shows an example of mapping between model behavior and the power distribution measured on the system. In this example, the power distribution is measured while running several times a software on a server. The blue cross on the x-axis represents the end of a single execution. Requests arrive periodically and require CPU and disk at the same time. The CPU, depicted by the gray area, is occupied for a longer interval than the disk, which is depicted in orange. White areas represent the time interval when resources are idle. Each colored segment underneath the power distribution represents the energy consumed by a resource during that time interval (see Section 6.1.1). Therefore, it is possible to obtain the energy consumption of a resource by integrating the areas where the resource was busy (6.1). Thus, given a resource, the sum of all the intervals in which a resource was busy (i.e., same color intervals in Figure 6.2) represents the total energy demand (i.e., $ED(res)$) during the observation time (Equation 6.2). Accordingly, we can derive the average consumption per visit of each resource, namely $E(res)$ (Equation 6.3). This value is tied to a visit,

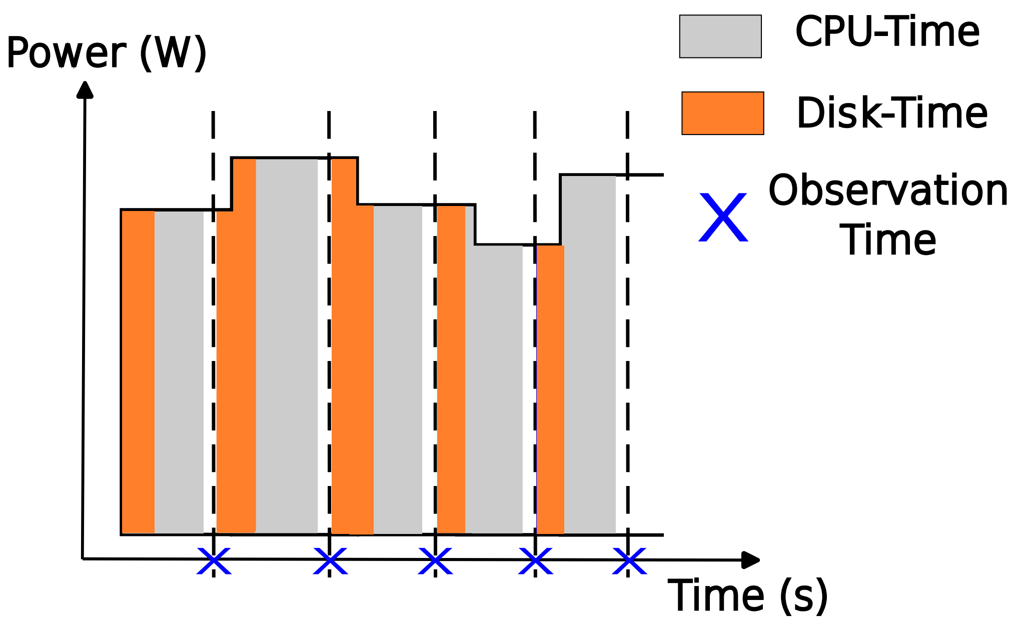


Figure 6.2: Sample Power Distribution highlighting CPU and Disk busy time, respectively, in gray and orange.

so it reflects the energy spent serving a specific software task. For example, if the CPU spent 2 seconds serving a visit, the energy spent per visit refers to the consumption during two seconds. So, the energy consumed per visit is measured in $\frac{Joule}{Visit}$. The Joule consumed per second (i.e., $e(res)$) can be obtained from $E(res)$, which is unbound to the size of a software task. We calculated $e(res)$ by dividing $E(res)$ by the average time the resource is busy performing a software task (Equation 6.4). Since $e(res)$ is decoupled from the size of the task visiting a resource, we can use it to estimate the energy consumed by a resource busy serving a task with whatever size. In other words, since $e(res)$ is measured in $\frac{Joule}{s}$, it is untied from the workload and becomes a property of the resource. Thus, we assume that it does not sensibly vary by scaling the workload. In this way, we can use $e(res)$ as a multiplier of the busy time of a resource and estimate the energy consumption in case of larger workload sizes. This method allows designers to scale up the estimations retrieved during low-effort experiments to predict the energy and performance of more complex scenarios. In this way, designers can very quickly obtain estimates for complex cases and thus avoid laborious and complex experiments.

$$E(res,i) = \int_{t0,i}^{S_{res}} P\,dt[\frac{Joule}{Visit}] \tag{6.1}$$

$$ED(res) = \sum_{i=1}^{\#Visit} \int_{t0,i}^{S_{res,i}} P\,dt \tag{6.2}$$

$$E(res) = \frac{ED(res)}{\#Visit} \tag{6.3}$$

$$e(res) = \frac{E(res)}{S(res)}[\frac{Joule}{s}] \tag{6.4}$$

where:

res	= a resource
$t0$	= the instant a resource starts to be busy
i	= ith visit to a resource
$\#Visit$	= total number of visits to a resource
S_{res}	= the average time the resource spends serving a software task

Running example: the Digital Camera

We deployed the application of a Digital Camera (DC) on a BeagleBone Black (BBB) development platform. The BBB is a ARM-based single core platform equipped with Linux Debian, which, in our setting, executes only the services of the operating system and the DC. As our approach envisions (see Section 6.2), we set an experiment in which the DC is subject to a synthetic workload. While the DC processes the workload, we measure the power required by the BBB using a Monsoon Power Monitor [136], which is placed between the BBB and a notebook. The notebook orchestrates the experiment, stores the power consumption recorded by the power monitor, and records the performance of the BBB. Performance measures include the time DC takes to process an image (i.e., the response time) and resources utilization. The notebook queries the operating system of the BBB and retrieves, for each execution, the busy time of the CPU.

The workload consists of a stream of image batches. A batch contains 30 pictures of the same format chosen between 2K, 4K, and 8K. A total of thirty batches are provided to the application, i.e., 10 per format. The DC processes all images sequentially in a batch, then pauses for two minutes before continuing. Their arrival is randomized to avoid any influence of image size on measurements [195]. The batch size is set to 30 to achieve statistical significance of metrics derived from observed behavior, such as resource utilization.

We determined the energy spent by the CPU to operate a batch of a particular format based on Equation 6.3. Therefore, using the measured response time, we calculated the e multiplier using Equation 6.4, which represents the average power in $\frac{Joule}{s}$ spent by the CPU. As Section 6.2 remarks, e is a property of the resource, which is detached from the characteristics of the visiting task and therefore has the same value across scaled workloads. If we know or estimate the $S(res)$ variable of Equation 6.4, we can use the e profiled for the batches of 2K images to discover the average energy spent for batches of format 4K and 8K. Consequently, we obtain estimates of the energy consumed for batches of 4K and 8K images without taking any measurements. We used an LQN to simulate the arrival of 30 images of 4K and 8K and estimate the response time for the corresponding batch. The LQN has a single task, representing the DC, connected to a single processor, which embodies the CPU. Incoming images arrive following a rate (i.e., λ). By varying the λ parameter, we replicated sequential arrival of 4K and 8K images, while changing the service time of the DC task, we set the average CPU processing time per image. In fact, the service time of the CPU differs based on image size. The model yields estimations concerning both the time it takes the CPU to handle a batch of 30 images and CPU usage.

Table 6.1 provides the results according to the image format of a batch. The table includes the arrival rate λ, the time spent handling a batch, i.e. the response time, CPU utilization, and energy consumed per batch. Columns containing two values show the measured value on the left and the corresponding estimates on the right. The estimates for the energy consumed per batch

Table 6.1: Results for the Digital Camera and Train Ticket Booking System according to the size of the input. Legend: IS: Input Size; λ: Arrival Rate; R: Response Time; U: CPU Utilization; e: Average Power Consumption; EC: Average Energy Consumed. Columns with double values indicate measured and estimated values (on the right).

IS	λ ($\frac{images}{s}$)	R (s)	U (%)	e ($\frac{J}{s}$)	EC (J)
		Digital Camera			
2K	0.48	60.30 - 60.30	96.30 - 96.48	1.57	95.27 - 95.16
4K	0.12	240.36 - 240.30	96.76 - 96.12	1.59	382.46 - 379.24
8K	0.03	960.73 - 960.60	97.39 - 96.06	1.59	1537.96 - 15016.04
		Train Ticket Booking System			
75	6.45	4.09 - 4.63	35.96 - 39.86	78.56	321.99 - 364.17
150	9.17	8.89 - 9.27	50.72 - 56.73	80.89	719.82 - 728.34
225	10.32	13.86 - 13.90	57.88 - 63.77	82.45	1143.19 - 1092.51
300	11.02	18.96 - 18.54	61.70 - 68.10	82.28	1560.54 - 1456.68
375	11.34	24.95 - 23.17	64.91 - 70.08	82.02	2047.20 - 1820.85
450	11.51	29.89 - 27.81	66.32 - 71.13	82.77	2474.40 - 2185.03
500	11.64	33.73 - 30.90	67.67 - 71.94	82.67	2788.76 - 2427.81

are calculated multiplying $1.57\frac{J}{s}$, which corresponds to the e calculated for batches of 2K images, by $240.30s$ and $960.60s$, which is the estimated response time for batches of 4K and 8K images. By comparing estimates to the measurements, it can be concluded that the results are promising. Finally, the BBB data sheet reports a range of $1.04\frac{J}{s}$ to $2.3\frac{J}{s}$ consumed by the platform when subject to various load [37]. The $e(CPU)$ multiplier of the DC falls within this range. This observation confirms the reliability of the $e(CPU)$ used for the analysis and the value of the approach for evaluating particular combinations of hardware and software. The small complexity of this running example and the availability of the DC source code, have simplified the mapping process between power distribution and the busy time of the CPU. Indeed, the DC executes only few functions in sequence, and from the source code we could see when the CPU operations were performed. In addition to the plethora of analyses that can be done by varying LQN parameters, we also gain benefits in terms of time. In fact, we obtained energy estimations without performing experiments in the 4K and 8K cases.

