PERFORMANCE MODELS FOR EMBEDDED SOFTWARE PRODUCT LINES

Dissertation

zur Erlangung des Grades einer

DOKTORIN DER NATURWISSENSCHAFTEN

der Universität Osnabrück
am Fachbereich Mathematik/Informatik/Physik

von

BIRTE KRISTINA FRIESEL

Osnabrück

2024

Tag der mündlichen Prüfung:	17. Februar 2025
Dekan:	Prof. Dr. Tim Römer
Gutachter:	Prof. Dr.-Ing. Olaf Spinczyk
	Prof. Dr.-Ing. Timo Hönig

ABSTRACT

Software product lines deal with functional properties (features) of configurable software systems. However, these systems also have non-functional properties such as latency, energy usage, or memory footprint. Performance models formalize knowledge about those properties and their relation to individual features; they are often crucial to building competitive products. While the literature offers a variety of performance modeling methods for conventional product lines, these do not address configurable device drivers and hardware components. Moreover, many approaches either rely on time-consuming and error-prone manual annotations, or result in models that are so complex that engineers cannot gain insights by analysing them.

This thesis covers interpretable performance models for configurable embedded systems, including software product lines, configurable hardware components, and hybrid product lines that combine both. It focuses on automation for the entire life cycle, ranging from unattended data acquisition over machine learning methods for model generation to performance-aware product line configuration. In doing so, it shows that the disjoint Product Line Engineering and Internet of Things communities address similar challenges, and combines and extends aspects from both to obtain accurate and interpretable performance models for hybrid and hardware-centric software systems.

Its key contribution is the Regression Model Tree data structure and machine learning method. Regression model trees resemble the structure of feature models, but exclusively rely on benchmark data for model generation. They can be used with software product lines, configurable hardware components and hybrid product lines, and can be learned – and understood – even if no feature model is available. An evaluation on eight product lines and product line-like system components shows that regression model trees are more accurate and less complex than other tree-based modeling methods when applied to hybrid product lines and hardware components.

In addition, this thesis contributes a method for energy measurement automation even if no out-of-band synchronization methods are available, an analysis of whether performance models should be part of feature models or kept separate, a product line perspective on configurable hardware components and device drivers, and case studies that apply regression model trees to real-world product lines. The analysis finds that performance models should be kept separate; regression model trees follow this approach. The case studies cover both manual model analysis and tool-assisted performance-aware configuration of Kconfig-based product lines.

ACKNOWLEDGMENTS

This thesis builds upon decades of prior research and dozens of people who directly and indirectly supported it, and would not be possible without either of those.

First and foremost, I would like to thank my advisor, Prof. Dr.-Ing. Olaf Spinczyk, for giving me the opportunity to pursue this endeavour to begin with and for nudging me into the right direction when needed. Similarly, I am thankful to my parents for supporting me on the long way towards handing in the dissertation.

I am also grateful for the colleagues around the two research groups that I was part of during this journey. This includes (but is certainly not limited to) Hendrik, Ulrich, Horst and Alex of the Embedded System Software group at TU Dortmund, and Marcel of the Embedded Software Systems group at Universität Osnabrück. No matter whether it was research-related or not, you always had a helpful comment at hand. I have fond memories of occasionally getting sore face muscles just from laughing, especially during the 11:30-and-not-a-minute-later Mensa breaks in Dortmund.

Outside of academia, I would like to extend special thanks to Markus, with whom I have had the pleasure of sharing the experience called "life" for the past eleven years — I do not wish to know where I would be without you. I am also very thankful to my close friends, notably Lara, Christian, Nadine, Larissa, and Luzia.

Finally, I would like to acknowledge that occasional random encounters are important and helpful additions to long-standing support from family, friends, and colleagues. This includes people I met – and occasionally even kept in touch with – at conferences, the local hackspace community, and the numerous academia-adjacent entities that I had the joy of interacting with online.

PUBLICATIONS

Parts of the contributions presented in this thesis have been published in the following peer-reviewed conference and workshop proceedings.

1. Birte Friesel, Markus Buschhoff, and Olaf Spinczyk. "Annotations in Operating Systems with Custom AspectC++ Attributes". In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. PLOS '17. Shanghai, China: Association for Computing Machinery, Oct. 2017, pp. 36–42. ISBN: 978-1-4503-5153-9. DOI: 10.1145/3144555.3144561 [FBS17]

2. Birte Friesel, Markus Buschhoff, and Olaf Spinczyk. "Parameter-Aware Energy Models for Embedded-System Peripherals". In: *Proceedings of the 13th International Symposium on Industrial Embedded Systems*. SIES '18. Graz, Austria: IEEE, June 2018. DOI: 10.1109/SIES.2018.8442096 [FBS18]

3. Birte Friesel and Olaf Spinczyk. "Poster Abstract: I^2C Considered Wasteful: Saving Energy with Host-Controlled Pull-Up Resistors". In: *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*. IPSN '19. Montreal, QC, Canada: Association for Computing Machinery, Apr. 2019, pp. 315–316. ISBN: 978-1-4503-6284-9. DOI: 10.1145/3302506.3312606 [FS19]

4. Birte Friesel, Lennart Kaiser, and Olaf Spinczyk. "Automatic Energy Model Generation with MSP430 EnergyTrace". In: *Proceedings of the Workshop on Benchmarking Cyber-Physical Systems and Internet of Things*. CPS-IoTBench '21. Nashville, TN, USA: Association for Computing Machinery, May 2021, pp. 26–31. ISBN: 978-1-4503-8439-1. DOI: 10.1145/3458473.3458822 [FKS21]

5. Birte Friesel and Olaf Spinczyk. "Data Serialization Formats for the Internet of Things". In: *Electronic Communications of the EASST* 80 (Sept. 2021). DOI: 10.14279/tuj.eceasst.80.1134 [FS21]

6. Birte Friesel and Olaf Spinczyk. "Performance is not Boolean: Supporting Scalar Configuration Variables in NFP Models". In: *Tagungsband des FG-BS Frühjahrstreffens 2022*. Hamburg, Germany: Gesellschaft für Informatik e.V., Mar. 2022. DOI: 10.18420/fgbs2022f-03 [FS22b]

7. Birte Friesel and Olaf Spinczyk. "Regression Model Trees: Compact Energy Models for Complex IoT Devices". In: *Proceedings of*

the Workshop on Benchmarking Cyber-Physical Systems and Internet of Things. CPS-IoTBench '22. Milan, Italy: IEEE, May 2022, pp. 1–6. DOI: 10.1109/CPS-IoTBench56135.2022.00007 [FS22c]

8. Birte Friesel and Olaf Spinczyk. "Black-Box Models for Non-Functional Properties of AI Software Systems". In: *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. CAIN '22. Pittsburgh, PA, USA: Association for Computing Machinery, May 2022, pp. 170–180. ISBN: 978-1-4503-9275-4. DOI: 10.1145/3522664.3528602 [FS22a]

9. Birte Friesel et al. "On the Relation of Variability Modeling Languages and Non-Functional Properties". In: *Proceedings of the 26th International Systems and Software Product Line Conference - Volume B*. SPLC '22. Graz, Austria: Association for Computing Machinery, Sept. 2022, pp. 140–144. ISBN: 978-1-4503-9206-8. DOI: 10.1145/3503229.3547055 [Fri+22b]

10. Birte Friesel et al. "kconfig-webconf: Retrofitting Performance Models onto Kconfig-Based Software Product Lines". In: *Proceedings of the 26th International Systems and Software Product Line Conference - Volume B*. SPLC '22. Graz, Austria: Association for Computing Machinery, Sept. 2022, pp. 58–61. ISBN: 978-1-4503-9206-8. DOI: 10.1145/3503229.3547026 [Fri+22a]

11. Birte Friesel, Marcel Lütke Dreimann, and Olaf Spinczyk. "A Full-System Perspective on UPMEM Performance". In: *Proceedings of the 1st Workshop on Disruptive Memory Systems*. DIMES '23. Koblenz, Germany: Association for Computing Machinery, Oct. 2023, pp. 1–7. ISBN: 979-8-4007-0300-3. DOI: 10.1145/3609308.3625266 [FLS23]

I have extended and re-evaluated some of the corresponding algorithms since then. Hence, the evaluation data sets and results presented in this thesis are not necessarily identical with the ones presented in the publications listed above.

The tools, algorithms, and observations presented in this thesis have also contributed to the following peer-reviewed publications.

1. Robert Falkenberg et al. "PhyNetLab: An IoT-Based Warehouse Testbed". In: *Proceedings of the Federated Conference on Computer Science and Information Systems*. FedCSIS '17. Prague, Czech Republic: IEEE, Sept. 2017, pp. 1051–1055. DOI: 10.15439/2017F267 [Fal+17]

2. Markus Buschhoff, Birte Friesel, and Olaf Spinczyk. "Energy Models in the Loop". In: *Procedia Computer Science* 130 (2018). Proceedings of the 9th International Conference on Ambient

Systems, Networks and Technologies (ANT 2018) / the 8th International Conference on Sustainable Energy Information Technology (SEIT 2018) / Affiliated Workshops, pp. 1063–1068. ISSN: 1877-0509. DOI: 10.1016/j.procs.2018.04.154 [BFS18]

3. Marcel Lütke Dreimann, Birte Friesel, and Olaf Spinczyk. "HetSim: A Simulator for Task-based Scheduling on Heterogeneous Hardware". In: *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*. ICPE '24 Companion. London, UK: Association for Computing Machinery, May 2024, pp. 261–268. ISBN: 979-8-4007-0445-1. DOI: 10.1145/3629527.3652275 [LFS24]

Finally, I have supervised the following Bachelor's theses over the course of my research.

1. Leon Nienhüser. "Bewertung der Energieeffizienz FRAM-basierter Zustandssicherungsverfahren". Feb. 2020.

2. Kevin Lass. "Automatische Zustandssicherung und -wiederherstellung für Gerätetreiber". Aug. 2020.

3. Janis Falkenhagen. "Verfeinerung parametrisierter Energiemodelle mithilfe von Changepoint Detection". Sept. 2020.

4. Lennart Kaiser. "Automatisierte Energiemodellerstellung mit MSP430 EnergyTrace". Sept. 2020.

5. Tom Brüning. "Entwicklung eines AR-basierten Demonstrators für KI in ressourcenbeschränkten eingebetteten Systemen". Sept. 2023.

6. Johannes Horas. "Generierung von ressourcenoptimierten Entscheidungsbäumen für eingebettete Systeme". May 2024.

CONTENTS

1 INTRODUCTION

From an outside perspective, building an embedded system may seem trivial. After all, there is a wide range of commercially available offers for hardware components as well as a variety of commercial and open-source software components to choose from. Hardware components include processors, sensors, actuators, and communication interfaces; software components include operating systems, user-space applications, and data serialization formats and libraries.

Consider a wireless sensor node that regularly performs air quality measurements and wirelessly transmits them to a central hub. One might assume that the system designer simply needs to combine a suitable air quality sensor, microcontroller, and radio chip with an operating system, a radio protocol, and some configuration and glue code. Once they have verified that the result fulfils the *functional requirements* (i.e., it successfully measures and transmits air quality data), they can deploy it or start selling it.

Of course, in practice, it is not so easy. A battery-powered environmental sensor installed in a remote location defeats its purpose if it only works for a few days before the battery is empty, and an operating system kernel is useless if it takes up so much memory that user-space applications frequently encounter out-of-memory errors. Hence, in order to build a useful product, developers need to take *non-functional requirements* such as battery runtime or memory usage into account [Glio7]. These depend on the *non-functional properties* (or *performance attributes*) of each system component, which in turn depend on the configuration and combination of system components: a compute-intensive algorithm on an energy-efficient CPU or a light-weight algorithm on a less efficient CPU may be fine, while an inefficient algorithm on an inefficient CPU is not.