6.3 Evaluation

In light of the promising results obtained from the experiments with the Digital Camera, we decided to validate the method using a widely used illustrative example: Train Ticket Booking System (TTBS) [92], which is an application comprised of 68 Docker containers that manages bookings of a railway system.[2]

Consistently with the method described in Section 6.2, we set up a testbed for profiling TTBS and supplied a synthetic workload to the application. Therefore, we modeled TTBS through an LQN (see Fig. 6.3) and parametrized the model with the measurements retrieved during the fastest experiment. For the sake of space, we do not show the iterations, described in Section 6.2, performed

[2]For our measurements, we used release 0.0.4: `https://github.com/FudanSELab/train-ticket/tree/release-0.0.4`

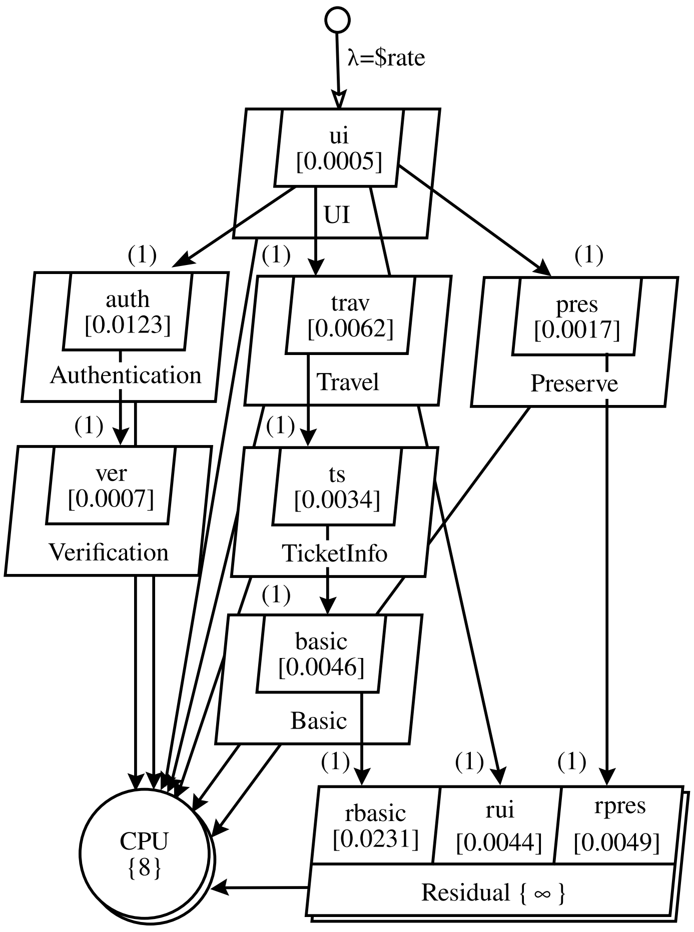
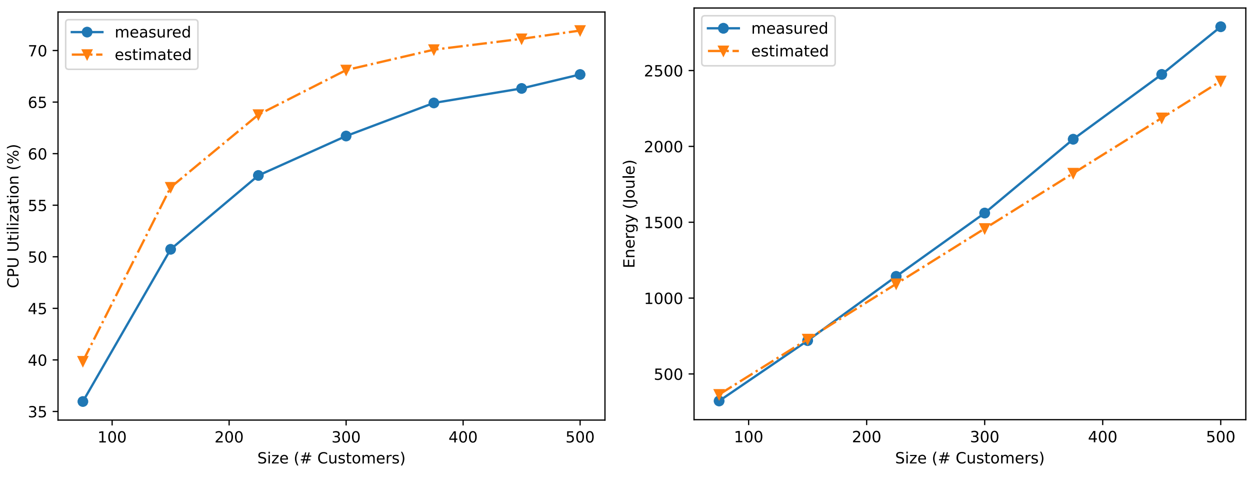


Figure 6.3: Layered Queuing Network of Train Ticket Booking System. The variable $\$rate$ varies based on the workload to provide to the model.



(a) Performance **(b)** Energy Consumption

Figure 6.4: Comparison between measured and estimated results.

to obtain the LQN. Each task in the model embodies a Docker container, which has its service time indicated in square brackets. We remark that this value stands for the average time a container kept the CPU busy. We included a task called `Residual` that models the time spent on the CPU by unrepresented containers. Thus, this task can be seen as a delay that is triggered as requests arrive.

The testbed consists of two machines, M1 and M2, used, respectively, to record the measurements and run TTBS. Both machines run Ubuntu Linux. M2 has 32 GB of RAM and 8 CPUs. It is worth to remark that using two different machines we reduce the perturbation on the machine where the application is running. Hence, we collected results as cleaner as possible. We provide TTBS with a variable-sized burst of customers and measure the performance and the energy expense of M2 for each burst. The bursts can have size equal to 75 (i.e., the one we used to parameterize the LQN model), 150, 225, 300, 375, 450, 500 and were randomly supplied to TTBS for 30 times. This means for each size we collected 30 readings for a total of 210 executions. Randomization is necessary to remove the burst size from the factors that might influence the readings [195]. Further, we inserted a one-minute pause between executions to allow M2 to cool and thus prevent subsequent executions from affecting the profiled data. We empirically validated the pause we need to have a fresh machine. Measurements are coordinated through a bash script running on M1. The script runs JMeter [25], which generates a burst of customers operating on TTBS. At the same time, it records application and M2 performance from the operating system of M2, while the power consumption of TTBS is recorded from a Wattsup Pro power monitor [193] connected to M2. Additionally, we measured disk utilization, but excluded it from the analysis because it was too low to be meaningful.

We set the arrival rate, i.e., the λ parameter of the LQN in Figure 6.3, and the service time of each task according to the data collected while supplying 75-customer bursts. Therefore, we incremented the λ parameter to replicate the arrival of 150, 225, 300, 375, 450, and 500 customers. The LQN returns CPU utilization predictions that we compared against the measurements. Table 6.1 reports performance and energy metrics for each batch size. It provides the arrival rate, i.e., the one we also supplied to the LQN, the response time, CPU utilization, the average power consumed by the CPU, i.e., the e multiplier, and the average energy consumed per batch. The columns containing two values show the measured value on the left and the corresponding predicted value on the right.

As expected, the CPU utilization rises according to the size of the burst. In our experiment, CPU utilization ranges from 35.96% when supplying a burst of 75 customers to 67.67% with a burst size of 500 customers. Figure 6.4a shows the distance between CPU utilization estimates and measurements varying burst size. The predictions slightly overestimate the measurements. This overestimation is quantified by the Root Mean Squared Error (RMSE) which equals 5.27%. Moreover, we obtained a Mean Absolute Percentage Error (MAPE) of 9.24%, which confirms the accuracy of the CPU utilization predictions. Besides utilization, the LQN returns response time estimates for each burst size. Therefore, it is possible to estimate the average energy consumed for a given burst by combining the corresponding response time prediction with the e multiplier calculated for the 75-customer burst, i.e., $\frac{E(CPU)}{S(CPU)} = \frac{321.99J}{4.098s} = 78.56\frac{J}{s}$. For example, given the response time estimation for a 500-customer burst, i.e., 30.90 seconds, we calculated the corresponding energy consumption by multiplying it by e using Equation 6.4. We obtained an estimation of 2427.81 J, which is lower than the measured energy consumption, i.e., 2788.76 J. Figure 6.4b summarizes energy consumption predictions for each tested burst size. The energy consumption estimates are quite accurate as also evidenced by the RMSE and the MAPE which are equal to 200.16 J and 8.72%, respectively. However, we can observe that as the burst size increases, the difference between the prediction and the measured value also increases. As it can be seen from Figure 6.4, the RMSE between the energy estimation and the measured one shows a divergence trend meaning that we can suppose having a larger RMSE as the size of the burst grows up. Whereas, the response time trend appears to be convergent. We suppose that this phenomenon may be due to the amount of data collected during the shortest experiment. In our case, this can be noticed by comparing the value of the e across burst sizes. There is a difference of approximately $4\frac{J}{s}$ between the e calculated with 75-customers burst data and the one measured for greater workloads, which is nearly $82\frac{J}{s}$. Finally, by exploiting the LQN, we gain savings in terms of experimentation time. We spent nearly 5 hours collecting all the data for different burst sizes. This period can be reduced by measuring the system undergoing bursts of 75 customers and exploiting the model for predicting energy and performance for greater workloads. By doing so, we would have spent only 35 minutes for experimentation versus 5 hours for measuring the 7 cases.

6.4 Threats To Validity

This discussion of threats to validity follows the classification made by Wohlin et el. [195]. The results of the study might be affected by *Conclusion Validity* threats due to the low significance of the sample collected during the lower-effort experiment. In fact, due to the short duration of this experiment, the sample collected may not be enough to accurately characterize either the LQN parameters (e.g., service time of containers) or the energy data (e.g., value of $e\frac{J}{s}$). This inaccuracy affects the estimates since we use this dataset to parameterize the LQN and derive energy consumption values in the case of heavier workloads. In both Digital Camera and Train Ticket Booking System, we consider only the load handled by the CPU and scaled workloads. The findings might not be confirmed in situations involving more resources and different types of workload. Therefore, they might not be generalizable and the work might be affected by *Construct Validity* threats. Finally, we do not consider a broad sample of hardware/software systems. For example, we do not examine battery-powered systems, which may have power-saving modes. These characteristics might impact the measurements of energy consumption and performance. As a result, the study might be affected by *External Validity* threats, making it difficult generalize the findings to all types of systems.

6.5 Related Work

To the best of our knowledge, this is the first study investigating and quantitatively evaluating how performance models (specifically, LQNs) can be exploited to make accurate energy estimations of software systems. Moreover, in our case, the models are used to support measurement-based experiments and reduce experimentation time, thus assessing situations that designers aimed to measure. Several papers in the literature use queuing models to define energy-aware behaviors. These works come from different domains, such as robotics [83], wireless sensor networks (WSNs) [94, 109, 201], or cloud computing [34]. Cerotti et al. [60] use a queuing model to improve the utilization of the servers in a data center. Indeed, depending on the workload, some servers may be subject to long periods of low utilization, which still generate significant energy consumption. The queuing model incorporates a controller which manages the incoming workload so that servers maximize their throughput and resource utilization. So, each request will require less energy to serve, reducing the total energy consumption. Marsan and Meo [17] apply queuing models to optimize the energy footprint of a university WLAN. The authors consider the areas covered by multiple access points (APs). Co-located APs can be turned on/off, depending on the capacity of the group of APs to provide service and the number of active users accessing the WLAN. This situation is modeled with a queuing system which outputs the number of APs of a group that should be active to handle a given workload. In some of the situations, the authors manage to save even more than half of the energy usually expended to power the WLAN.

6.6 Conclusions

In this chapter we have introduced a model-based approach for simplifying the energy consumption estimation of software systems. We have exploited the linear dependency of energy consumption and performance to extrapolate estimations of the former one in scenarios that would require high measurement times in practice. We tested the approach using a running example: Digital Camera, then validated the results on a more complex application, Train Ticket Booking System. The experimental results are quite promising, thus we plan to apply our approach to larger-size energy-critical software systems. Besides, we intend to examine the performance and energy consumption of resources other than the CPU, such as the disk and network. Although the approach has been implemented on top of LQN model, the whole process is independent of the modeling notation adopted for sake of performance analysis. Therefore, as further future work, we plan to consider different modeling notations that could be more suitable in specific application domains.

Part III

Conclusion

Chapter 7

Conclusion

HW/SW Co-Design has been introduced to improve the design loop of the traditional design flow of embedded systems. The main drawback of the traditional flow lies in its design loop, which envisages separate design and implementation of the hardware and the software of embedded systems. Due to the growing complexity of embedded systems over the years, the traditional design flow became intractable and time-consuming due to the increasing number of iterations of the design loop. The objective of HW/SW Co-Design is to minimize design time by simultaneously developing both the hardware and software of the system. This methodology exploits an abstract model, called the technology-independent model, which represents the desired behavior without providing any details about implementation. The model is used during DSE to evaluate different HW/SW implementations of the represented behavior.

This thesis investigated if improving the technology-independent model benefits the quality of the design resulting from DSE. We focused on integrating approaches and metrics for identifying eventual defects of the technology-independent model and anticipate validation of extra-functional attributes. These purposes are fulfilled by formal methods, which usually use formal languages to produce executable or verifiable models for verification and validation. Since formal models are based on a well-defined syntax and semantics, their execution aims at discovering logical inconsistencies and estimate non-functional attributes according to a mathematical framework. Nonetheless, formal methods have yet to gain widespread adoption because their effectiveness has not been fully proven and the learning curve required to use them is high. Therefore, this thesis studied: (G1) how to integrate formal verification and validation before DSE, (G2) which verification and validation analyses to perform using the technology-independent model, and (G3) how to facilitate the adoption of formal methods to practitioners.

Part I of the thesis describes the first steps of the V&V-based HW/SW Co-Design. As mentioned in the Introduction, this novel HW/SW Co-Design methodology introduces multiple steps of verification and validation at the beginning of the design flow (G1), which aims at improving a platform-agnostic model. A platform-agnostic model describes the logical structure and the behavior of system elements and does not include any information about possible systems implementation. The main goal of the first part of the methodology is to early detect modeling flaws and avoid their propagation in the later stages as well as improve the overall quality of the model by estimating its quality attributes. We have integrated multiple formal methods for functional correctness verification and validation of timing, performance, and reliability constraints (G2). V&V-based Co-Design uses a UML/MARTE-based platform-agnostic model, which is then converted through multiple M2M transformations into several formal models for verification and validation. It is possible to ease the knowledge transfer of formal methods to practitioners by defining a mapping between informal and formal models and use the insights gathered from V&V to improve the source model (G3). Part II focuses on Design Space Exploration (DSE), which operates on the model resulting from verification and validation. DSE includes metrics and methods to compare different HW/SW implementations and that can be used to implement what described in the source model. In Chapter 4 we build a ML-model providing accurate performance values of a given software on different microprocessor. This ML-model aids designers during DSE in the choice of a microprocessor. Instead, in Chapter 5 and 6 we focus on the energy consumption of the system under design. We provide, in Chapter 5, a metric for comparing different microprocessors according to the energy consumed while executing a given software, i.e., JC4S, while Chapter 6 outlines a model-driven approach for estimating the energy consumed by a system prototype. As the reader may have noticed, energy consumption is an extra-functional property that is examined only in Part II. To the best of our understanding, energy consumption can only be measured or estimated when there is a prototype of the system in place, i.e. a candidate system implementation of the HW/SW. In Chapter 6, we study the cases in which energy consumption is proportional to performance. A proven relationship between energy and performance could eliminate the need to have a prototype of the system, since there are already works and tools that estimate performance from abstract models.

7.1 Findings

Each chapter of the thesis addresses particular steps of the V&V-based HW/SW Co-Design.

7.1.1 Chapter 2: From UML/MARTE Specifications to Functional Verification & Timing Validation

Chapter 2 described Co-Specification, thus the creation of the technology-independent model, and Co-V&V, so functional verification and timing validation. We used UML to model the FIRGCD toy example [160] and UML/MARTE to plug time constraints into the model. FIRGCD consists of a set of components that react to the reception of messages, which are sent through channels. A message can be lost during the communication. The execution of FIRGCD is repeated according to a timer.

This chapter shows how the modeling framework, defined for Co-Specification, facilitates verification and validation of FIRGCD, as well as how the insights resulting from verification and validation help improve the source model. We incremented the behavioral model of the traditional HW/SW Co-Design methodology with information about the domain and the structure of the system through a class and a component diagram included in the application view. Activity diagrams detail the behavior of each system element in the application view and model the communication protocol used by system elements in the communication view. In the time view, instead, we used UML/MARTE elements to define delays and timers, as well as the Value Specification Language (VSL), included in UML/MARTE, to constrain the duration of execution traces. We provided a semantic mapping, which is not yet automatic, from the above-described view model to a network of timed automata. A set of clocks is added to an automaton to create a timed automaton, where the clocks increase as the automaton is executed. A set of communicating automata composes a network of timed automata. The latter is then processed by the UPPAAL model checker which uses a verifier to check system properties [74]. The UML/MARTE model structure defines the topology of the network, while each behavioral description produces an automaton. In our case, time constraints impose a duration to the execution of a behavior or define a delay. UML/MARTE delays are converted into delays on automaton locations, while constraints on traces duration are not converted but verified using UPPAAL formulas.

Computational Tree Logic (CTL) is used to write UPPAAL formulas for verifying execution traces time constraints, reachability of locations, and communication correctness. We discovered that FIRGCD execution success and satisfiability of time constraints are largely determined by communication protocol and the duration of the timer used to time the execution. The system as modeled can lead to continuous loss of messages and thus failure of an execution. In addition, we found that the parameters set in the UML/MARTE model led to the unsatisfiability of a time constraint set on system-level behavior.

7.1.2 Chapter 3: Many-objective optimization of non-functional attributes based on refactoring of software models

Chapter 3 presented a potential approach for Co-Analysis, aiming to enhance the extra-functional attributes of a model through a process of iterative refactoring and evaluation. The main contributions of Chapter 3 consists in: an approach that uses refactoring in combination with NSGA-II for model optimization and techniques to automatically assess eventual improvements made to a UML model after a refactoring action.

The approach requires an initial model and a set of refactoring actions, which are randomly applied to the source model to generate several model alternatives. Each model alternative is evaluated according to performance, reliability, architectural distance to the initial model, and number of performance antipatterns. This loop is integrated into a many-objective evolutionary algorithm, i.e., NSGA-II, that chooses the solutions with better performance and reliability but presenting less changes and performance antipatterns. This process is repeated for a set number of iterations, where each iteration optimizes the best solutions resulting from the previous iteration. Models are created with UML and supplemented with MARTE and DAM profiles to represent performance and reliability concepts. For model evaluation, we defined an automatic conversion from MARTE to LQNs for performance evaluation and from DAM to a closed-form model for reliability assessment. Performance antipatterns, instead, are identified quantifying the probability of their occurrence and tallied if this probability exceeds a fixed threshold. Finally, the architectural distance is determined by multiplying the weight of the model element - identified by its number of links to other model elements - with the effort required to apply a given refactoring action. The approach is tested through two illustrative examples: Train Ticket Booking System and CoCoME, which, for each case, resulted in a set of solutions improving the initial model.

The experimentation showed, in both illustrative examples, that the majority of the solutions resulting from our approach exhibit notable improvements in both performance and reliability *(RQ2)*. Moreover, we studied how the optimization process is influenced by experimental parameters and how the quality of the solutions is impacted by the objectives *(RQ1)*. Additional experiments were conducted to explore how performance antipatterns *(RQ1.1)* and their fuzzy detection technique *(RQ1.2)* impact finding better solutions, as well as the extent to which architectural distance contributes to the quality of solutions *(RQ1.3)*.

We performed two experiments to determine the effect of integrating performance antipatterns into the optimization process on the quality of solutions. One experiment involved their inclusion, while in the other we excluded them. Instead, to evaluate fuzzy detection, we compared the solutions of an experiment using fuzzy detection with those resulting from an experiment using fixed thresholds for performance antipatterns detection. Including performance antipatterns in the optimization process increases, for both illustrative examples, the number of solutions having improved performance and reliability. The same doesn't apply to the fuzzy detection of performance antipatterns, which adoption didn't result in improved solutions. Instead, we assessed the effect of architectural distance by examining a scenario where a refactoring action requires zero effort. We obtained better solutions when accounting refactoring actions effort in the optimization process. Finally, by examining the output of the experimentation, we identified a subset of refactoring actions that have a higher likelihood of producing improved solutions *(RQ3)*.

7.1.3 Chapter 4: Statement-Level Timing Estimation for Embedded System Design Using Machine Learning Techniques

In Chapter 4, we examined how to use machine learning (ML) models to predict the execution time of C-based functions on a set of processors. The provided approach applies in contexts where designers wish to compare various HW/SW implementations, making the method useful to quickly estimate the performance of a system prototype. The main reason to involve ML-based techniques for performance prediction is to check if they lead to improved estimation accuracy and reduced estimation time. The main contributions include an assessment of prediction accuracy and time of five ML models, as well as a framework to automatically generate the dataset to train these models.