This thesis takes a deeper look into the what, why, and how of performance attributes and performance models for embedded system components. Specifically, it looks into ways of predicting how a component's configuration affects its performance attributes, and how performance models and *product lines* interact. At this point, we can think of product lines as configurable software or hardware components – I will provide a detailed definition, and an explanation as to why this simplification is not quite correct, in Chapter 2. Before that, let us look into challenges that embedded developers face when dealing with non-functional requirements, corresponding research questions, and the resulting contributions to the state of the art.

1.1 CHALLENGES

The importance of performance attributes and non-functional requirements stems from the variety of resource constraints faced by embedded systems. Available energy, ROM and RAM space, and processing power are typically limited and should be used responsibly to minimize hardware cost and maximise operating time. However, functional requirements and system properties do not account for this – they only describe aspects such as environmental attributes supported by a certain sensor, or operating system APIs exposed to user-space applications.

Performance attributes[1] describe the energy usage of hardware components, memory usage of software components, and similar. They do not explain *what* a component does, but *how* efficient, expensive, or similar it is. For instance, if two algorithms A and B have identical functional attributes, but A only needs half as much processing time, choosing A over B may allow the system designer to use a less powerful, and thus cheaper, microcontroller. Similarly, if a radio chip supports a low-power sleep mode, using this sleep mode may require less power than keeping it idle all the time. However, as starting a transmission from sleep mode typically takes more time than doing so from idle mode, the expected usage scenario determines whether using sleep mode really is more efficient.

Non-functional requirements may state, for instance, that a certain product must have a minimum battery runtime of four weeks under certain usage conditions. In order to reason about them, system designers need to know the non-functional properties of each system component. This is not a trivial task: many software and hardware components resemble product lines that offer a variety of configuration options, all of which can influence their non-functional properties in often unexpected and undocumented manners [Ach+22]. So, there is not just a choice between different hardware or software components, but also between different configurations of the same component.

In theory, system designers could run benchmarks for each different combination and configuration of system components that fulfils the specified functional requirements, and then determine the optimal setup for their non-functional requirements. In practice, this is not feasible, as n boolean configuration options result in up to 2^n different configurations, and software projects like busybox or the Linux kernel expose thousands of options.

Performance models reduce the need for benchmarks by predicting non-functional properties of arbitrary system configurations. If system designers have such a tool at their disposal, they can obtain estimates for performance attributes of individual system components. This

1 Performance attributes appear under a variety of names; common synonyms in the literature include non-functional properties and extra-functional properties [DZT12].

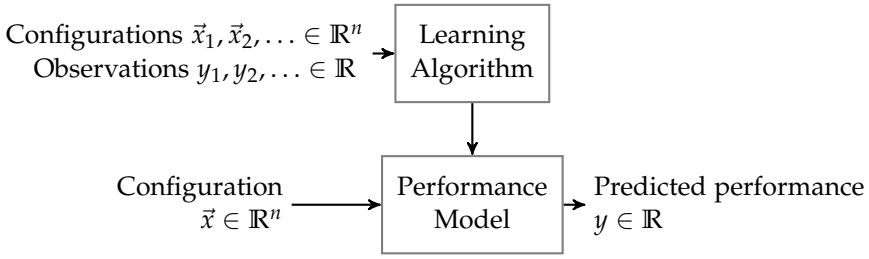


Figure 1.1: Relation between learning algorithms and performance models.

allows them to reason about the effect of toggling individual configuration options before a prototype has been built, or to devise performance-aware run-time algorithms [SSS10; HK17; Her+21]. As all models come with a certain level of uncertainty, benchmarks are still relevant for verification of model predictions, e.g. before committing on a configuration for manufacturing [Hur+11]. However, time-consuming benchmark campaigns that cover all valid system configurations are no longer necessary.

Ideally, a performance model is both accurate enough to generate useful predictions and simple enough to be understandable by humans, thus allowing them to reason about the effect of individual configuration options by looking at the model rather than by comparing predictions for dozens of pairs of configurations. Otherwise, users should at least have helpful tooling that annotates the performance effect of configuration options. At the same time, performance models do not appear out of thin air, and generating or updating them is at least as important as knowing how to use them [Har+16]. Machine learning algorithms help with this, but may rely on manually specified model templates or come up with complex models that are beyond human understanding.

Fig. 1.1 illustrates the relation between benchmarks, machine learning algorithm, and the resulting performance model. First, domain engineers perform a series of benchmarks that cover a range of system configurations $\vec{x}_i$ and obtain corresponding performance observations y_i (e.g. latency or power). Next, they pass configurations and observations to a machine learning algorithm that learns to predict performance y from configuration $\vec{x}$. Embedded developers can then use the resulting performance model to predict the performance of individual system configurations, simulate how changing configuration options would affect performance, and – if the model is understandable – gain further insights into system behaviour. Crucially, at this stage, they do not need access to raw observations or a test bench setup.

These challenges and goals finally bring us to the title of this thesis: *performance models for embedded software product lines*. I have spent the past years examining this topic from several angles, ranging from automated energy measurements over machine learning algorithms for performance model generation to tools for performance-aware

product line configuration. This document serves as a collection of my findings and scientific contributions towards the aforementioned goals. It combines extended and updated versions of peer-reviewed papers that I authored over the course of my PhD studies with a common background, motivation, and set of evaluation targets.

1.2 GOALS

The concept of performance models for configurable hardware and software components is far from being a new idea. It has been around for at least two decades, and two different scientific communities have been examining it closely.

On the one hand, the Cyber-Physical Systems and Internet of Things (CPS and IoT) community uses *energy models* to predict the energy requirements and timing behaviour of embedded systems and individual hardware components. The focus lies on hardware states (e.g. transmit, receive, and idle modes of a radio chip), transitions between them, and their timing and energy attributes. A key component of energy model generation is benchmark automation, as manual energy measurements are tedious and error-prone [ZO13]. Running energy benchmarks also requires suitable measurement equipment, and a benchmark application that exercises all hardware states and transitions.

On the other hand, the Software Product Line Engineering (SPLE) community uses *non-functional property models* to predict non-functional attributes of software product lines. Here, the focus lies on *variability models* that help manage the design space by organizing individual configuration options in a tree hierarchy and specifying cardinalities and dependencies. Another important aspect is handling the hundreds or even tens of thousands of configuration options that prohibit an exhaustive state space exploration and hinder reasoning about the importance of individual options. Instead, benchmarks first need to decide on a sampling strategy, and models must be able to predict non-functional attributes of previously unseen configurations.

Both communities have a variety of additional names for energy models and non-functional property models. These include resource models, power models, quality models, performance-influence models, and performance prediction models [VS08; McC+11; Sie+12b; Sie+15; Guo+18]. In this thesis, I use the umbrella term *performance models* to cover all of them.

As far as I am aware, there is little overlap or cooperation between CPS/IoT and SPLE researchers. While there are indeed differences in the challenges they face and priorities they set, I do not consider them to be prohibitive, and in fact think that both communities can benefit from each other. Hence, this thesis stands at the intersection of Energy Modeling and Product Line Engineering research. It covers

performance models for hardware and software components, and also examines hybrid product lines whose variability entails both hardware and software configuration options.

More specifically, my goal is to make performance models more accessible to the general public by minimizing the amount of specialized equipment, knowledge, and manual labour required for data acquisition, performance model generation, and performance model usage. All methods should be applicable to performance models for software product lines, energy models for embedded devices, and hybrid product lines that encompass both. At the same time, I want the automatically generated models to provide insights into system behaviour rather than just behaving as black boxes. Again, gaining these insights should not require an extensive skill set: performance models should be simple enough to be understandable just from looking at them, and performance-aware configuration software should be intuitive.

1.3 CONTRIBUTIONS

On the way towards unattended data acquisition, this thesis contributes the *dfatool* benchmark and model generation framework that supports both Kconfig-based software product lines and energy measurements of embedded peripherals. The latter requires a method for synchronizing benchmark events to energy measurements in order for its automation to work. However, the *out-of-band signals* that conventional synchronization methods rely on may not be available e.g. due to hardware not exposing suitable outputs or due to lack of funding for measurement equipment with appropriate inputs.

This challenge leads to research question **RQ1**: are automated and accurate CPS/IoT energy measurements feasible on hardware that lacks suitable out-of-band synchronization methods? As part of the answer, I will present a generic *drift compensation* algorithm that exclusively relies on on-board timers and in-band signalling, and show how it allows for automated energy measurements on $20 commercial off-the-shelf hardware.

When it comes to performance model generation, the first question is how variability model and performance model should relate. Hence, **RQ2** is: should performance models be integrated into variability models, or should they be separate entities? This is a fundamental question, as it dictates requirements for performance model structure and machine learning methods.

In a similar vein, the differences and similarities between non-functional property models and energy models beg the question whether it is viable to devise a common machine learning algorithm for both domains. I.e., **RQ3**: can a common machine learning algorithm for SPLE and CPS/IoT performance models provide lower prediction

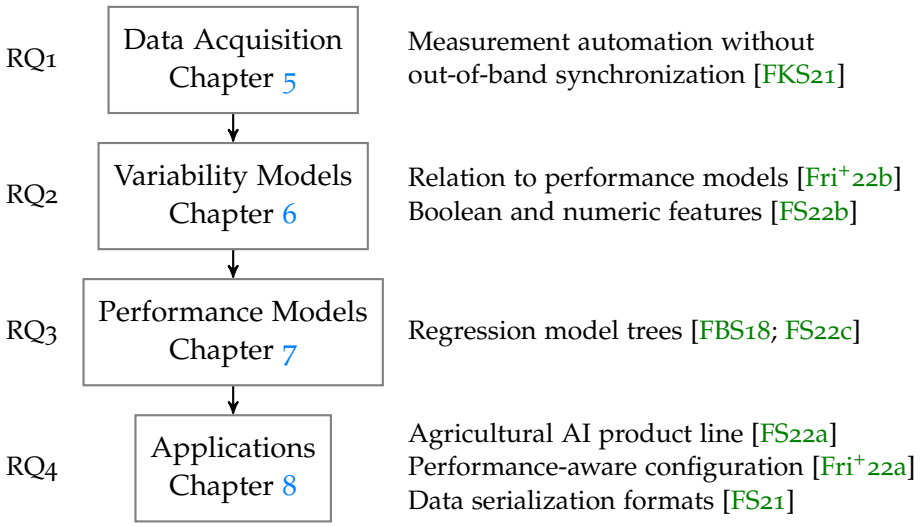


Figure 1.2: Research questions addressed in this thesis as well as contributions and corresponding publications in relation to a typical performance model generation and usage workflow.

error and model complexity than conventional approaches, without requiring manually provided domain information or model structure? Here, my key contribution is the novel *regression model tree* machine learning algorithm and performance model data structure.

Finally, model usage covers practical considerations. These include performance trade-offs in an artificial intelligence (AI) application and in data serialization format selection for wireless sensor networks. The AI application deviates from typical approaches in the literature by using an SPL-inspired black-box approach rather than the white-box variant that is common in the AI domain. This warrants an additional question (**RQ4**): are product line engineering and performance modeling techniques also applicable to product lines that cover soft- and hardware variability? In addition, I contribute a utility for *performance-aware configuration* of Kconfig-based product lines.

Fig. 1.2 gives an overview of the research questions, contributions, and publications covered in this thesis. It follows the typical workflow of performance model generation for an existing product line, starting out with data acquisition. Once all benchmark data is available, a machine learning algorithm (such as regression model trees) can build a performance model, and users can apply it to real-world engineering and optimization tasks.

1.4 STRUCTURE

The next two chapters cover software product lines and performance models for software product lines (Chapter 2) as well as energy models for hardware peripherals (Chapter 3). This includes an introduction

to variability models, related work on performance models and machine learning algorithms, quality metrics that will be used for model evaluation, and other definitions that are relevant throughout the entire document. They also introduce the product lines and hardware components that serve as application examples and evaluation targets throughout this thesis. Chapter 4 provides the problem statement: a re-cap of the research questions and how they relate to the state of the art presented in the introduction chapters. The following four chapters answer RQ1 through RQ4 and present additional contributions.