The aim of this approach is to produce prediction models for the execution time of C code on a specific processor, which can be alternative of simulators or emulators. In this study, we compared 5 ML models, namely Linear Regression (LR), Fine Trees (FT), Boosted Trees (BOT), Bagged Trees (BAT), and Support Vector Machine (SVM), according to prediction time and accuracy. In order to be compared, we trained these models with data retrieved from the execution of a benchmark, written using the C language, on a set of target processors. Similarly to what described in Chapter 5, we generated a set of inputs for each function included in a benchmark of 15 well-known functions written using the C language. For each input within the corresponding input set, we simulated the execution of each function in the benchmark on multiple processors. We considered, as target processors, the Cobham Gaisler LEON3, the Intel 8051, and the ATmega328/P. The simulations generate the dataset used for training the ML models. The evaluation of the models involved three phases: feature extraction, which reduces the number of features used for training to the most influential ones, followed by a test/train and validation phases. The step train/test is used to preliminary assess the precision of the predictions on a subset of models and processor, while the validation considers the full set of models and processors. The validation indicates that regression tree models, such as Fine Trees, Bagged Trees, and Boosted Trees, result in the highest prediction accuracy and time efficiency. However, these models only yield satisfactory results for the 8051 processor. Indeed, the Root Mean Square Percentage Error (RMSPE) obtained from the results retrieved for the 8051 processor remains under the 10%. The RMSPE for the LEON3 ranges between

6% and 70%, while for the Atmega328/P only Fine Trees produces a RMSPE under 60%. Finally, out of all the models, Bagged Trees require the shortest amount of time to generate predictions. Based on the results, we concluded that the study requires additional investigations and refinements for improving the accuracy of the predictions.

7.1.4 Chapter 5: An Early-Stage Statement-Level Metric for Energy Characterization of Embedded Processors

Chapter 5 outlined J4CS, a metric that quantifies the energy consumed by a processor in relation to the executed C statements. J4CS is defined as $\bar{E}'(p_j) \cdot \overline{I}'(p_j, z_i, b_{i,k})$, which is the average energy consumed by a processor p_j multiplied by the average assembly instructions spent by p_j to execute a C code z_i with a fixed input $b_{i,k}$. J4CS is suitable for assessing the energy consumption of diverse C-based software solutions on a target processor. Chapter 5 provides the definition of J4CS plus its evaluation, which studies the accuracy of the metric across different processors using a benchmark. Moreover, we provided the J4CS evaluation framework, which automatically generates the quantities needed to calculate J4CS (e.g., number of executed assembly instructions and c statements) and the metric itself. The chapter ends presenting J4CS-A, which is a refinement of J4CS aiming at improving the accuracy of J4CS.

We validated J4CS through an experiment involving two target processors: Cobham Gaisler Leon3 and Intel 8051. We first defined a benchmark including 15 well-known functions (e.g., Quick-Sort, Binary Search). The mentioned framework creates a range of inputs for each function by varying input values and data types (such as int8, int16, and float). Every function and its associated set of inputs are executed on both simulators and actual processors, producing a dataset of J4CS values measured and estimated. The framework orchestrates input generation and the execution of the benchmark on the simulators. We used three boards to obtain empirical data of J4CS: LEON3FT-RTAX and LEON3FT-UT699, having the Leon3 processor, and the 8051-based AT89C51 development board. We executed the benchmark on the boards and compared the measurements against the estimations. We obtained, in all the cases, highly variable relative and absolute error ranges, which suggests further validation or a refinement of the metric. To reduce the error, we introduced J4CS-A, which integrates the pre-existing concept of affinity into the J4CS formula. The affinity metric indicates the most appropriate processor class for executing a given function and ranges between 0 and 1. This quantity embodies the affinity between architectural and functional features of the target processor and the executed software functionality. We used J4CS-A to decrease the error margin of measurements retrieved from the AT89C51 board. The reduction in relative and absolute errors is promising for all datatypes. Indeed, we obtained an error reduction, for both relative and absolute error, in the range of {50%...110%}.

7.1.5 Chapter 6: An approach using performance models for supporting energy analysis of software systems

In chapter 6 we provided a method to estimate energy consumption and performance of a given system prototype, thus a candidate HW/SW implementation, and reduce the time spent testing the system. This chapter contributes to the methods usable during DSE to evaluate a candidate system implementation according to energy consumption and performance. The method utilizes an experiment to generate a LQN. The experiment should take as little time as possible to set up and monitor the system, which can be achieved supplying to the system, for example, a low-utilization workload. The LQN replicates the behavior of the system during the experiment. After checking the accuracy of LQN estimates, performance can be estimated in different scenarios by changing LQN parameters. Moreover, we provided a relationship between performance and energy consumption to obtain the corresponding energy consumed by the scenarios evaluated with the LQN. We first tested the method through a running example, namely Digital Camera, and then validated it using the Train Ticket Booking System illustrative example. Digital Camera involves an application that compresses and stores images, while Train Ticket Booking System is a train bookings management system. We focused on cases where designers aim to increase the workload handled by the system. Therefore, for validation purposes, we observed the system undergoing a set of scaled-up workloads and predicted performance and energy in the same cases. Finally, we compared the measurements against the predictions. For Digital Camera, we measured the system subject to a stream of 2K images and evaluated the cases where a stream of 4K and 8K images is supplied. Instead, for Train Ticket Booking System, we observed the system handling a burst of 75 customers and evaluated the cases of 150, 225, 300, 375, 450, and 500 customers burst. We evaluated, for both cases, CPU utilization and energy consumption.

The results for Digital Camera are very accurate. For example, we get an estimate of 379.24 J in the case of an 4K image stream versus a measured value of 382.46 J. Similarly, this applies to an 8K image stream where the estimated value is 15016.04 J and the measured value is 1537.96 J. Train Ticket Booking System confirms promising results. Indeed, we reduced measurement time from 5 hours to 35 minutes by exploiting our model, this reflected in a Mean Absolute Percentage Error of 9.24% in the estimates of CPU utilization and 8.72% in energy consumption predictions. We noticed, in both cases, that as the burst size grows larger, the gap between the estimation and the measured value also expands. One possible reason for this is that the minimal experiment data used to parameterize the LQN may not accurately reflect system behavior.

7.2 Publications

This section lists the publications presenting the contributions of this thesis. The list is divided in four types: journal (JN), conference (CN), workshop (WS), and doctoral symposium (DS) papers. Chapter 1 describes the V&V-based HW/SW Co-Design which is the main contribution of the thesis and has been discussed at the Doctoral Symposium of MODELS 2021 (DS1) and at the WOSP-C workshop (WS1). The second work discusses the contributions found in Chapter 2. Thus, the paper introduces the methodology and presents Co-Specification and Co-V&V. and that we improved with the findings described in the journal paper JN2. Finally, the contributions presented in Chapter 5, 6, and 4 are published in JN1, WS2, and CN2, respectively. The papers indicated in this section are the result of several collaborations. For the studies described in JN1 e CN2, I implemented the microbenchmarks and the framework that generates the dataset employed for ML model training and evaluate J4CS. Additionally, I wrote the sections regarding the microbenchmarks and the framework, as well as reviewed the paper. I contributed to JN2 and CN1 by implementing the transformation from the UML/MARTE models and part of the EASIER framework. Moreover, I created the model of Train Ticket Booking System and analyzed the data for RQ3 in JN2. Finally, I wrote the section dedicated to RQ3 in JN2 and edited and revised the papers. As the main author of WS1, WS2 and DS1, I contributed to the conceptualization of the studies, the creation of the models, the design and execution of the experiments, and the data analysis of the results.

JN1. V. Muttillo, P. Giammatteo, V. Stoico, and L. Pomante. An early-stage statement-level metric for energy characterization of embedded processors. *Microprocessors and Microsystems*, 77:103200, 2020 [140].

JN2. V. Cortellessa, D. Di Pompeo, V. Stoico, and M. Tucci. Many-objective optimization of non-functional attributes based on refactoring of software models. *Information and Software Technology*, 157:107159, 2023 [66].

CN1. V. Cortellessa, D. Di Pompeo, V. Stoico, and M. Tucci. On the impact of performance antipatterns in multi-objective software model refactoring optimization. In M. T. Baldassarre, G. Scanniello, and A. Skavhaug, editors, *47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2021, Palermo, Italy, September 1-3, 2021*, pages 224–233. IEEE, 2021 [65]

CN2. V. Muttillo, P. Giammatteo, and V. Stoico. Statement-level timing estimation for embedded system design using machine learning techniques. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '21, page 257–264, New York, NY, USA, 2021. Association for Computing Machinery [139]

WS1. V. Cortellessa, L. Pomante, and V. Stoico. From uml/marte specifications to esl hw/sw co-design: Early functional verification and timing validation. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23 Companion, page 373–380, New York, NY, USA, 2023. Association for Computing Machinery [70]

WS2. V. Stoico, V. Cortellessa, I. Malavolta, D. Di Pompeo, L. Pomante, and P. Lago. An approach using performance models for supporting energy analysis of software systems. 19th European Performance Engineering Workshop (EPEW 2023).

DS1. V. Stoico. A model-driven approach for early verification and validation of embedded systems. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion, Fukuoka, Japan, October 10-15, 2021*, pages 684–688. IEEE, 2021 [179]

7.3 Future Work

This thesis defines a new design methodology called V&V-based HW/SW Co-Design. This methodology converts a source non-formal model to several formal models to perform rigorous preliminary analyses before the implementation. Literature certifies that using models can reduce the complexity of designing modern embedded systems. However, as the introduction extensively discusses, there are three problems stemming from using models. They are as follows:

1. Design models can include flaws. Therefore, we introduced formal methods in the early stages of the provided methodology to discover and fix eventual flaws (G1)

2. The analysis of functional and non-functional attributes of the system can be anticipated at the early stages of the design. We focus on how and which type of analysis can be performed by setting the right level of abstraction (G2)
3. Formal methods hamper the usability of a design methodology due to their high learning curve. We used multiple model-to-model transformations to ease the adoption of formal methods for practitioners (G3)