First, Chapter 5 covers benchmark generation and data acquisition. For software product lines, this boils down to an implementation of existing data acquisition methods. For embedded peripherals, automation is not always so easy – here, I will present my answer to RQ1. Next, Chapter 6 examines the relation between variability models and performance models, and what kind of configuration options (boolean or numeric) a performance model should consider for performance prediction. Based upon these findings, Chapter 7 motivates, defines, and evaluates the regression model tree data structure and machine learning algorithm. Chapter 8 contains further applications of performance models to practical issues such as data serialization format selection in wireless sensor networks and performance optimization of an agricultural AI product line. Finally, chapter 9 concludes and gives an outlook into future research avenues.

2

SOFTWARE PRODUCT LINES

To introduce the concepts of software product lines and performance models, let us consider the case of an experienced embedded engineer who wants to design a minimal embedded *Operating System* (OS) to support the evaluation of hardware and software components for *Internet of Things* (IoT) devices. It must support several target architectures, provide device drivers for common peripheral interfaces, and should be limited to the bare necessities apart from that. Since the target architecture dictates which compiler and low-level implementations (e.g. context switch code or on-board peripheral drivers) must be used to obtain a binary image that can be executed on it, the engineer cannot compile a single OS image that works on any architecture. However, maintaining an individual operating system code base for each target architecture is not desirable either.

Thanks to their domain knowledge, the engineer knows that an operating system contains numerous architecture-agnostic components that do not need to be adapted for different target architectures. In this case, architecture-specific low-level modules are the only *variable* components within the operating system. If the engineer ensures that architecture-specific components are only linked into the operating system image when needed, and defines a common programming interface for accessing them, they can compile binaries for any supported architecture by selectively including / excluding components during compilation. This reasoning about features, components, and interfaces between them is called *domain engineering*. The resulting operating system is not a complete and ready-to-use *product* by itself, but rather a *Software Product Line* (SPL) that can be compiled into several different products depending on the selected *features* [Sin+07; Keh+21]. In this example, each target architecture is a feature, so there is one operating system product per target architecture.

Now, assume that a developer wants to use this operating system to run a signal processing application. The application is simple enough to run on any supported target architecture – hence, all products have the same *functional* properties. However, hardware cost, energy usage, latency, and throughput can vary. Some platforms may have low cost, but high latency and energy requirements, while more expensive hardware may offer improved throughput or energy efficiency. In general, there is no single best product; instead, each developer must choose according to their optimization goals and design constraints. As such, these *performance attributes* – also known as *Non-Functional*

Properties (NFPs) – play a vital role in the configuration of software product lines for embedded platforms.

While configuring an optimal product may be easy for this toy example, it is much less so in real-world product lines with hundreds or even thousands of features. Here, *models* that predict how individual features affect the non-functional properties of individual products come into play. They can be used to predict the effect of features at configuration time, and to automatically configure features that are not relevant for functional product properties [Ola+12; SSS10]. Preferably, these models should be accurate, easy to understand, and require a minimal amount of manual intervention during training.

This chapter gives an overview of methods and definitions for each of these aspects:

- feature models for software product lines,
- performance models for non-functional product properties,
- variability modeling languages that express feature models as text, with a special focus on the Kconfig language,
- machine learning methods for performance model generation, and
- quality metrics that assess model accuracy and interpretability.

The operating system product line introduced in the previous paragraphs will serve as a running example throughout the chapter. At the end, I will present the real-world software product lines that serve as evaluation targets in this thesis.

2.1 FEATURE MODELS

As the introductory example shows, product lines must be configured into concrete products before they can be used. With an appropriate configuration interface, this is easy to achieve – it can be as simple as using conditional blocks in Makefiles and source code, and setting compile-time variables and pre-processor flags to indicate the target architecture. For instance, a user could run `make arch=msp430` to compile an operating system product for the MSP430 architecture. This, in turn, would pass the `-DARCH_MSP430` flag to the compiler, causing it to include architecture-specific code that is guarded by `#ifdef ARCH_MSP430` statements and leave out code that is specific to other architectures.

When dealing with real-world software product lines, such as the busybox multi-call binary[1] or the Linux kernel, such a simple approach

1 In busybox documentation, the term *multi-call binary* describes a single binary that provides distinct applets depending on how it is called. For instance, `busybox ls` behaves like the `ls` utility, and `busybox telnet` provides a `telnet` client. Users define the set of supported applets at compile time.

is no longer sufficient. These expose thousands of features with complex dependencies, allowing them to be used in – and optimized for – a wide range of use cases. Manual feature and dependency management is not feasible at this scale. Instead, they use a formal *feature model* that defines and structures the features within the SPL, where each *feature* is a product characteristic such as support for specific architectures or device drivers [Kan+90]. Combined with a configuration frontend that can read the feature model and save individual configurations, this allows users to define and build valid configurations [Ape+13].

A feature model breaks down a product line's variability into a hierarchy of features [Kan+90]. *Abstract* features group sub-features but do not provide any product characteristic by themselves, whereas *concrete* features can be enabled or disabled and affect the resulting product. Individual features are either *mandatory* (they must be enabled if the parent is enabled) or *optional* (they can be enabled if the parent is enabled). In both cases, a sub-feature cannot be enabled if the parent feature is disabled. Each group of sub-features has one of three relations:

- *and*: each mandatory feature must be selected (this is the default);
- *alternative* (exclusive or): exactly one feature must be selected;
- *or* (inclusive or): at least one feature must be selected.

Product line engineers can also specify more complex cardinalities, such as the minimum and maximum number of enabled sub-features. By default, concrete features are boolean, i.e., they can be disabled or enabled. Users may also specify *numeric* features that take a number from the domain $\mathbb{N}$ or $\mathbb{R}$ (depending on application), and *string* features that take arbitrary user-provided data (e.g. interface names).

A feature model can be limited to functional product characteristics, or include implementation details such as different algorithms that achieve the same function from an end-user perspective, but have different non-functional properties [CHE05]. The latter allows stakeholders to compare and optimize performance attributes of configurations that fulfil the same functional requirements. Product line engineers may also design a hierarchy of feature models to separate functional aspects and implementation details [Ros+11]. Feature models in this thesis contain both kinds of features – however, the findings also apply to disaggregated models.

When using a feature model to configure an SPL, each product is defined by a unique configuration of concrete features, or in formal terms: a *feature vector* that maps each feature to a value [Nai+20]. In the simplest case, where features can only be disabled or enabled, products of a product line with n features can be unambiguously described with feature vectors from the set $\{0,1\}^n$. In general, the domain may also include numeric and string entries, but the concept of a feature vector remains the same.

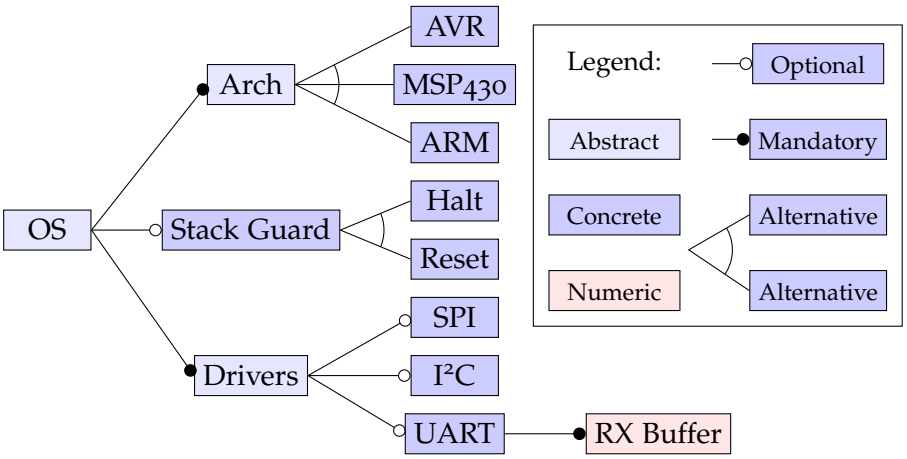


Figure 2.1: Feature model for a sample operating system product line.

Fig. 2.1 shows the feature model of the OS product line that serves as running example in this chapter, using the established visualization method [Kan+90]. The *OS* root feature is abstract, so it cannot be enabled or disabled individually. It defines that an operating system product consists of the architecture it is compiled for, peripheral interface drivers, and an optional stack guard feature. Architecture and drivers are abstract as well.

As each product only supports a single architecture, and compiling an operating system without underlying hardware is not sensible in this case, the mandatory *Arch* feature indicates that exactly one of its sub-features must be selected: AVR, MSP430, or ARM. These serve as short placeholders for specific microcontroller types, such as AVR ATMega2560 or MSP430FR5994.

The optional *Stack Guard* feature uses canary values to detect stack overflows. If enabled, it can either halt the operating system once it has detected a stack overflow or trigger a hardware reset; hence, *Halt* and *Reset* are alternative features. Due to the implicit dependency relation imposed by the tree structure, Halt and Reset can only be enabled if Stack Guard is enabled.

The *SPI*, *I²C* and *UART* drivers can be enabled and disabled individually. SPI and I²C use application-provided buffers, whereas the UART driver provides an OS-managed buffer for incoming transmissions. Therefore, if the UART driver is enabled, users must configure RX buffer size.

Product line engineers can extend the feature model with explicit dependencies, also known as cross-tree constraints. For example, an engineer may limit RX buffer size depending on the selected architecture to ensure that sufficient RAM is available for other tasks: $\text{RX Buffer} \leq 256 \wedge (\text{AVR} \Rightarrow \text{RX Buffer} \leq 64)$. Or, due to limited processing resources, they may specify that Stack Guard is not available on ATMega: $\text{ATMega} \Rightarrow \neg\text{Stack Guard}$.

These constraints ensure that any valid configuration results in a working product. The feature vector encodes configurations as follows.

$$\vec{x} = (x_{\text{AVR}}, x_{\text{MSP430}}, x_{\text{ARM}}, x_{\text{Stack Guard}}, x_{\text{Halt}}, x_{\text{Reset}}, x_{\text{SPI}}, x_{\text{I²C}}, x_{\text{UART}}, x_{\text{RX Buffer}}) \in \{0,1\}^9 \times \mathbb{N}^1$$

However, there is no description of non-functional product attributes, and thus users without domain knowledge cannot aim for low cost or resource usage when configuring products. Performance models address just that.

2.2 PERFORMANCE MODELS

Mathematically speaking, a *performance model* (or *NFP model*) is a function $f : \vec{x} \mapsto y$ that predicts a performance attribute (non-functional property, NFP) y of a product line configuration $\vec{x}$. For instance, it may predict the size of a kernel image or busybox binary from a `.config` file, the throughput of a video encoder from its command-line configuration, the latency of a neural network from its layout, or operating system size from the feature vector shown above.

For the operating system example, assume that the developer is interested in hardware cost (€) and binary size (kB). The former is relevant for the sales department, and the latter dictates how much space is available for applications and whether the operating system product fits onto its target platform in the first place.

Performance models can be designed and interpreted manually by a domain expert, generated automatically by a machine learning algorithm and interpreted by performance-aware configuration software, or built and used with a combination of both. This section only covers manual specification and interpretation using feature- and variant-wise annotation – automation methods follow in Section 2.5.

Feature-Wise Annotation (FW) is one of the simplest methods for performance modeling. It associates each (boolean) feature with a static performance attribute, and assumes that the performance attribute of the entire product is the sum of performance attributes of enabled features. The model can be created manually by a domain expert or automatically via machine learning. This section showcases the manual method by providing fictional hardware cost and binary size values for the example operating system product line. In a real-world use case, practitioners would obtain these using a combination of domain knowledge and benchmarks.

As the name suggests, hardware cost is dictated by the hardware architecture the product runs on, and is independent of Stack Guard and driver configuration. Thus, annotating each Arch sub-feature with the bulk price of the respective hardware platform (say, 2 € for AVR, 4 € for MSP430, and 3 € for ARM) is sufficient and the hardware cost model is complete.