We believe that this thesis serves as a foundation for the development and enhancement of HW/SW Co-Design methodologies that includes formal analyses early in the design flow. This thesis explores the advantages of such a methodology. Nevertheless, in our view, a full-fledged validation of the V&V-based HW/SW Co-Design methodology is necessary in the future. Moreover, this thesis provides opportunities for further research in the embedded systems domain, highlighting three main points: the (i) definition of the right level of abstraction of models used in the initial stages, the (ii) proof of the efficacy of formal analysis, and (iii) how to facilitate knowledge transfer of formal methods to practitioners. The level of abstraction in our case has been established by HW/SW Co-Design, which forms the basis of our approach. As a result, we employ platform-specific models; future studies may explore whether platform-specific models can be further refined or abstracted to carry out the same or multiple analyses. Point (ii) stresses the necessity of proving that the analyses done in the early stages using formal methods apply in reality. For example, if any flaw fixed in the early stages does not appear during system implementation. We tackled point (iii) by implementing several model-to-model transformations from a non-formal model to several formal models. These transformations facilitate the creation and adoption of formal methods by practitioners. To date, however, this claim has not been backed up by empirical research, such as a user study. We plan to focus on two points that are missing in the dissertation: the methodology should be entirely executed by performing all the steps in sequence. In this way, we could observe whether the improvements of the technology-independent model resulted in a better design solution after DSE. The second aspect concerns the application of the methodology in an industrial setting. Through a field experiment, we can verify if the introduced steps of V&V increase the complexity of using HW/SW Co-Design for practitioners and evaluate whether employing M2M transformations eases the adoption of formal methods. Moreover, we want to supply the improved technology-independent model, i.e., the one obtained before our DSE, to existing HW/SW Co-Design methodologies that already implement an automatic or semi-automatic DSE [141]. Additional investigations are necessary for each step of the methodology. Co-Specification and Co-V&V are tested with a running example: FIRGCD. This choice influenced the subset of UML/MARTE, which addresses component-based architectures and reactive behaviors. Further validation of Co-Specification and Co-V&V is required through more complex case studies, resulting in an incremented subset of UML/MARTE, which could be suitable for modeling different types of system. Further investigation is required for Co-Analysis to determine the impact of experimental parameters on the optimization process and to make a comparison between various optimization techniques. Instead, for the contributions presented in Chapters 5 and 4, i.e., those regarding J4CS and performance estimation via machine learning, we obtain wide ranges of error percentage values that suggest further refinements. Finally, for the study in Chapter 6, we plan to estimate the utilization of other resources besides the CPU, for example, the disk.

Bibliography

[1] *AT89C51 ATMEL Development Board*, 2018 (accessed: 21.10.2019). https://www.indiamart.com/proddetail/at89c51-atmel-development-board-15939053291.html.

[2] *CC4CS benchmark*, 2018 (accessed: 21.10.2019). https://github.com/vnzstc/cc4cs.

[3] *GCov Profiler*, 2018 (accessed: 21.10.2019). https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[4] *LEON Bare-C Cross Compilation System (BCC)*, 2018 (accessed: 21.10.2019). https://www.gaisler.com/index.php/products/operating-systems/bcc.

[5] *LEON3-FT SPARC V8 Processor LEON3FT-RTAX*, 2018 (accessed: 21.10.2019). https://www.gaisler.com/doc/leon3ft-rtax-ag.pdf.

[6] *LEON3 processor*, 2018 (accessed: 21.10.2019). https://www.gaisler.com/.

[7] *SDCC - Small Device C Compiler*, 2018 (accessed: 21.10.2019). http://sdcc.sourceforge.net/.

[8] *Synthesizable VHDL Model of 8051*, 2018 (accessed: 21.10.2019). http://newit.gsu.by/resources/CPUs/i8051/VHDL/Synthesizeable%20VHDL%20Model%20of%208051.htm.

[9] *TSIM2 ERC32/LEON simulator*, 2018 (accessed: 21.10.2019). https://www.gaisler.com/.

[10] *UT699 32-bit Fault-Tolerant SPARCTM V8/LEON 3FT Processor*, 2018 (accessed: 21.10.2019). https://www.cobhamaes.com/pagesproduct/datasheets/leon/UT699LEON3FTDatasheet.pdf.

[11] Cc4cs-ml open-source git repository, 2020 (accessed: 15.03.2020). https://github.com/vnzstc/cc4cs.

[12] *Dalton Project: 8051 microcontroller, University of California*, 2020 (accessed: 15.03.2020). http://www.ann.ece.ufl.edu/i8051/.

[13] *Frama-C Software Analyzers*, 2020 (accessed: 15.03.2020). https://frama-c.com/.

[14] *Regression Learner App*, 2020 (accessed: 15.03.2020). https://it.mathworks.com/help/stats/regression-learner-app.html.

[15] *eSSYN Software Synthesis Tool*, 2023 (accessed: 21.05.2023). https://essyn.unican.es/.

[16] A. B. Abril Garcia, J. Gobert, T. Dombek, H. Mehrez, and F. Petrot. Cycle-accurate energy estimation in system level descriptions of embedded systems. In *9th International Conference on Electronics, Circuits and Systems*, volume 2, pages 549–552 vol.2, Sep. 2002.

[17] M. Ajmone Marsan and M. Meo. Queueing systems to study the energy consumption of a campus WLAN. *Computer Networks*, 66:82–93, June 2014.

[18] A. Aleti, S. Björnander, L. Grunske, and I. Meedeniya. Archeopterix: An extendable tool for architecture optimization of AADL models. In *ICSE 2009 Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES 2009, May 16, 2009, Vancouver, Canada*, pages 61–71. IEEE Computer Society, 2009.

[19] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Trans. Software Eng.*, 39(5):658–683, 2013.

[20] S. Ali, P. Arcaini, D. Pradhan, S. A. Safdar, and T. Yue. Quality indicators in search-based software engineering: An empirical evaluation. *ACM Trans. Softw. Eng. Methodol.*, 29(2):10:1–10:29, 2020.

[21] T. Altamimi and D. C. Petriu. Incremental change propagation from UML software models to LQN performance models. In M. Mindel, K. A. Lyons, and J. Wigglesworth, editors, *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering, CASCON 2017, Markham, Ontario, Canada, November 6-8, 2017*, pages 120–131. IBM / ACM, 2017.

[22] T. Altamimi, M. H. Zargari, and D. C. Petriu. Performance analysis roundtrip: automatic generation of performance models and results feedback using cross-model trace links. In M. Mindel, B. Jones, H. A. Müller, and V. Onut, editors, *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, CASCON 2016, Toronto, Ontario, Canada, October 31 - November 2, 2016*, pages 208–217. IBM / ACM, 2016.

[23] P. Altenbernd, J. Gustafsson, B. Lisper, and F. Stappert. Early execution time-estimation through automatically generated timing models. *Real-Time Syst.*, 52(6):731–760, Nov. 2016.

[24] D. Ameller, X. Franch, C. Gómez, S. Martínez-Fernández, J. Araújo, S. Biffl, J. Cabot, V. Cortellessa, D. M. Fernández, A. Moreira, H. Muccini, A. Vallecillo, M. Wimmer, V. Amaral, W. Böhm, H. Brunelière, L. Burgueño, M. Goulão, S. Teufl, and L. Berardinelli. Dealing with non-functional requirements in model-driven development: A survey. *IEEE Trans. Software Eng.*, 47(4):818–835, 2021.

[25] Apache Software Foundation. Apache JMeter. `https://jmeter.apache.org`. Accessed: 2023-04-12.

[26] D. Arcelli, V. Cortellessa, M. D'Emidio, and D. Di Pompeo. EASIER: an evolutionary approach for multi-objective software architecture refactoring. In *IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018*, pages 105–114. IEEE Computer Society, 2018.

[27] D. Arcelli, V. Cortellessa, and D. Di Pompeo. A metamodel for the specification and verification of model refactoring actions. In A. Ouni, M. Kessentini, and M. Ó. Cinnéide, editors, *Proceedings of the 2nd International Workshop on Refactoring, IWoR@ASE 2018, Montpellier, France, September 4, 2018*, pages 14–21. IWoR@ACM, 2018.

[28] D. Arcelli, V. Cortellessa, and D. Di Pompeo. Performance-driven software model refactoring. *Inf. Softw. Technol.*, 95:366–397, 2018.

[29] D. Arcelli, V. Cortellessa, and D. D. Pompeo. Automating performance antipattern detection and software refactoring in UML models. In X. Wang, D. Lo, and E. Shihab, editors, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 639–643. IEEE, 2019.

[30] D. Arcelli, V. Cortellessa, and C. Trubiani. Performance-based software model refactoring in fuzzy contexts. In A. Egyed and I. Schaefer, editors, *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9033 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2015.

[31] A. Arcuri and G. Fraser. On parameter tuning in search based software engineering. In M. B. Cohen and M. Ó. Cinnéide, editors, *Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings*, volume 6956 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2011.

[32] A. Arcuri and G. Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empir. Softw. Eng.*, 18(3):594–623, 2013.

[33] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, and W. Yi. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, 13(4), Mar. 2014.

[34] F. Balde, H. Elbiaze, and B. Gueye. Greenpod: Leveraging queuing networks for reducing energy consumption in data centers. In *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 1–8, 2018.

[35] J. R. Bammi, W. Kruijtzer, L. Lavagno, E. Harcourt, and M. T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign. CODES 2000*, CODES '00, page 82–86, New York, NY, USA, 2000. Association for Computing Machinery.

[36] G. Bavota, M. D. Penta, and R. Oliveto. Search based software maintenance: Methods and tools. In T. Mens, A. Serebrenik, and A. Cleve, editors, *Evolving Software Systems*, pages 103–137. Springer, 2014.

[37] BeagleBoard.org Foundation. The BeagleBone Black Development Platform. `https://beagleboard.org/black`. Accessed: 2022-11-11.

[38] M. Becker, R. Metta, R. Venkatesh, and S. Chakraborty. Scalable and precise estimation and debugging of the worst-case execution time for analysis-friendly processors. *International Journal on Software Tools for Technology Transfer*, 02 2018.

[39] S. Becker, H. Koziolek, and R. H. Reussner. The palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82(1):3–22, 2009.

[40] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*. Springer, Berlin, Heidelberg, 2004.

[41] L. Belkhir and A. Elmeligi. Assessing ICT global emissions footprint: Trends to 2040 & recommendations. *Journal of Cleaner Production*, 177:448–463, 2018.

[42] L. Berardinelli, A. Di Marco, S. Pace, L. Pomante, and W. Tiberti. Energy consumption analysis and design of energy-aware wsn agents in fuml. In G. Taentzer and F. Bordeleau, editors, *Modelling Foundations and Applications*, pages 1–17, Cham, 2015. Springer International Publishing.

[43] S. Bernardi, J. Merseguer, and D. C. Petriu. A dependability profile within MARTE. *Softw. Syst. Model.*, 10(3):313–336, 2011.

[44] H. Blume, D. Becker, L. Rotenberg, M. Botteck, J. Brakensiek, and T. Noll. Hybrid functional- and instruction-level power modeling for embedded and heterogeneous processor architectures. *Journal of Systems Architecture*, 53:689–702, 10 2007.

[45] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece. *Software cost estimation with COCOMO II*. Prentice Hall Press, 2009.