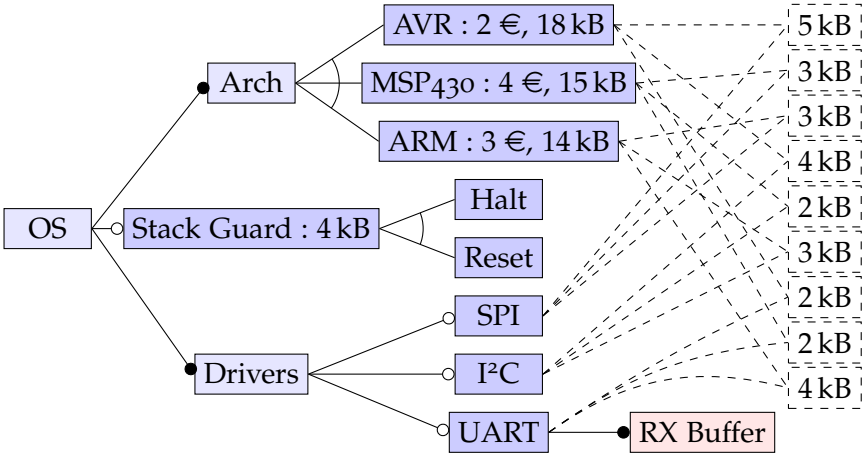


Figure 2.2: Feature model for a sample operating system product line, using feature- and variant-wise annotations to model hardware cost and kernel size.

We can start with the same approach for binary size. As each product is built for exactly one architecture, product line engineers can annotate each Arch sub-feature with the combined size of common and architecture-specific operating system components, excluding drivers and stack guard code: 18 kB for AVR, 15 kB for MSP430, 14 kB for ARM.

The stack guard feature is largely architecture-independent, and can also be annotated this way. When rounding to whole kilobytes, it increases binary size by 4 kB regardless of halt/reset configuration.

Drivers, on the other hand, work with architecture-specific peripheral interfaces. While the SPI driver introduces just 3 kB on MSP430 and ARM, it increases total size by 5 kB on AVR, so a simple annotation is not sufficient. This phenomenon is known as *feature interaction*, and it is becoming more and more common in today's increasingly complex software and ever-expanding feature models [Tër+22]. Prominent examples are cross-cutting concerns such as debug flags, optimization options, security features, and the hardware selection shown here.

Variant-wise annotations address this by considering interactions between related features [Sie+12b]. They can express performance attributes that are only relevant if two or more features are enabled. For instance, assuming that SPI takes up 3 kB on MSP430 and ARM, and 5 kB on AVR, a product line engineer will annotate the pairs (MSP430, SPI) and (ARM, SPI) with 3 kB, and (AVR, SPI) with 5 kB. They can do the same for I²C and UART. As the RX buffer is located in RAM, it does not affect binary size. Fig. 2.2 shows the resulting *feature interaction model*.

This combined variability and performance model allows for performance-aware configuration of the operating system product line. Anyone can predict the cost and size of arbitrary products within the product line by calculating the sum of all feature- and variant-wise

annotations of enabled features, and also see how each feature affects them. For example, if a user is looking for an I²C-enabled kernel with minimal footprint, they immediately see that both MSP430 and ARM result in a 16 kB binary, whereas the AVR variant would take up 22 kB. The model also shows that ARM is cheaper than MSP430, so they may be inclined to use an ARM processor for this task.

While this works well in a toy example with just nine features and nine interactions, it turns into a significant engineering effort when applied to real-world product lines with hundreds to thousands of features and interactions. Machine learning offers an alternative approach for these cases: run benchmarks for different configurations and automatically generate a performance model using a suitable machine learning algorithm. Doing this at scale relies on the ability to extract configurations and feature vectors from variability models, run automated benchmark campaigns, and feed observations into model learning algorithms. This, in turn, benefits from machine-readable variability models and well-defined variability modeling languages.

2.3 VARIABILITY MODELING LANGUAGES

From a language design point of view, the previous sections introduced variability models in a deliberately informal manner. Before moving on to textual models and actual variability modeling languages, let us take a step back and examine how these relate to the expressiveness and graphical representation of variability models.

Variability modeling languages (and domain-specific languages in general) distinguish between abstract syntax and concrete syntax [Don+84; BV04]. Abstract syntax defines structure and expressiveness – in this case, feature hierarchies, feature types, and cardinalities. Concrete syntax defines the representation of language components – for instance the tree structure shown in Fig. 2.1.

A specific *variability modeling language* builds upon an abstract syntax that defines its features and expressiveness, e.g. whether it supports numeric features or complex cardinalities. In addition, it typically has at least one concrete syntax that defines the representation of feature models, for instance in a graphical or textual manner. Textual variability models are machine-readable by design. When created with appropriate tooling rather than printed in a document, graphical variability models are machine-readable as well and can be transformed into textual models or other representations, and vice versa.

Considering the relevance of machine-readability in this thesis, we will now examine existing variability modeling languages (also known as *feature modeling languages*) with a focus on their expressiveness and textual representation. In addition to expressiveness, the language choice also dictates the available options for performance prediction models, tooling support, and more [BSE19].

Variability modeling languages have been an active research subject for the past two decades [ES15]. Early approaches, such as the Feature Description Language, define text-based representations that build upon already-existing graphical representations [VK02]. They are limited to boolean features, relations, and constraints. While this allows for reasoning about any variability model by means of propositional calculus, the designers of the Feature Description Language already noted that numeric features and relations may be helpful extensions.

Forfamel and VSL, published only a few years later, support numeric feature attributes such as RX buffer size [AMS06; Abe+10]. In fact, Forfamel handles arbitrary attribute types, including strings and numeric attributes with optional range limits. Its authors also suggest using a constraint language to express dependencies on feature attributes rather than just features. However, in the provided concepts and implementations, all attributes are purely informative. They can be configured by the user and read out by software, but cannot be reasoned upon. For example, they cannot express the RX Buffer $\leq 256 \wedge (\text{AVR} \Rightarrow \text{RX Buffer} \leq 64)$ constraint from Section 2.1.

Still, this provides an early method for augmenting features with performance attributes. By specifying constant feature attributes such as hardware cost or binary size, and assuming that the cost or size of a product is the sum of the cost or size of all selected features, practitioners and algorithms can predict overall product cost or size. This is a hack, though: feature attributes were not meant for this task, and there are neither syntax nor semantic rules that properly define the aggregation of non-functional properties.

TVL and VELVET introduce just this: aggregate functions for feature attributes [Bou+10; Ros+11; Sie+11]. For instance, the following TVL expression declares that a parent feature's text segment size is the sum of the text segment size of all enabled child features:

```
int textSize is sum(selectedChildren.textSize);
```

Thus, practitioners can define product performance based on non-functional properties, and reason about optimal configurations. However, they can only do so for performance properties that are a simple aggregation of feature-specific values – i.e., the contribution of each feature is independent of the configuration of other features. As we have already seen, this is not always the case.

Hence, modern variability modeling approaches like Clafer/ClaferMoo and SPL Conqueror include methods to specify feature interaction [BCW10; Bąk+16; Ant+13; Sie+12b; Sie+12a]. For example, in Clafer, engineers may specify that enabling the Debug feature increases total binary size by 10 % on average:

```
binarySize = (sum Feature.binarySize) * (Debug ? 1.1 : 1)
```

Listing 2.1: UVL feature model for a sample operating system product line.

```
features
  OS {abstract}
    mandatory
      Arch {abstract}
        alternative
          AVR
          MSP430
          ARM
      Drivers {abstract}
        optional
          SPI
          I2C
          UART
    optional
      StackGuard
        alternative
          Halt
          Reset

constraints
  StackGuard => MSP430 | ARM
```

One of the most recent variability modeling language specimen is the Universal Variability Modeling Language (UVL). It is part of an initiative that aims to find a community consensus on best practices for variability modeling, and to define a corresponding universal variability modeling language [Sun+21b]. In this spirit, there is support for interactive generation and usage of UVL models [Sun+21a], and for conversion to and from other variability modeling languages [Rom+22].

As part of this initiative, practitioners expressed their priorities for variability modeling language features in questionnaires. For example, seven of 20 participants indicated that feature attributes are nice to have, and two deemed them as absolute necessities [Sun+21b].

As of late 2022, a first formal definition of UVL exists, and the authors state that it may be refined in the future. This UVL version does not support feature attributes by itself, but there is an experimental superset of UVL with feature attribute support. So, even after two decades of research, choosing a suitable variability modeling language is far from trivial.

After this brief history lesson, let us examine how two recent textual variability modeling languages express the OS feature model defined in Fig. 2.2. Listing 2.1 contains a UVL model for this operating system product line example. Its nested definitions closely resemble the tree structure of Fig. 2.1. As UVL does not (yet) support numeric features, the model does not contain RX buffer size and the associated constraint.

Listing 2.2 shows a Clafer model for the same product line, including performance attributes using feature- and variant-wise annotations.

Listing 2.2: Clafer feature model for a sample operating system product line, annotated with cost and size attributes.

```
abstract Feature
  cost : integer
  size : integer

abstract OS
  xor Arch
    AVR    : Feature [ (cost = 2), (size = 18) ]
    MSP430 : Feature [ (cost = 4), (size = 15) ]
    ARM    : Feature [ (cost = 3), (size = 14) ]
  Drivers
    SPI : Feature ?  [ size = AVR ? 5 : 3 ]
    I2C : Feature ?
      [ size = AVR ? 4 : MSP430 ? 2 : 3 ]
    UART : Feature ? [ size = ARM ? 4 : 2 ]
      RXBuffer : integer
   xor StackGuard : Feature ? [ size = 4 ]
     Halt : Feature
     Reset : Feature

total_cost : integer [ total_cost = sum Feature.cost ]
total_size : integer [ total_size = sum Feature.size ]

[ StackGuard  => MSP430 || ARM ]
[ UART        => RXBuffer <= 256 ]
[ UART && AVR => RXBuffer <= 64 ]
```

Here, too, there is a clear relation between textual variability model and tree structure. However, due to language-level support for abstract features, inheritance, and performance attributes, the textual representation is harder to digest.

My own research focuses on working with existing real-world product lines, or configurable software projects that behave similar to product lines. In general, these do not come with a feature model defined in UVL, Clafer, or a similar formal variability modeling language. Instead, they either do not have a formal variability model at all, or rely on the Kconfig variability modeling language. Hence, although the methods I will present in this thesis are applicable to any of the aforementioned variability modeling languages, the implementation builds upon Kconfig-based product lines and software projects.

2.4 THE KCONFIG LANGUAGE

Kconfig is far from an ideal variability modeling language. It is meant to solve the practical issue of Linux kernel configuration and has not been developed according to software engineering research principles [EKS15]. However, its wide-spread adoption in open-source software projects such as Linux or busybox makes it a suitable vari-

ability modeling language for evaluation of real-world applications. More so, even if projects like Linux are not developed according to the guidelines of *Software Product Line Engineering* (SPLE), they can be understood as software product lines [Sin+07]. Thus, building on top of Kconfig ensures compatibility with several ready-to-evaluate software projects. Using those as evaluation targets also helps ensure that the results presented in this thesis apply to real-world product lines.

The Kconfig ecosystem uses a `Kconfig` text file to define feature models. In contrast to UVL and other languages from the SPL community, a Kconfig model does not define a feature tree via nesting or references. While the `choice` and `menu` keywords allow engineers to define a group of features with a common parent, these are not mandatory. Instead, Kconfig configuration frontends infer a tree structure from feature dependencies as well as `choice` and `menu` groups. So, the tree structure is only present at configuration time, and only when using a suitable Kconfig frontend.

A Kconfig file is made up of `config` entries that define individual features. Each feature is either `bool` (yes / no, a boolean feature), `tristate` (yes / module / no, often used to decide whether features should be compiled into the kernel, into separate module files, or not at all), `int` / `hex` (numeric), or `string` (e.g. the path to the build toolchain). Features have a user-visible `prompt` and optional `help` text. Additionally, they can depend on boolean expressions that reference other features (`depends on`), reverse-depend on features (`select`/`imply`), have `default` values, and be limited to a numeric `range`.

Entries without a `prompt` are legal, but invisible in all configuration frontends and thus do not express features. Combined with reverse dependencies of (visible) features, these express common traits of individual features and can help reduce clutter in conditional source code blocks.

`menu` and `choice` entries define parts of the menu tree that users see when using a Kconfig frontend for configuration. A `menu` is an abstract group of related features, such as the Drivers node in the operating system example (Fig. 2.1). A `choice` offers an alternative between several child features in an exclusive-or relation. By default, it is abstract (and, thus, mandatory); when declared as `optional` it becomes a concrete feature that can be enabled and disabled.

Kconfig frontends such as kconfig-qconf or kconfiglib[2] store the user-provided configuration in a `.config` file. Each line corresponds to a single config entry; however, the `.config` file only provides values for entries whose dependencies are satisfied. Each user-configurable feature (and invisible entry) with satisfied dependencies has an associated value in the `.config` file, and features (and entries) whose

2 https://github.com/ulfalizer/Kconfiglib

dependencies are not met do not. Thus, even if Kconfig has not been designed with product line engineering in mind, extraction of feature vectors from `.config` files is viable.

Listing 2.3 shows a Kconfig definition of the feature model defined in Fig. 2.1. For details, please refer to the language specification[3]. Note that, just like UVL, Kconfig does not support performance attributes.

2.5 MACHINE LEARNING

As stated earlier, rather than relying on manual annotation by a domain expert, developers can also employ machine learning algorithms for automatic generation of performance models, or use a combination of machine learning and domain knowledge.

Of course, machine learning algorithms do not come up with performance models out of thin air. In order to be usable for this task, they must have access to training data consisting of product line configurations (feature vectors) $X = \{\vec{x}_1, \ldots, \vec{x}_m\}$ and performance attributes (e.g. binary size or processing throughput) $Y = \{y_1, \ldots, y_m\}$. Each pair $(\vec{x}_j, y_j)$ describes a configuration (benchmark input) $\vec{x}_j \in \mathbb{R}^n$ and the corresponding performance attribute (benchmark output) $y_j \in \mathbb{R}$. In cases where the index in the sets X and Y is not relevant, $(\vec{x}, y)$ refers to any pair of matching benchmark data. The variable x_i refers to the i-th feature vector element (the i-th configurable feature), regardless of its concrete value.

From X and Y, machine learning algorithms infer a model function $f : \vec{x} \to y$ that ideally behaves just like the manually specified performance prediction models examined earlier. Some algorithms require a user to provide the model structure (e.g. a function template or a neural network layout); others are capable of generating it by themselves and only require training data as input. In both cases, users may be able to configure training *hyper-parameters* that affect e.g. trade-offs between model accuracy and model complexity.

Machine learning is a popular research field, with applications as diverse as anomaly detection in time series, identification of people in images, or generating media from text prompts. My own work only touches a small part of that, namely learning to predict individual numeric performance attributes from numeric (and, in Chapter 7, categorical) feature vectors. While anomaly detection and computer vision will come into play later, they are not relevant for performance model generation, and therefore not within the scope of this chapter.

Note that I only consider machine learning methods and models that predict a single performance attribute. By combining several models (e.g. one for hardware cost and one for binary size), users can still obtain integrated performance models that predict multiple attributes, such as the one shown in Fig. 2.2. I will also leave out

3 https://kernel.org/doc/html/latest/kbuild/kconfig-language.html

Listing 2.3: Kconfig model for a sample operating system product line.

```
choice arch
bool "Architecture"

config avr
bool "AVR"

config msp430
bool "MSP430"

config arm
bool "ARM"

endchoice

choice stack_guard
bool "Stack Guard"
optional
depends on msp430 || arm

config guard_halt
bool "Halt System on Stack Overflow"

config guard_reset
bool "Reset System on Stack Overflow"

endchoice

menu "Drivers"

config spi
bool "SPI"

config i2c
bool "I2C"

config uart
bool "UART"

config uart_rx_buf
int "RX Buffer"
default 32
range 0 256
range 0 64 if avr
depends on uart

endmenu
```

string features – in my experience, these generally refer to attributes that do not influence system performance, such as compiler paths or component identifiers. Hence, $\vec{x} \in \mathbb{R}^n$ holds for all feature vectors in this chapter.

With these criteria in mind, two kinds of machine learning algorithms are especially interesting: least-squares regression and regression trees. Both predict a single output variable from numeric input variables and use model structures with limited complexity. Also, both have been used for SPL performance prediction in the past. Least-squares regression is closely related to feature- and variant-wise annotations; it relies on a fixed, user-provided model structure. Regression tree learning algorithms build the model structure from training data and expose hyper-parameters to limit its complexity.

As it relies on user-provided model templates, least-squares regression alone is not a suitable solution for RQ3 ("... without requiring manually provided domain information or model structure"). It is, however, useful as an evaluation target, and – as we will see in the next chapter – it is also possible to extend least-squares regression so that it learns the structure by itself.

The remainder of this section covers both learning algorithms in detail. It also presents a lookup table model that is not useful for performance prediction of configurations that were not present in training data, but will prove helpful when it comes to putting the complexity and accuracy of least-squares regression, regression forests, and other models into perspective.

2.5.1 *Least-Squares Regression*

Broadly speaking, regression analysis is a machine learning method that adjusts (*fits*) weights $\vec{\beta}$ in a function template $f(\vec{x})$ so that $f(\vec{x}) \approx y$ for training data pairs $(\vec{x}, y)$ [DS98]. It does not build the function template itself, but relies on users or pre-processing algorithms to provide it.

Regression analysis algorithms achieve this by minimizing the *loss* of the error term $\varepsilon = y - f(\vec{x})$. Loss represents prediction uncertainty and thus model error. In least-squares regression, it is the eponymous *Sum of Squared Residuals* (SSR) $\sum_i \varepsilon_i^2$.

$$SSR(f, X, Y) = \sum_{i=1}^{m} (y_i - f(\vec{x}_i))^2 \tag{2.1}$$

The method for adjusting $\vec{\beta}$ in the learning process depends on the regression algorithm in question and the function template. These also decide whether the loss of the fitted output function really is a global minimum in the $\left|\vec{\beta}\right|$-dimensional plane of functions $f(\vec{x})$. If it is a global minimum, the resulting weights $\vec{\beta}$ are guaranteed to

give optimal predictions with the provided function template on the available training data. Otherwise, there might be a different set of weights $\vec{\beta}$ that gives a better fit with lower model error.

Algorithms

The function template $f(\vec{x}) = \beta_0 + \sum_{i=1}^{n} \beta_i x_i$ is a special case that is also known as *Linear Regression*. Here, the function f and the loss function *SSR* are differentiable. By setting the partial derivatives of *SSR* to zero and solving for $\vec{\beta}$, anyone – including computer algorithms – can find weights that correspond to a global minimum of the loss *SSR*. Hence, in linear regression, the resulting weights $\vec{\beta}$ are guaranteed to provide an optimal fit on the provided training data.

Some non-linear function templates can be handled in the same manner. When working with arbitrary function templates, though, f and *SSR* may not be differentiable – or solving for $\vec{\beta}$ is not feasible due to an insufficient amount of available computing resources. In this case, a common approach is approximating $\vec{\beta}$ via gradient descent.

In a nutshell, given some values for $\vec{\beta}$, this method computes its gradient on $SSR(f, X, Y)$ – so, the direction of change that corresponds to the steepest increase of model loss. Adjusting $\vec{\beta}$ in the inverse direction decreases it and thus reduces loss. Afterwards, a gradient descent algorithm computes a new gradient using the updated $\vec{\beta}$ values, and continues until the algorithm converges (i.e., the loss reduction from adjusting $\vec{\beta}$ is below a user-defined threshold) or another stop criterion (e.g. maximum number of iterations, or change in $\vec{\beta}$ between consecutive iterations below threshold) has been reached.

While this works with arbitrary functions, it has limitations that users must be aware of. Most prominently, as gradient descent approximates the differential of the loss function and adjusts $\vec{\beta}$ in steps rather than working with a continuous and differentiable function, there is a risk of not converging within a limited time or ending up in a local rather than a global minimum of the loss function. It also requires users to declare initial values for $\vec{\beta}$, and may be non-deterministic e.g. if the initial $\vec{\beta}$ is random. In this thesis, the algorithm always starts with $\vec{\beta} = \vec{1}$.

Nevertheless, when users are aware of these limitations, least-squares regression can be a powerful machine learning algorithm. It has been used successfully for numerous tasks in the past, including performance models for software product lines with a custom pre-processing algorithm for function template generation [Sie+15].

Finally, *Symbolic Regression* is an extension that is capable of coming up with function templates on its own. It uses genetic algorithms to combine function snippets into a function template that balances prediction accuracy and function complexity. This enables it to, for instance, determine basic natural laws from empiric observations [SL09]. However, it is prone to overfitting: it tends to learn training data by

BENCHMARK #	1	2	3	4	5	6	7	8	9	10	11	12
x_1 (AVR)	1			1	1	1						
x_2 (MSP430)		1					1	1	1			
x_3 (ARM)			1							1	1	1
x_4 (Halt)												
x_5 (Reset)												
x_6 (SPI)				1			1			1		
x_7 (I²C)					1			1			1	
x_8 (UART)						1			1			1
y (Size [kB])	18	15	14	23	22	20	18	17	17	17	17	18

Table 2.1: Excerpt of configurations and corresponding kernel sizes for a sample operating system product line. Each row describes a configuration variable x_i or performance attribute y. Empty cells indicate disabled features (0). Configurations 13 through 24 (not shown) correspond to 1 through 12 with Halt enabled and a size of $y + 4$. Configurations 25 through 36 (not shown) correspond to 1 through 12 with Reset enabled and a size of $y + 4$.

heart rather than expressing the underlying behaviour [Ray+19]. This can already happen in low-dimensional configuration spaces and is exacerbated by noisy data [FBS18]. Considering these limitations, and that I am not aware of researchers applying symbolic regression to performance model generation, I will not go into further detail here.

Example

Assume that we have benchmarked our sample operating system product line and obtained the results shown in Table 2.1. Note that this is a deliberately simplified benchmark example that is only meant to illustrate the relation between least-squares regression, performance models, and feature- and variant-wise annotation. The benchmark results are not suitable in practice, as they do not contain configurations where more than one driver is enabled. Thus, learning algorithms cannot determine whether driver features interact with each other e.g. due to shared code.

As noted earlier, the function template must be provided by the user or a pre-processing algorithm. In this case, we use a simple pre-processing algorithm which assumes that each boolean feature contributes to the modeled performance attribute, and that there are no constant components and no feature interaction. So, given eight boolean features x_1 to x_8, the function template is $f(\vec{x}) = \sum_{i=1}^{8} \beta_i x_i$.

After fitting, e.g. using Python3's `scipy.optimize.least_squares`, and rounding $\vec{\beta}$ to kilobytes, the result is $\vec{\beta} = (18, 14, 14, 4, 4, 4, 3, 3)$. When transformed to feature-wise annotations, this means that AVR, MSP430 and ARM result in a base kernel size of 18, 14, and 14 kB, respectively. Halt and Reset contribute 4 kB each, and the SPI, I²C and UART features contribute 4, 3, and 3 kB.

This does not align with the observations shown in Table 2.1. Common reasons for this kind of model error are noisy data and unsuitable function templates. In this case, it is the latter: as we have already seen in Fig. 2.2, there are feature interactions between architecture selection and driver code, so a pure feature-wise annotation template is insufficient.

So, let us assume that a more sophisticated pre-processing algorithm has identified feature interactions in the product line, and generated the function template $f(\vec{x}) = \sum_{i=1}^{5} \beta_i x_i + \sum_{i=1}^{3} \sum_{j=6}^{8} \beta_{i,j} x_i x_j$. Now, β_i corresponds to feature-wise attributes, and $\beta_{i,j}$ to the interaction between features x_i and x_j. After fitting this function, $\vec{\beta}$ is nearly identical to the model shown in Fig. 2.2. The only difference is that $\vec{\beta}$ annotates Halt and Reset with 4 kB each instead of the parent Stack Guard feature – which is not surprising, as Table 2.1 does not contain a feature vector component for Stack Guard.