[46] L. Bogdanov. Look-up table-based microprocessor energy model. In *Fifth International Scientific Conference "Engineering, Technologies and Systems" (TECHSYS 2016)*, pages 180–185, 05 2016.

[47] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Software Engineering. Springer International Publishing, 2017.

[48] C. Brandolese, S. Corbetta, and W. Fornaciari. Software energy estimation based on statistical characterization of intermediate compilation code. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 333–338, Aug 2011.

[49] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, and D. Sciuto. A multi-level strategy for software power estimation. In *Proceedings 13th International Symposium on System Synthesis*, pages 187–192, Sep. 2000.

[50] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, and D. Sciuto. Affinity-driven system design exploration for heterogeneous multiprocessor soc. *IEEE Transactions on Computers*, 55(5):508–519, May 2006.

[51] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, and D. Sciuto. Affinity-driven system design exploration for heterogeneous multiprocessor soc. *IEEE Transactions on Computers*, 55(5):508–519, May 2006.

[52] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-level execution time estimation of c programs. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign (CODES '01)*, page 98–103, New York, NY, USA, 2001. ACM.

[53] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-level execution time estimation of c programs. In *Ninth International Symposium on Hardware/Software Codesign. CODES 2001 (IEEE Cat. No.01TH8571)*, pages 98–103, April 2001.

[54] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Timing and energy estimation of c programs. *ACM Transactions on Embedded Computing Systems (TECS), Special issue on Power Aware Embedded Computing*, 2002.

[55] C. Brandolese, F. Salice, W. Fornaciari, and D. Sciuto. Static power modeling of 32-bit microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(11):1306–1316, Nov 2002.

[56] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth and Brooks, Monterey, CA, 1984.

[57] V. Cardellini, E. Casalicchio, V. Grassi, F. L. Presti, and R. Mirandola. Qos-driven runtime adaptation of service oriented architectures. In H. van Vliet and V. Issarny, editors, *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 131–140. ACM, 2009.

[58] Carleton University Software Performance Research Group. Layered queuing network solver. `https://github.com/layeredqueuing`. Accessed: 2023-03-23.

[59] J. Castillo, H. Posadas, E. Villar, M. Martínez, and C. R. Darwin. Energy consumption estimation technique in embedded processors with stable power consumption based on source-code operator energy figures. In *XXII Conference on Design of Circuits and Integrated Systems, DCIS'07*, page 1, 2007.

[60] D. Cerotti, M. Gribaudo, P. Piazzolla, R. Pinciroli, and G. Serazzi. Multi-class queuing networks models for energy optimization. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '14, page 98–105, Brussels, BEL, 2014. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[61] B. Chen, X. Li, and X. Zhou. Model checking of MARTE/CCSL time behaviors using timed I/O automata. *Journal of Systems Architecture*, 88:120–125, 2018.

[62] J. Choi, E. Jee, and D.-H. Bae. Timing consistency checking for UML/MARTE behavioral models. *Software Quality Journal*, 24(3):835–876, 2016.

[63] V. Cortellessa and D. Di Pompeo. Analyzing the sensitivity of multi-objective software architecture refactoring to configuration characteristics. *Inf. Softw. Technol.*, 135:106568, 2021.

[64] V. Cortellessa, D. Di Pompeo, R. Eramo, and M. Tucci. A model-driven approach for continuous performance engineering in microservice-based systems. *J. Syst. Softw.*, 183:111084, 2022.

[65] V. Cortellessa, D. Di Pompeo, V. Stoico, and M. Tucci. On the impact of performance antipatterns in multi-objective software model refactoring optimization. In M. T. Baldassarre, G. Scanniello, and A. Skavhaug, editors, *47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2021, Palermo, Italy, September 1-3, 2021*, pages 224–233. IEEE, 2021.

[66] V. Cortellessa, D. Di Pompeo, V. Stoico, and M. Tucci. Many-objective optimization of non-functional attributes based on refactoring of software models. *Information and Software Technology*, 157:107159, 2023.

[67] V. Cortellessa, R. Eramo, and M. Tucci. From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement. *Inf. Softw. Technol.*, 127:106362, 2020.

[68] V. Cortellessa and V. Grassi. A modeling approach to analyze the impact of error propagation on reliability of component-based systems. In H. W. Schmidt, I. Crnkovic, G. T. Heineman, and J. A. Stafford, editors, *Component-Based Software Engineering, 10th International Symposium, CBSE 2007, Medford, MA, USA, July 9-11, 2007, Proceedings*, volume 4608 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2007.

[69] V. Cortellessa, A. D. Marco, and C. Trubiani. An approach for modeling and detecting software performance antipatterns based on first-order logics. *Softw. Syst. Model.*, 13(1):391–432, 2014.

[70] V. Cortellessa, L. Pomante, and V. Stoico. From uml/marte specifications to esl hw/sw co-design: Early functional verification and timing validation. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23 Companion, page 373–380, New York, NY, USA, 2023. Association for Computing Machinery.

[71] V. Cortellessa, H. Singh, and B. Cukic. Early reliability assessment of UML based software models. In *Third International Workshop on Software and Performance, WOSP@ISSTA 2002, July 24-26, 2002, Rome, Italy*, pages 302–309. ACM, 2002.

[72] L. Cruz and R. Abreu. Performance-based guidelines for energy efficient mobile applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 46–57, 2017.

[73] G. D'Andrea, T. Di Mascio, and G. Valente. Self-adaptive loop for cpss: Is the dynamic partial reconfiguration profitable? In *2019 8th Mediterranean Conference on Embedded Computing, MECO 2019 - Proceedings*, 2019.

[74] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17:397–415, 2015.

[75] Z. Daw and R. Cleaveland. Comparing model checkers for timed UML activity diagrams. *Science of Computer Programming*, 111:277–299, 2015.

[76] E. de Araújo Silva, E. Valentin, J. R. H. Carvalho, and R. da Silva Barreto. A survey of model driven engineering in robotics. *Journal of Computer Languages*, 62:101021, 2021.

[77] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182–197, 2002.

[78] P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

[79] D. Di Pompeo and M. Tucci. Search budget in multi-objective refactoring optimization: a model-based empirical study. In *48th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2022*, pages 406–413. IEEE, 2022.

[80] D. Di Pompeo, M. Tucci, A. Celi, and R. Eramo. A microservice reference case study for design-runtime interaction in MDE. In A. Bagnato, H. Brunelière, L. Burgueño, R. Eramo, and A. Gómez, editors, *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019), Eindhoven, The Netherlands, July 15 - 19, 2019*, volume 2405 of *CEUR Workshop Proceedings*, pages 23–32. CEUR-WS.org, 2019.

[81] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 2009.

[82] K. Eder, J. P. Gallagher, P. López-García, H. Muller, Z. Banković, K. Georgiou, R. Haemmerlé, M. V. Hermenegildo, B. Kafle, S. Kerrison, M. Kirkeby, M. Klemen, X. Li, U. Liqat, J. Morse, M. Rhiger, and M. Rosendahl. ENTRA: Whole-systems energy transparency. *Microprocessors and Microsystems*, 47:278–286, Nov. 2016.

[83] B. Y. Ekren and A. Akpunar. An open queuing network-based tool for performance estimations in a shuttle-based storage and retrieval system. *Applied Mathematical Modelling*, 89:1678–1695, 2021.

[84] A. Enrici, L. Apvrille, and R. Pacalet. A Model-Driven Engineering Methodology to Design Parallel and Distributed Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems*, 22:34:1–34:25, 2017.

[85] J. Erickson and K. Siau. Can UML be simplified? practitioner use of UML in separate domains. In E. Proper, T. A. Halpin, and J. Krogstie, editors, *Proceedings of the 12th International Workshop on Exploring Modeling Methods for Systems Analysis and Design, EMMSAD 2008, held in conjunction with the 19th Conference on Advanced Information Systems (CAiSE 2007), Trondheim, Norway, 11-15 June, 2007*, volume 365 of *CEUR Workshop Proceedings*, pages 81–90. CEUR-WS.org, 2007.

[86] H. Esmaeilzadeh, T. Cao, X. Yang, S. Blackburn, and K. McKinley. What is happening to power, performance, and software? *IEEE Micro*, 32(3):110–121, 2012.

[87] R. Etemaadi and M. R. V. Chaudron. New degrees of freedom in metaheuristic optimization of component-based systems architecture: Architecture topology and load balancing. *Sci. Comput. Program.*, 97:366–380, 2015.

[88] P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language*. SEI series in software engineering. Addison-Wesley, 2012.

[89] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2009.

[90] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2009.

[91] N. Frid, D. Ivošević, and V. Sruk. Elementary operations: a novel concept for source-level timing estimation. *Automatika*, 60(1):91–104, 2019.

[92] Fudan Software Engineering Laboratory. Train Ticket Booking System. `https://github.com/FudanSELab/train-ticket`. Accessed: 2023-04-12.

[93] K. Georgiou, S. Xavier-de Souza, and K. Eder. The iot energy challenge: A software perspective. *IEEE Embedded Systems Letters*, 10(3):53–56, 2018.

[94] S. Ghosh and S. Unnikrishnan. Reduced power consumption in wireless sensor networks using queue based approach. In *2017 International Conference on Advances in Computing, Communication and Control (ICAC3)*, pages 1–5, 2017.

[95] P. Giammatteo, F. V. Fiordigigli, L. Pomante, T. Di Mascio, and F. Caruso. Age gender classifier for edge computing. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4, 2019.

[96] M. Gleirscher, S. Foster, and J. Woodcock. New opportunities for integrated formal methods. *ACM Comput. Surv.*, 52(6), oct 2019.

[97] O. M. Group. A UML profile for MARTE: modeling and analysis of real-time embedded systems. Object Management Group, 2008.

[98] K. Grüttner, R. Görgen, S. Schreiner, F. Herrera, P. Peñil, J. Medina, E. Villar, G. Palermo, W. Fornaciari, C. Brandolese, D. Gadioli, E. Vitali, D. Zoni, S. Bocchio, L. Ceva, P. Azzoni, M. Poncino, S. Vinco, E. Macii, S. Cusenza, J. Favaro, R. Valencia, I. Sander, K. Rosvall, N. Khalilzad, and D. Quaglia. Contrex: Design of embedded mixed-criticality control systems under consideration of extra-functional properties. *Microprocessors and Microsystems*, 51:39 – 55, 2017.

[99] F. Gu, X. Zhang, M. Chen, D. Große, and R. Drechsler. Quantitative timing analysis of uml activity diagrams using statistical model checking. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, DATE '16, page 780–785, San Jose, CA, USA, 2016. EDA Consortium.