Limitations

The lack of Stack Guard is a deliberate omission that is motivated by an important limitation of least-squares regression analysis: the configuration options that make up $\vec{x}$ must be independent variables. If this is not the case, e.g. because a pair of features is not independent, the model may fail to converge or the fitted weights may be bogus.

Assume that we had introduced a feature x_9 and weight β_9 for the Stack Guard feature, so $f(\vec{x}) = \beta_1 x_1 + \cdots + \beta_8 x_8 + \beta_9 x_9$. The variables (x_4, x_5, x_9) are not independent due to $x_4 \vee x_5 \Rightarrow x_9$ and $x_9 \Rightarrow x_4 \veebar x_5$, so $x_9 = x_4 + x_5$. Attempting to fit such a function template may not provide a useful model. Although a least-squares algorithm may come up with the expected result of $\beta_4 = 0, \beta_5 = 0, \beta_9 = 4$ when fitting the model, it might just as well output $\beta_4 = 14, \beta_5 = 14, \beta_9 = -10$ or any other set of weights that satisfies the following equations.

$$\begin{aligned} \beta_4 x_4 + \beta_5 x_5 + \beta_9 (x_4 + x_5) &= 0 && \text{if } x_4 = 0 \wedge x_5 = 0 \\ \beta_4 x_4 + \beta_5 x_5 + \beta_9 (x_4 + x_5) &= 4 && \text{if } x_4 = 1 \veebar x_5 = 1 \end{aligned}$$

The first one is always true. The second one can be transformed as follows.

$$\beta_4 + \beta_9 = 4$$
$$\beta_5 + \beta_9 = 4$$

As this is a system of two equations and three free variables, it has an infinite number of solutions.

So, least-squares regression is not a hands-off approach. Users or pre-preprocessing algorithms must provide a suitable template function, and they must leave out configuration variables (features) that depend on other variables.

2.5.2 *Regression Trees*

In contrast to least-squares regression, decision tree-based machine learning algorithms come up with the model structure by themselves. Users need only provide data and optional training hyper-parameters for maximum tree height and other accuracy versus complexity trade-offs. In some cases, they must also ensure that input variables are pair-wise independent; however, this is not a general requirement.

A decision tree is a binary tree in which each non-leaf node corresponds to a boolean condition such as "is it raining?" or "$x_2 \leq 3$". Leaf nodes correspond to output values such as "bring an umbrella" or "$y = 23$". Decision trees that predict non-numeric classes (e.g. security attributes) are also referred to as classification trees. Decision trees that predict numeric values are also known as regression trees. This section presents common data structures and learning algorithms that will also serve as related work for evaluation purposes later on.

Classification and Regression Trees

Classification and Regression Trees (CART) and the CART learning algorithm are one of the oldest decision tree-based machine learning methods, dating back to 1984 [Bre+84]. As the name suggests, CART can be used both for classification (categorical output variable y) and regression ($y \in \mathbb{R}$). This thesis only uses the regression part, hence the following definitions apply.

Definition 2.5.1 A CART is a binary tree that expresses a piecewise constant function $f : \mathbb{R}^n \to \mathbb{R}$. Each non-leaf node holds a decision "$x_i \leq z$" for some index $i \in \{1, \ldots, n\}$ and threshold $z \in \mathbb{R}$, and each leaf node holds an output value $y \in \mathbb{R}$.

Definition 2.5.2 For a CART f, $\textsc{decision}(f) = \langle i, z \rangle$ indicates that its root node holds the decision "$x_i \leq z$". $\textsc{child}(f, \mathrm{y}) = f'$ refers to the sub-tree f' for $x_i \leq z$, and $\textsc{child}(f, \mathrm{n}) = f''$ refers to the sub-tree f'' for $x_i > z$. If the root node is a leaf node annotated with the value y, $\textsc{decision}(f) = \bot$ and $\textsc{value}(f) = y$.

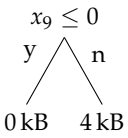


Figure 2.3: CART model for predicting the influence of Stack Guard (x_9) on operating system kernel size. Each leaf holds a prediction value for the corresponding system configuration.

For example, Fig. 2.3 shows a CART model f that predicts the influence of Stack Guard (x_9) on kernel size, ignoring all other features for the sake of simplicity. In this case, $\text{DECISION}(f) = \langle 1, 0 \rangle$, $\text{VALUE}(\text{CHILD}(f, \text{y})) = 0$, and $\text{VALUE}(\text{CHILD}(f, \text{n})) = 4$.

The learning algorithm builds the tree structure in a top-down manner by greedily adding decision nodes that minimize the loss function, which is again the SSR (see Equation 2.1). This way, it recursively refines the tree until a stop criterion has been reached.

As usual, model generation starts out with a set of n-dimensional feature vectors $X = \{\vec{x}_1, \ldots, \vec{x}_m\}$ and corresponding performance attributes (benchmark results) $Y = \{y_1, \ldots, y_m\}$. To simplify the syntax for splitting observations and feature vectors, let the set S contain pairs $(\vec{x}_j, y_j)$ of feature vectors and corresponding observations, and define the following shortcuts.

$$S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\} \tag{2.2}$$

arithmetic mean
$$\mu(S) = \frac{1}{m} \sum_{i=1}^{m} y_i \tag{2.3}$$

standard deviation
$$\sigma(S) = \sqrt{\frac{1}{m} \sum_{i=1}^{m} (y_i - \mu(S))^2} \tag{2.4}$$

loss function
$$SSR(f, S) = \sum_{i=1}^{m} (y_i - f(\vec{x}_i))^2 \tag{2.5}$$

unique values of x_i
$$Val_i(S) = \{x_i \mid (\vec{x}, y) \in S\} \tag{2.6}$$

With these shortcuts, the CART learning algorithm works as defined in Algorithm 1 [Bre+84]. The following list outlines the model generation steps in a less formal but otherwise identical manner.

1. If a stop criterion is satisfied: return a leaf node using the mean of observed data $\mu(S)$ as model value. Common stop criteria are:
   a) $\forall \vec{x}, \vec{x}' \in S : \vec{x} = \vec{x}'$ (there is nothing left to decide on), or
   b) tree depth has exceeded a user-provided threshold T_d, or
   c) $|S| < T_m$ for a user-provided threshold T_m, or
   d) $\sigma(S) < T_\sigma$ for a user-provided threshold T_σ.

Algorithm 1 Build a CART from observations S.

function BUILDCART(S)
  $f \leftarrow$ new CART
  if stop criterion satisfied **then**
    DECISION(f) $\leftarrow \bot$
    VALUE(f) $\leftarrow \mu(S)$
    return f
  for $i \in \{1, \ldots, n\}$ **do**
    for $z \in Val_i(S)$ **do**
      $S_{i,z,\mathrm{y}} \leftarrow \{(\vec{x}, y) \in S \mid x_i \leq z\}$
      $S_{i,z,\mathrm{n}} \leftarrow \{(\vec{x}, y) \in S \mid x_i > z\}$
      $SSR_{i,z} \leftarrow SSR(\vec{x} \mapsto \mu(S_{i,z,\mathrm{y}}), S_{i,z,\mathrm{y}})$
        $+ SSR(\vec{x} \mapsto \mu(S_{i,z,\mathrm{n}}), S_{i,z,\mathrm{n}})$
  $\langle i, z \rangle \leftarrow$ **argmin**$(\langle i, z \rangle, SSR_{i,z})$
  DECISION(f) $\leftarrow \langle i, z \rangle$
  CHILD(f, y) $\leftarrow$ BUILDCART($S_{i,z,\mathrm{y}}$)
  CHILD(f, n) $\leftarrow$ BUILDCART($S_{i,z,\mathrm{n}}$)
  return f

2. For each feature x_i, and each unique value z of x_i in S: Split S into partitions $S_{i,z,\mathrm{y}}$ and $S_{i,z,\mathrm{n}}$ so that $S_{i,z,\mathrm{y}}$ only contains entries with $x_i \leq z$ and $S_{i,z,\mathrm{n}}$ only contains entries with $x_i > z$, and calculate the model error incurred by splitting S on "$x_i \leq z$": $SSR_{i,z} = SSR(\vec{x} \mapsto \mu(S_{i,z,\mathrm{y}}), S_{i,z,\mathrm{y}}) + SSR(\vec{x} \mapsto \mu(S_{i,z,\mathrm{n}}), S_{i,z,\mathrm{n}})$.
3. Find the pair x_i, z with the lowest loss $SSR_{i,z}$ and transform it into a decision node "$x_i \leq z$".
4. Repeat recursively with $S_{i,z,\mathrm{y}}$ (left child) and $S_{i,z,\mathrm{n}}$ (right child).

The user-provided hyper-parameters T_d, T_m and T_σ affect the trade-off between model accuracy and model complexity. Querying a CART (i.e., calculating $f(\vec{x})$) consists of simply following the decision nodes from the root until reaching the leaf node that contains the function output. Algorithm 2 gives a formal definition of this. While one might argue that CART are sufficiently straightforward to be understandable without formal algorithm definitions, we will examine more complex decision tree learning algorithms later on. Understanding how those work and how they relate to CART will benefit from using a common formal framework right from the start.

In contrast to least-squares regression, CART do not associate individual features or feature pairs with weights. Instead, each leaf node (i.e., each path through the tree) defines a sub-set of system configurations and associates it with a constant performance value. This goes with the implicit assumption that all configuration variables that are not part of the path from leaf to root do not affect the predicted performance value – otherwise, the learning algorithm would

Algorithm 2 Calculate $f(\vec{x})$ for a CART f.

```
function QueryCART(f, x⃗)
    if decision(f) = ⊥ then
        return value(f)
    ⟨i, z⟩ ← decision(f)
    if x_i ≤ z then
        return QueryCART(child(f, y), x⃗)
    else
        return QueryCART(child(f, n), x⃗)
```

have generated decision nodes for them. It is similar to variant-wise annotation, but with variable-sized feature sets rather than just pairs of features.

The greedy nature of the learning algorithm also means that influential features end up in decision nodes close to the root, whereas less performance-relevant features end up close to leaf nodes or do not become part of the decision tree at all. So, examining the tree structure allows users to draw conclusions about the importance of individual features. Before looking into an example for this, let us first examine another version of the CART algorithm.

Data-Efficient Classification and Regression Trees

If all features are boolean, CART can be simplified to *Data-Efficient Classification and Regression Trees* (DECART) [Guo+18]. Instead of a piecewise constant function $\mathbb{R}^n \to \mathbb{R}$, these express a piecewise constant function $\{0,1\}^n \to \mathbb{R}$. This simplifies decision nodes from threshold comparisons to "is feature i enabled?"

In principle, the CART learning algorithm (Algorithm 1) applies to DECART unchanged. Finding the optimal split automatically becomes simpler as there cannot be more than two unique values for each feature x_i, so there is just one sensible split per feature: "$x_i \leq 0$?". However, this leads to $S_{i,0,\mathrm{y}}$ containing entries with x_i disabled and $S_{i,0,\mathrm{n}}$ containing entries with x_i enabled, which counters the intuition of "y" for enabled features and "n" for disabled features.

Hence, this thesis uses a DECART learning algorithm with partitions $S_{i,\mathrm{y}}$ ($x_i = 1$) and $S_{i,\mathrm{n}}$ ($x_i = 0$). The generated decision node asks "Feature i enabled?" rather than "$x_i \leq 0$"; the left child covers $S_{i,\mathrm{n}}$ and the right child covers $S_{i,\mathrm{y}}$. The following definitions apply.