[100] S. Ha and J. Teich, editors. *Handbook of Hardware/Software Codesign*. Springer Netherlands, 2017.

[101] F. Han, P. Herrmann, and H. Le. Modeling and verifying real-time properties of reactive systems. In *Proceedings of the 2013 18th International Conference on Engineering of Complex Computer Systems*, ICECCS '13, page 14–23, USA, 2013. IEEE Computer Society.

[102] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[103] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[104] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. Cocome - the common component modeling example. In *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *LNCS*, pages 16–53, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[105] F. Herrera, J. Medina, and E. Villar. *Modeling Hardware/Software Embedded Systems with UML/MARTE: A Single-Source Design Approach*, pages 1–45. 01 2016.

[106] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 1*, pages 883–891. Curran Associates Inc., 2010.

[107] M. Hubner and J. Becker. *Multiprocessor system-on-chip: Hardware design and tool integration*. Springer, 01 2011.

[108] H. Ishibuchi, H. Masuda, and Y. Nojima. Sensitivity of performance evaluation results by inverted generational distance to reference points. In *IEEE Congress on Evolutionary Computation, CEC 2016, Vancouver, BC, Canada, July 24-29, 2016*, pages 1107–1114. IEEE, 2016.

[109] F.-C. Jiang, D.-C. Huang, and K.-H. Wang. Design approaches for optimizing power consumption of sensor node with n-policy m/g/1 queuing model. In *Proceedings of the 4th International Conference on Queueing Theory and Network Applications*, QTNA '09, New York, NY, USA, 2009. Association for Computing Machinery.

[110] M. Kessentini, H. A. Sahraoui, M. Boukadoum, and O. Benomar. Search-based model transformation by example. *Softw. Syst. Model.*, 11(2):209–226, 2012.

[111] A. Koziolek, H. Koziolek, and R. H. Reussner. Peropteryx: automated application of tactics in multi-objective software architecture optimization. In I. Crnkovic, J. A. Stafford, D. C. Petriu, J. Happe, and P. Inverardi, editors, *7th International Conference on the Quality of Software Architectures, QoSA 2011 and 2nd International Symposium on Architecting Critical Systems, ISARCS 2011. Boulder, CO, USA, June 20-24, 2011, Proceedings*, pages 33–42. ACM, 2011.

[112] S. Lee, A. Ermedahl, S. L. Min, and N. Chang. An accurate instruction-level energy consumption model for embedded risc processors. *SIGPLAN Not.*, 36(8):1–10, Aug. 2001.

[113] S. Lee, A. Ermedahl, S. L. Min, and N. Chang. Statistical derivation of an accurate energy consumption model for embedded processors, 2002.

[114] I. Letteri, G. Della Penna, and P. Caianiello. Feature selection strategies for http botnet traffic detection. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pages 202–210, 2019.

[115] C. Li, T. Altamimi, M. H. Zargari, G. Casale, and D. C. Petriu. Tulsa: A tool for transforming UML to layered queueing networks for performance analysis of data intensive applications. In N. Bertrand and L. Bortolussi, editors, *Quantitative Evaluation of Systems - 14th International Conference, QEST 2017, Berlin, Germany, September 5-7, 2017, Proceedings*, volume 10503 of *Lecture Notes in Computer Science*, pages 295–299. Springer, 2017.

[116] R. Li, R. Etemaadi, M. T. M. Emmerich, and M. R. V. Chaudron. An evolutionary multiobjective optimization approach to component-based software architecture design. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2011, New Orleans, LA, USA, 5-8 June, 2011*, pages 432–439. IEEE, 2011.

[117] R. Li, R. Etemaadi, M. T. M. Emmerich, and M. R. V. Chaudron. An evolutionary multiobjective optimization approach to component-based software architecture design. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2011, New Orleans, LA, USA, 5-8 June, 2011*, pages 432–439. IEEE, 2011.

[118] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *SoSyM*, 17:91–113, 2018.

[119] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide.* Cambridge University Press, USA, 2000.

[120] D. J. Lilja. Measuring computer performance: A practitioner's guide. *SIAM Review*, 43:383–384, 01 2001.

[121] k. Liu. A simulation based approach to estimate energy consumption for embedded processors. Master's thesis, Wrocław University of Technology, 2015.

[122] J. Livonius, H. Blume, and T. Noll. Flpa-based power modeling and power aware code optimization for a trimedia dsp. *Proceedingsof the ProRISC Workshop*, 01 2005.

[123] A. Lojo, L. Rubio, J. M. Ruano, T. Di Mascio, L. Pomante, E. Ferrari, I. G. Vega, F. K. Gürkaynak, M. L. Esnaola, V. Orani, and J. Abella. The ecsel fractal project: A cognitive fractal and secure edge based on a unique open-safe-reliable-low power hardware platform. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 393–400, 2020.

[124] Y. Z. Lun, A. D'Innocenzo, F. Smarra, I. Malavolta, and M. D. D. Benedetto. State of the art of cyber-physical systems security: An automatic control perspective. *Journal of Systems and Software*, 149:174 – 216, 2019.

[125] S. M, H. Blume, and T. Noll. Power estimation on functional level for programmable processors. *Advances in Radio Science - Kleinheubacher Berichte*, 2, 01 2004.

[126] S. Malik, M. Martonosi, and Y. S. Li. Static timing analysis of embedded software. *Proceedings of the 34th Design Automation Conference*, pages 147–152, 1997.

[127] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. *Softw. Qual. J.*, 25(2):473–501, 2017.

[128] T. Mariani and S. R. Vergilio. A systematic review on search-based refactoring. *Inf. Softw. Technol.*, 83:14–34, 2017.

[129] A. Martens, D. Ardagna, H. Koziolek, R. Mirandola, and R. H. Reussner. A hybrid approach for multi-attribute qos optimisation in component based software systems. In G. T. Heineman, J. Kofron, and F. Plasil, editors, *Research into Practice - Reality and Gaps, 6th International Conference on the Quality of Software Architectures, QoSA 2010, Prague, Czech Republic, June 23 - 25, 2010. Proceedings*, volume 6093 of *Lecture Notes in Computer Science*, pages 84–101. Springer, 2010.

[130] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In A. Adamson, A. B. Bondi, C. Juiz, and M. S. Squillante, editors, *Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering, San Jose, California, USA, January 28-30, 2010*, pages 105–116. ACM, 2010.

[131] W. McUmber and B. Cheng. A general framework for formalizing uml with formal languages. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 433–442, 2001.

[132] I. Meedeniya, B. Buhnova, A. Aleti, and L. Grunske. Architecture-driven reliability and energy optimization for complex embedded systems. In G. T. Heineman, J. Kofron, and F. Plasil, editors, *Research into Practice - Reality and Gaps, 6th International Conference on the Quality of Software Architectures, QoSA 2010, Prague, Czech Republic, June 23 - 25, 2010. Proceedings*, volume 6093 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2010.

[133] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malek, and J. P. Sousa. A framework for utility-based service oriented design in SASSY. In A. Adamson, A. B. Bondi, C. Juiz, and M. S.

Squillante, editors, *Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering, San Jose, California, USA, January 28-30, 2010*, pages 27–36. ACM, 2010.

[134] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernandez, B. Nordmoen, and M. Fritzsche. Where does model-driven engineering help? experiences from three industrial cases. *Software & Systems Modeling*, 12:619–639, 2013.

[135] Monika and O. P. Sangwan. Predicting software effort estimation using machine learning techniques. In *7th International Conference on Cloud Computing, Data Science Engineering - Confluence*, pages 92–98, 2017.

[136] Monsoon Solutions. Monsoon power monitor. `https://www.msoon.com/`. Accessed: 2021-09-26.

[137] A. Moro, F. Federici, G. Valente, L. Pomante, M. Faccio, and V. Muttillo. Hardware performance sniffers for embedded systems profiling. In *2015 12th International Workshop on Intelligent Solutions in Embedded Systems (WISES)*, pages 29–34, Oct 2015.

[138] V. Muttillo. J4cs: An early-stage statement-level metric for energy consumption of embedded sw. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–5, June 2019.

[139] V. Muttillo, P. Giammatteo, and V. Stoico. Statement-level timing estimation for embedded system design using machine learning techniques. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '21, page 257–264, New York, NY, USA, 2021. Association for Computing Machinery.

[140] V. Muttillo, P. Giammatteo, V. Stoico, and L. Pomante. An early-stage statement-level metric for energy characterization of embedded processors. *Microprocessors and Microsystems*, 77:103200, 2020.

[141] V. Muttillo, G. Valente, D. Ciambrone, V. Stoico, and L. Pomante. Hepsycode-rt: A real-time extension for an esl hw/sw co-design methodology. In *ACM International Conference Proceeding Series*, 2018.

[142] V. Muttillo, G. Valente, D. Ciambrone, V. Stoico, and L. Pomante. Hepsycode-rt: A real-time extension for an esl hw/sw co-design methodology. In *Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '18, pages 6:1–6:6, New York, NY, USA, 2018. ACM.

[143] V. Muttillo, G. Valente, L. Pomante, V. Stoico, F. D'Antonio, and F. Salice. Cc4cs: An off-the-shelf unifying statement-level performance metric for hw/sw technologies. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 119–122, New York, NY, USA, 2018. ACM.

[144] V. Muttillo, G. Valente, L. Pomante, V. Stoico, F. D'Antonio, and F. Salice. Cc4cs: An off-the-shelf unifying statement-level performance metric for hw/sw technologies. In *Companion*

of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18, pages 119–122, New York, NY, USA, 2018. ACM.

[145] V. Muttillo, G. Valente, L. Pomante, V. Stoico, F. D'Antonio, and F. Salice. Cc4cs: An off-the-shelf unifying statement-level performance metric for hw/sw technologies. In *Companion of the 2018 ACM/SPEC International Conference*, New York, NY, USA, 2018. ACM.

[146] F. N. Najm. A survey of power estimation techniques in vlsi circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):446–455, Dec 1994.

[147] A. J. Nebro, J. J. Durillo, and M. Vergne. Redesigning the jmetal multi-objective optimization framework. In S. Silva and A. I. Esparcia-Alcázar, editors, *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, pages 1093–1100. ACM, 2015.

[148] J. E. Neilson, C. M. Woodside, D. C. Petriu, and S. Majumdar. Software bootlenecking in client-server systems and rendezvous networks. *IEEE Trans. Software Eng.*, 21(9):776–782, 1995.

[149] Y. Ni, X. Du, P. Ye, L. L. Minku, X. Yao, M. Harman, and R. Xiao. Multi-objective software performance optimisation at the architecture level using randomised search rules. *Inf. Softw. Technol.*, 135:106565, 2021.