Definition 2.5.3 A DECART is a binary tree that expresses a piecewise constant function $f : \{0,1\}^n \to \mathbb{R}$. Each non-leaf node holds a decision "x_i?" for some index $i \in \{1, \ldots, n\}$, and each leaf node holds an output value $y \in \mathbb{R}$.

Definition 2.5.4 For a DECART f, $\textsc{decision}(f) = \langle i, \bot \rangle$ indicates that its root node holds the decision "x_i?". $\textsc{child}(f, \mathrm{y}) = f'$ refers to the

Algorithm 3 Build a DECART from observations S.

function BUILDDECART(S)
  $f \leftarrow$ new DECART
  if stop criterion satisfied **then**
    $\textsc{decision}(f) \leftarrow \bot$
    $\textsc{value}(f) \leftarrow \mu(S)$
    return f
  for $i \in \{1, \ldots, n\}$ **do**
    $S_{i,\mathrm{y}} \leftarrow \{(\vec{x}, y) \in S \mid x_i = 1\}$
    $S_{i,\mathrm{n}} \leftarrow \{(\vec{x}, y) \in S \mid x_i = 0\}$
    $SSR_i \leftarrow SSR(\vec{x} \mapsto \mu(S_{i,\mathrm{y}}), S_{i,\mathrm{y}})$
      $+ SSR(\vec{x} \mapsto \mu(S_{i,\mathrm{n}}), S_{i,\mathrm{n}})$
  $i \leftarrow \mathbf{argmin}(i, SSR_i)$
  $\textsc{decision}(f) \leftarrow \langle i, \bot \rangle$
  $\textsc{child}(f, \mathrm{y}) \leftarrow$ BUILDDECART($S_{i,\mathrm{y}}$)
  $\textsc{child}(f, \mathrm{n}) \leftarrow$ BUILDDECART($S_{i,\mathrm{n}}$)
  return f

Algorithm 4 Calculate $f(\vec{x})$ for a DECART f.

function QUERYDECART($f, \vec{x}$)
  if $\textsc{decision}(f) = \bot$ **then**
    return $\textsc{value}(f)$
  $\langle i, \bot \rangle \leftarrow \textsc{decision}(f)$
  if $x_i = 1$ **then**
    return QUERYDECART($\textsc{child}(f, \mathrm{y}), \vec{x}$)
  else
    return QUERYDECART($\textsc{child}(f, \mathrm{n}), \vec{x}$)

sub-tree f' for $x_i = 1$ (feature enabled), and $\textsc{child}(f, \mathrm{n}) = f''$ refers to the sub-tree f'' for $x_i = 0$ (feature disabled). If the root node is a leaf node annotated with the value y, $\textsc{decision}(f) = \bot$ and $\textsc{value}(f) = y$.

Algorithms 3 and 4 show the DECART learning and query algorithms. A comparison to CART (Algorithms 1 and 2) confirms that a boolean-only configuration space leads to simpler algorithms.

A notable advantage of DECART is that a small amount of random samples is often sufficient for acceptable model accuracy. For instance, Guo et al. mention less than 10 % model error when using just $10 \cdot n$ random samples for product lines with n boolean features [Guo+18].

Fig. 2.4 shows a DECART model that has been generated by the Python3 `scikit-learn` DecisionTreeRegressor module from a subset of the benchmark results shown in Table 2.1. To make sure that the resulting tree fits onto a single page, the subset only contains observations 1 through 12; Stack Guard is always disabled.

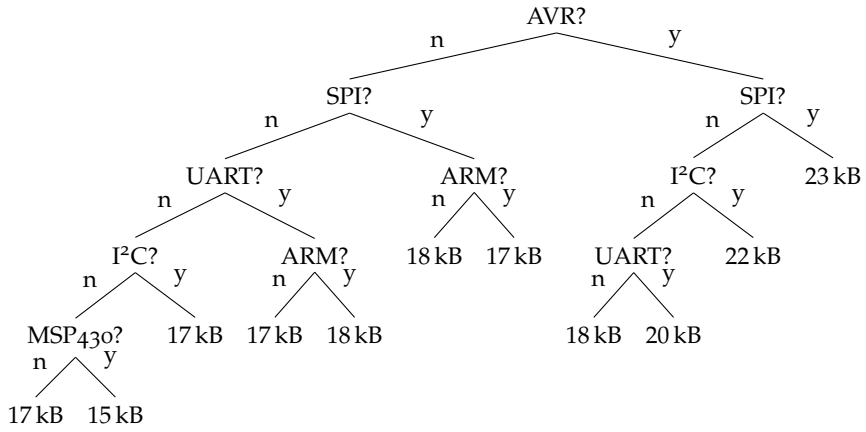


Figure 2.4: DECART model for predicting the kernel size of a sample operating system product line. Each leaf holds a prediction value for the corresponding system configuration.

Even such a simple model can already contain helpful information. For example, the path no AVR → no SPI → no UART → I²C indicates that it does not matter whether the I²C driver is used on an MSP430 or ARM platform, at least when it comes to binary size. As Fig. 2.1 shows, this is correct.

The top decisions are AVR and SPI, which makes sense: ARM and MSP430 have a similar binary size footprint, whereas AVR is larger; and the SPI driver is, on average, also the largest one. So, the decision tree has indeed placed more influential features closer to the top.

Again, this specific example is deliberately simplified and not useful for actual prediction of system performance. The input to the learning algorithm does not contain samples with Stack Guard enabled, and it also does not contain samples in which more than one peripheral driver is enabled. So, the learning algorithm had no opportunity to observe system behaviour with multiple drivers enabled, and as a consequence of that the regression tree cannot properly predict those cases. For example, the (incorrectly) predicted kernel size for AVR with SPI, I²C, and UART drivers enabled is the same as the (correctly) predicted kernel size for AVR with SPI only (23 kB), even though the size of these configurations differs by 6 kB.

Note that, when using a pre-processing algorithm for function template generation rather than manually providing one, this issue would also have appeared in the least-squares regression example in the previous section. As usual, a machine learning algorithm rarely has a chance of being better than its training data. If the input is garbage, the output will likely be garbage as well.

In general, if practitioners desire a maximum amount of automation and do not want to provide domain knowledge, they must ensure that they use a suitable subset of the product line's configuration space for training. In this small example, they could simply perform an exhaus-

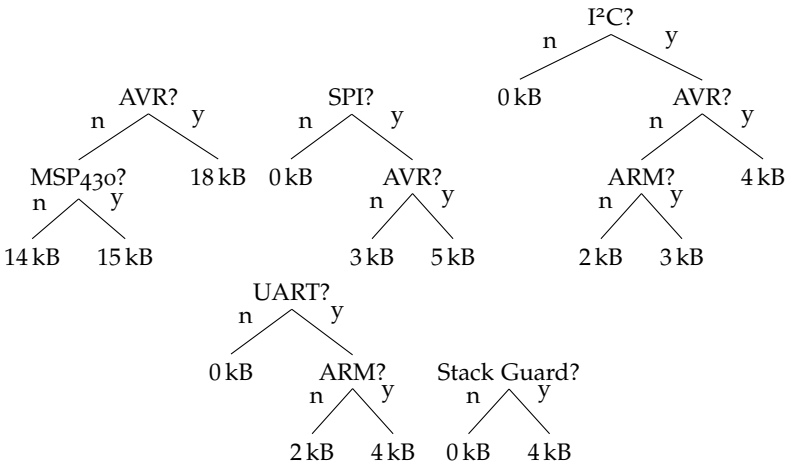


Figure 2.5: Regression forest for predicting the kernel size of a sample operating system product line.

tive state space exploration (i.e., benchmarking all possible products). Real-world applications rely on an appropriate *sampling* algorithm that decides which configurations are benchmarked [Guo+18]. In the field of product line engineering, random sampling is wide-spread and often sufficient [Per+21].

Regression Forests

The hierarchical nature of decision trees is both their biggest strength and their biggest weakness. When all features interact with each other, building a decision tree makes perfect sense as it captures these interactions. However, when the interaction is limited to subsets of features, and these subsets are pairwise independent, they must be replicated over several sub-trees.

For example, the kernel size footprint of the Stack Guard feature is independent of architecture and driver selection. In order to express its non-functional behaviour, the tree shown in Fig. 2.4 would need duplicates of each decision: one for Stack Guard disabled, and one for Stack guard enabled. So, in the worst case, each new independent feature doubles the number of decisions that are present in the tree. This makes the model more complex and harder to interpret.

Ensemble methods offer an approach for solving this challenge. Rather than using a single estimator f (here: a single decision tree), they rely on an *ensemble* $\mathcal{F} = \{f_1, \ldots, f_K\}$ of estimators (here: a *forest* consisting of several decision trees). The ensemble's output is an aggregate of each estimator's output, in this case $\mathcal{F}(\vec{x}) = \sum_{k=1}^{K} f_k(\vec{x})$.

Fig. 2.5 shows an example of a DECART-based regression forest for kernel size prediction. However, this example is purely for illustrative purposes: I have designed it manually to represent an ideal regression forest, and did not find a combination of learning algorithm and hyper-parameters that led to a similarly easy-to-understand model.

This comes from the fact that learning a forest is considerably more challenging than learning a single tree. Algorithms must balance the complexity of individual trees with the complexity of the entire model, select suitable groups of features for the individual trees, and avoid overfitting. To manage this, the implementations I am aware of combine a generic, randomized algorithm with hyper-parameters that allow for manual fine-tuning of individual algorithm components.

This thesis uses Python3's `xgboost` implementation of the *Extreme Gradient Boosting* (XGB) algorithm [CG16]. Essentially, it iteratively builds regression trees that aim to minimize the loss of the entire forest rather than just the loss of a single tree. When learning an XGB forest, users can specify several hyper-parameters, including

- number of regressors (trees) $K \in \mathbb{N}_{>0}$,
- sub-sampling rate $r \in (0, 1]$,
- maximum tree depth $T_d \in \mathbb{N}_{>0}$,
- tree complexity penalty $\gamma \in \mathbb{R}_{\geq 0}$,
- tree weight penalty $\lambda \in \mathbb{R}_{\geq 0}$, and
- shrinkage $\eta \in (0, 1]$.

The learning algorithm is more complex than CART and DECART, and methods for building regression forests continue to be an active research area. However, the details of the algorithm are not relevant for the contributions presented in this thesis. Hence, this section only gives a high-level overview of the ideas behind it; refer to Chen and Guestrin for details [CG16].

XGB starts with an empty ensemble, and iteratively adds K regressors to it. For each regressor, it randomly selects $r \cdot n$ features (where n is the number of features in the feature vector), and uses an adjusted CART algorithm to build a regression tree whose decision nodes only use the selected feature sub-set. This algorithm replaces the loss function *SSR* with a regularized variant $\mathcal{L}$ that takes the loss of the entire ensemble into account and penalizes complex models:

$$\mathcal{L}(\mathcal{F}, S) = \sum_{i=1}^{n} SSR(\mathcal{F}(\vec{x}_i), y_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{2.7}$$

Here, $\mathcal{F}$ represents the current state of the partially assembled ensemble, including the regression tree that is currently being generated. The regularization term $\Omega(f)$ calculates the complexity of each tree f from its number of leaves T and leaf values w_t.

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{t=1}^{T} w_t^2 \tag{2.8}$$

When a new tree is complete, the XGB algorithm scales each of its weights by the shrinkage parameter η. This controls the learning rate: A low η value reduces the influence of individual trees on model outcome, thus allowing additional trees to fine-tune the ensemble.

With these methods, regression forests can accurately model complex behaviour. However, especially considering the large number of trees, this comes at the cost of interpretability. We will see this effect when comparing prediction methods in Section 7.5.

Linear Model Trees

Up to this point, all regression tree variants have expressed piecewise constant functions. While these are well-suited for boolean-only feature models, they can only approximate the effect of numeric features such as RX buffer size in discrete steps. *Linear Model Trees* (LMT) address this by extending CART with linear functions in leaf nodes, thus expressing piecewise linear rather than piecewise constant functions [Qui+92; Mal+04].