[150] I. Nikolaidis. Arm system-on-chip architecture, 2nd edition [book review]. *Network, IEEE*, 14:4–4, 12 2000.

[151] S. Nikolaidis, N. Kavvadias, T. Laopoulos, L. Bisdounis, and S. Blionas. Instruction level energy modeling for pipelined processors. *J. Embedded Comput.*, 1(3):317–324, Aug. 2005.

[152] O. S. D. Organization. About the UML Profile for MARTE Specification Version 1.2, 2023. Last accessed 25 January 2023.

[153] O. S. D. Organization. The Unified Modeling Language 2.5.1 Specification, 2023. Last accessed 25 January 2023.

[154] A. Ouni, M. Kessentini, K. Inoue, and M. Ó. Cinnéide. Search-based web service antipatterns detection. *IEEE Trans. Serv. Comput.*, 10(4):603–617, 2017.

[155] A. Ouni, R. G. Kula, M. Kessentini, and K. Inoue. Web service antipatterns detection using genetic programming. In S. Silva and A. I. Esparcia-Alcázar, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, pages 1351–1358. ACM, 2015.

[156] M. Oyamada, F. Wagner, M. Bonaciu, W. Cesário, and A. Jerraya. Software performance estimation in mpsoc design. In *Asia and South Pacific Design Automation Conference*, volume 0, pages 38–43, 01 2007.

[157] B. Ozisikyilmaz, G. Memik, and A. Choudhary. Machine learning models to predict performance of computer system design alternatives. In *2008 37th Int. Conf. on Parallel Processing*, pages 495–502, 2008.

[158] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[159] Y. Park, S. Pasricha, F. J. Kurdahi, and N. Dutt. A multi-granularity power modeling methodology for embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(4):668–681, April 2011.

[160] L. Pomante, V. Muttillo, M. Santic, and P. Serri. SystemC-based electronic system-level design space exploration environment for dedicated heterogeneous multi-processor systems. *Microprocessors and Microsystems*, 72, Feb. 2020.

[161] C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[162] A. Rago, S. A. Vidal, J. A. Diaz-Pace, S. Frank, and A. van Hoorn. Distributed quality-attribute optimization of software architectures. In *Proceedings of the 11th Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS 2017, Fortaleza, CE, Brazil, September 18 - 19, 2017*, pages 7:1–7:10. ACM, 2017.

[163] A. Ramírez, J. R. Romero, and S. Ventura. A survey of many-objective optimisation in search-based software engineering. *J. Syst. Softw.*, 149:382–395, 2019.

[164] M. Ray and D. P. Mohapatra. Multi-objective test prioritization via a genetic algorithm. *Innov. Syst. Softw. Eng.*, 10(4):261–270, 2014.

[165] A. Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[166] F. Rosenberg, M. B. Müller, P. Leitner, A. Michlmayr, A. Bouguettaya, and S. Dustdar. Metaheuristic optimization of large-scale qos-aware service compositions. In *2010 IEEE International Conference on Services Computing, SCC 2010, Miami, Florida, USA, July 5-10, 2010*, pages 97–104. IEEE Computer Society, 2010.

[167] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria. Instruction-level power estimation for embedded vliw cores. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign. CODES 2000 (IEEE Cat. No.00TH8518)*, pages 34–38, May 2000.

[168] I. Sander, A. Jantsch, and S.-H. Attarzadeh-Niaki. ForSyDe: System Design Using a Functional Language and Models of Computation. In *Handbook of Hardware/Software Codesign*, pages 1–42. Springer Netherlands, Dordrecht, 2017.

[169] D. Sciuto, F. Salice, L. Pomante, and W. Fornaciari. Metrics for design space exploration of heterogeneous multiprocessor embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, page 55–60, New York, NY, USA, 2002. ACM.

[170] E. Senn, N. Julien, J. Laurent, and E. Martin. Power consumption estimation of a c program for data-intensive applications. In B. Hochet, A. J. Acosta, and M. J. Bellido, editors, *Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation*, pages 332–341, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[171] M. Siegesmund. *Embedded C Programming: Techniques and Applications of C and PIC MCUS*. Newnes, Newton, MA, USA, 1st edition, 2014.

[172] T. Simunic, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 867–872, June 1999.

[173] A. Sinha and A. P. Chandrakasan. Jouletrack-a web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 220–225, June 2001.

[174] F. Smarra, G. D. Di Girolamo, V. De Iuliis, A. Jain, R. Mangharam, and A. D'Innocenzo. Data-driven switching modeling for mpc using regression trees and random forests. *Nonlinear Analysis: Hybrid Systems*, 36:100882, 2020.

[175] C. U. Smith and L. G. Williams. Software performance antipatterns. In *Second International Workshop on Software and Performance, WOSP 2000, Ottawa, Canada, September 17-20, 2000*, pages 127–136. ACM, 2000.

[176] C. U. Smith and L. G. Williams. Software performance antipatterns; common performance problems and their solutions. In *27th International Computer Measurement Group Conference, Anaheim, CA, USA, December 2-7, 2001*, pages 797–806. Computer Measurement Group, 2001.

[177] C. U. Smith and L. G. Williams. More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot. In *29th International Computer Measurement Group Conference*, pages 717–725, 2003.

[178] C. U. Smith and L. G. Williams. Software performance engineering. In L. Lavagno, G. Martin, and B. Selic, editors, *UML for Real - Design of Embedded Real-Time Systems*, pages 343–365. Kluwer, 2003.

[179] V. Stoico. A model-driven approach for early verification and validation of embedded systems. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion, Fukuoka, Japan, October 10-15, 2021*, pages 684–688. IEEE, 2021.

[180] H. Sultan, G. Ananthanarayanan, and S. R. Sarangi. Processor power estimation techniques: A survey. *Int. J. High Perform. Syst. Archit.*, 5(2):93–114, May 2014.

[181] S. Sultan and S. Masud. Rapid software power estimation of embedded pipelined processor through instruction level power model. In *2009 International Symposium on Performance Evaluation of Computer Telecommunication Systems*, volume 41, pages 27–34, July 2009.

[182] J. Suryadevara, C. Seceleanu, F. Mallet, and P. Pettersson. Verifying marte/ccsl mode behaviors using uppaal. In *Proceedings of the 11th International Conference on Software Engineering and Formal Methods - Volume 8137*, SEFM 2013, page 1–15, Berlin, Heidelberg, 2013. Springer-Verlag.

[183] J. Teich. Hardware/Software Codesign: The Past, the Present, and Predicting the Future. *Proceedings of the IEEE*, 100:1411–1430, 2012.

[184] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 384–390, Nov 1994.

[185] V. Tiwari, S. Malik, A. Wolfe, and M. T. . Lee. Instruction level power analysis and optimization of software. In *Proceedings of 9th International Conference on VLSI Design*, pages 326–328, Jan 1996.

[186] A. Trendowicz. *Software Cost Estimation, Benchmarking, and Risk Assessment: The Software Decision-Makers' Guide to Predictable Software Development.* Springer Science & Business Media, 2013.

[187] M. Tribastone, P. Mayer, and M. Wirsing. Performance prediction of service-oriented systems with layered queueing networks. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 51–65, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[188] F. Vahid and T. Givargis. *Embedded System Design: A Unified Hardware/Software Introduction.* John Wiley & Sons, Inc., USA, 1st edition, 2001.

[189] G. Valente, T. Di Mascio, G. D'Andrea, and L. Pomante. Dynamic partial reconfiguration profitability for real-time systems. *IEEE Embedded Systems Letters*, pages 1–1, 2020.

[190] G. Valente, T. Di Mascio, L. Pomante, and V. Stoico. An esl methodology for hw/sw co-design of monitorable embedded systems: the "design for monitorability" project - work-in-progress. In *2020 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 40–42, 2020.

[191] R. Verdecchia, P. Lago, C. Ebert, and C. De Vries. Green it and green software. *IEEE Software*, 38(6):7–15, 2021.

[192] E. Villar, J. Merino, H. Posadas, R. Henia, and L. Rioux. Mega-modeling of complex, distributed, heterogeneous cps systems. *Microprocessors and Microsystems*, 78:103244, 2020.

[193] WattsUp. Watts up? pro power monitor. `https://github.com/isaaclino/wattsup`. Accessed: 2023-04-05.

[194] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell. *Experimentation in Software Engineering.* Springer, 2012.

[195] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering.* Springer, Berlin, Heidelberg, 2012.

[196] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):19:1–19:36, 2009.

[197] C. M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer. Performance by unified model analysis (PUMA). In *Proceedings of the Fifth International Workshop on Software and Performance, WOSP 2005, Palma, Illes Balears, Spain, July 12-14, 2005*, pages 1–12. ACM, 2005.

[198] M. Woodside and G. Franks. Tutorial introduction to layered modeling of software performance, 2002.

[199] G. Yang, A. Sangiovanni-Vincentelli, Y. Watanabe, and F. Balarin. Separation of concerns: overhead in modeling and efficient simulation techniques. In *Proceedings of the 4th ACM international conference on Embedded software*, EMSOFT '04, pages 44–53, Sept. 2004.

[200] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proceedings 37th Design Automation Conference*, pages 340–345, June 2000.

[201] Y. Zhang and W. Li. Modeling and energy consumption evaluation of a stochastic wireless sensor network. *EURASIP Journal on Wireless Communications and Networking*, 2012(1):282, Sept. 2012.

[202] A. Zhou, Y. Jin, Q. Zhang, B. Sendhoff, and E. P. K. Tsang. Combining model-based and genetics-based offspring generation for multi-objective optimization using a convergence criterion. In *IEEE International Conference on Evolutionary Computation, CEC 2006, part of WCCI 2006, Vancouver, BC, Canada, 16-21 July 2006*, pages 892–899. IEEE, 2006.

[203] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Trans. Software Eng.*, 47(2):243–260, 2021.

[204] Y. Zhou, L. Baresi, and M. Rossi. Towards a Formal Semantics for UML/MARTE State Machines Based on Hierarchical Timed Automata. *Journal of Computer Science and Technology*, 28(1):188–202, 2013.

[205] Q. Zhu and A. Sangiovanni-Vincentelli. Codesign Methodologies and Tools for Cyber–Physical Systems. *Proceedings of the IEEE*, 106(9):1484–1500, Sept. 2018.

[206] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Trans. Evol. Comput.*, 3(4):257–271, 1999.

[207] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Trans. Evol. Comput.*, 7(2):117–132, 2003.