Definition 2.5.5 An LMT is a binary tree that expresses a piecewise linear function $f : \mathbb{R}^n \to \mathbb{R}$. Each non-leaf node holds a decision "$x_i \leq z$" for some index $i \in \{1, \ldots, n\}$ and threshold $z \in \mathbb{R}$, and each leaf node holds a linear function $f' : \mathbb{R}^n \to \mathbb{R}$.

Definition 2.5.6 For an LMT f, $\textsc{decision}(f) = \langle i, z \rangle$ indicates that its root node holds the decision "$x_i \leq z$". $\textsc{child}(f, \mathrm{y}) = f'$ refers to the sub-tree f' for $x_i \leq z$, and $\textsc{child}(f, \mathrm{n}) = f''$ refers to the sub-tree f'' for $x_i > z$. If the root node is a leaf node annotated with the function f''', $\textsc{decision}(f) = \bot$ and $\textsc{value}(f) = f'''$.

In contrast to regression trees and forests, LMT appear to be less common in the product line engineering field. Still, they have been used to predict software faults based on software quality attributes in the past [RK16]. In this publication, Rathore and Kumar use the M5′ learning algorithm [WW97]. Again, the details of the M5′ learning algorithm are not relevant for the contributions presented in this thesis, so this section only gives a high-level overview.

Similar to xgboost, M5′ starts out with an adjusted CART algorithm: rather than minimizing a loss function, it aims to maximize the *Standard Deviation Reduction* (SDR).

$$SDR(f, S) = \sigma(S) - \sum_{i=1}^{m} \frac{|S_i|}{|S|} \cdot \sigma(S_i) \tag{2.9}$$

Just like SSR, SDR takes both accuracy and partition size into account: a small partition with a high error and a large partition with a low error behave similarly. In fact, the literature indicates that there is little difference between SSR and SDR [WW97].

BENCHMARK #	1	2	3	4	5
x_{UART}	0	1	1	1	1
$x_{\text{RX Buffer}}$	0	8	16	32	64
y (RAM [B])	64	76	84	100	132

Table 2.2: Excerpt of configurations and static memory usage for a sample operating system product line. Each row describes a configuration variable x_i or performance attribute y. In configurations with UART disabled, RX Buffer size is zero.

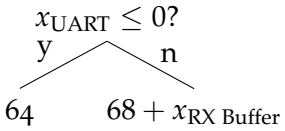


Figure 2.6: Linear Model Tree for predicting the static memory usage of a sample operating system product line. Each leaf holds a constant value or a linear regression formula.

Once the CART has been built, M5′ transforms sub-trees into linear functions by means of a bottom-up pruning algorithm. For each sub-tree (i.e., each non-leaf node), it builds a linear function template that only considers the variables that are queried within the sub-tree. It then uses linear regression to fit it on the corresponding observations (i.e., the set S that was used to determine the root node of the sub-tree). The resulting function is also subject to pruning: the M5′ algorithm greedily drops function terms that have little influence on the accuracy of the fitted function. Now, moving from the leaves to the top, it calculates the prediction error of both the CART-style sub-tree and the corresponding linear function for each non-leaf node it encounters. If the linear function's error is lower, it replaces the sub-tree with a single leaf node that contains the linear function. In order to simplify algorithmic notation, this thesis treats each leaf node that still contains a constant value y after the pruning step as a constant function $f(\vec{x}) = y$.

To illustrate the benefit of LMT, let us consider the static RAM usage of our sample operating system product line. For brevity, we pretend that it is identical for all architectures, and that the SPI, I²C, and Stack Guard features do not affect it either. However, the UART driver's RX buffer must be placed in memory, so UART buffer size ($x_{\text{RX Buffer}}$) and whether or not the UART driver is enabled to begin with (x_{UART}) affect RAM usage as shown in Table 2.2.

Feeding this into an open-source LMT implementation[4] results in the tree shown in Fig. 2.6. We immediately see the linear effect of buffer size on memory usage, and that it is only relevant if the UART

4 https://github.com/cerlymarco/linear-tree

Algorithm 5 Calculate $f(\vec{x})$ for an LMT f.

```
function QueryLMT(f, x⃗)
    if decision(f) = ⊥ then
        return value(f)(x⃗)
    ⟨i, z⟩ ← decision(f)
    if x_i ≤ z then
        return QueryLMT(child(f, y), x⃗)
    else
        return QueryLMT(child(f, n), x⃗)
```

driver is enabled. As shown in Algorithm 5, querying an LMT works just like querying a CART with the additional step of evaluating the leaf function rather than returning a leaf value.

Just like XGB, the implementation used here supports a variety of hyper-parameters for balancing model accuracy, complexity, and risk of overfitting. These include

- maximum tree depth T_d,
- the minimum number of samples T_m for splitting (stop criterion: $|S| < T_m$), and
- the minimum number of samples T_l per partition that a split must provide in order to be considered for tree generation (stop criterion: $\forall i, z : \left|S_{i,z,\mathrm{y}}\right| < T_l \wedge |S_{i,z,\mathrm{n}}| < T_l$).

While all of these serve as stop criteria, T_l also constrains which partitioning schemes "$x_i \leq z$ / $x_i > z$" are considered during tree generation. Note that there is an implementation-specific limitation of $T_d \in \{5, \ldots, 20\}$, whereas CART and XGB support $T_d \in \mathbb{N}_{>0}$. Likewise, $T_m \geq 6$ and $T_l \geq 3$; both may also be specified as fractions of the total number of samples m rather than absolute values.

2.5.3 *Lookup Tables*

Technically, *Lookup Table* (LUT) models do not fall under the term of machine learning. Rather than abstracting from observations, they are only capable of reproducing observations for known configurations. A LUT model partitions the training set by feature vector, builds a table that maps each feature vector $\vec{x}$ to corresponding training observations y, and returns the mean of these observations when asked to predict the performance of a specific configuration $\vec{x}$. If $\vec{x}$ is not part of the training data stored in the lookup table, it returns the mean of all observations instead.

Let $Uniq_X(S)$ refer to the unique configurations $\vec{x}$ in the training set S, and let $S_{\vec{x}}$ hold the pairs that correspond to a given configuration $\vec{x}$:

$$Uniq_X(S) = \{\vec{x} \mid \exists y : (\vec{x}, y) \in S\} \tag{2.10}$$

$$S_{\vec{x}} = \{(\vec{v}, y) \in S \mid \vec{v} = \vec{x}\} \tag{2.11}$$

With these shortcuts, the LUT model $f(\vec{x})$ is defined as follows.

$$f(\vec{x}) = \begin{cases} \mu(S_{\vec{x}}) & \text{if } \vec{x} \in Uniq_X(S) \\ \mu(S) & \text{otherwise} \end{cases} \tag{2.12}$$

Essentially, this is variant-wise feature modeling taken to the extreme: Each product is its own variant, and the model does not learn common traits or feature-specific attributes. As such, LUT models are not helpful for understanding performance behaviour, and no better than a feature-agnostic arithmetic mean for performance prediction of unseen configurations. However, by construction, they provide a lower bound on model error. When only evaluated on observations that were part of the training set, a LUT model's prediction error precisely captures the measurement uncertainty within its training data.

Given a collection of machine learning methods, this allows practitioners to determine whether individual algorithms are able to accurately capture product line performance regardless of absolute error values. For instance, if a benchmark series consists of ten measurements per configuration and there is a large variance between measurements of identical configurations, the LUT model will have a high model error. Regardless of modeling method, a machine learning model has no chance of coming up with results that are better than the measurement uncertainty. So, even if a method such as CART has a seemingly high model error, if it is close to LUT model error it is still as good as the available training data permits.

This concludes definitions and related work for machine learning methods. Having learned what a machine learning model is and how it behaves, we are now ready to examine quality metrics that allow for a quantitative comparison of machine learning models.

2.6 MODEL QUALITY METRICS

Performance-aware product line configuration benefits from accurate and understandable models. In quantifiable terms: product line engineers should strive for models with a low prediction error and, especially when configuration is done manually, low model complexity. This section presents common metrics for both, and defines the ones used in the remainder of this thesis.

2.6.1 *Prediction Error*

A key question when using a performance prediction model is: given a configuration $\vec{x}$, how close is the model prediction $p = f(\vec{x})$ to actual system performance, i.e., to the performance value y that users would obtain by building and benchmarking a product with this configuration? Or, in short: what is the prediction error?

Of course, the concrete configurations $\vec{x}$ that users will ask for are not known in advance – otherwise, engineers could include them in the benchmark set and would not need a performance model in the first place. Hence, the established method is to report mean prediction error for a representative set of configurations [Per+21]. As this set validates the model, it is also known as the *validation set*.

All error metrics presented in this section operate on such a validation set, and take two related sets of values as input: The *ground truth* $Y = \{y_1, \ldots, y_m\}$, and model predictions $P = \{p_1, \ldots, p_m\} = \{f(\vec{x}_1), \ldots, f(\vec{x}_m)\}$.

Error Metrics

Back in Equation 2.1, we have already seen an error measure that works this way: the SSR loss function. Adjusted for P rather than f and X, it is defined as follows.

$$SSR(P, Y) = \sum_{i=1}^{m} (y_i - p_i)^2 \tag{2.13}$$

This metric allows for comparing the prediction error of different models on the same data set, but is unsuitable for comparing models that were evaluated on distinct data sets. Given one model with $P_1 = \{1, 2\}$ and $Y_1 = \{2, 3\}$, and a second one with $P_2 = \{1, 2, 3\}$ and $Y_2 = \{2, 3, 4\}$, one might argue that both are equally accurate. However, as the SSR is a sum of individual deviations, the second model has a higher SSR.

To avoid this, model evaluation can consider the mean of deviations rather than their sum, using *Mean Square Error* (MSE) or *Mean Absolute Error* (MAE). These are also known as *Mean Square Deviation* (MSD) and *Mean Absolute Deviation* (MAD), respectively.

$$MSE(P, Y) = \frac{1}{m} \sum_{i=1}^{m} (y_i - p_i)^2 \tag{2.14}$$

$$MAE(P, Y) = \frac{1}{m} \sum_{i=1}^{m} |y_i - p_i| \tag{2.15}$$

The former is more sensitive to outliers, and is minimal for models that use the arithmetic mean for prediction, such as the LUT model. The latter is minimal for models that use the median for prediction, and easy to interpret: given a prediction p and mean absolute error

x, on average, the actual performance value will be in the range $[p - x, p + x]$.

These metrics are useful when comparing different models that aim to predict the same performance property for the same product line. They can lead to confusion when looking at different performance attributes or product lines, though: while an MSE or MAE of a few seconds would be excellent when predicting the duration of a complex, multi-hour number crunching algorithm, it would not be helpful for latency prediction of a driver function call with a typical duration in the hundred milliseconds range.

Mean Absolute Percentage Error (MAPE) resolves this inconvenience by calculating model error relative to ground truth values: a model that consistently reports 50 % too much or too little ($p_i = \frac{1}{2} y_i$ or $p_i = \frac{3}{2} y_i$) has a MAPE of 50 %. However, it is only defined for non-zero observations ($\forall i : y_i \neq 0$).

$$MAPE(P, Y) = \frac{100\,\%}{m} \sum_{i=1}^{m} \left| \frac{y_i - p_i}{y_i} \right| \tag{2.16}$$

Symmetric Mean Absolute Percentage Error (SMAPE) also indicates the relative deviation between prediction and observation. Unlike MAPE, it has a fixed range between 0 and 200 %, allowing for an easier graphical comparison during model evaluation at the cost of non-linear behaviour. It can only be computed if all validation set entries have a non-zero prediction or ground truth ($\forall i : |p_i| + |y_i| \neq 0$).

$$SMAPE(P, Y) = \frac{100\,\%}{m} \sum_{i=1}^{m} \frac{|p_i - y_i|}{\frac{|p_i| + |y_i|}{2}} \tag{2.17}$$

MAPE and SMAPE have two important differences. First, MAPE behaves the same for over- and undershoot, whereas SMAPE gives different results for the same relative deviation: a consistent overshoot of 50 % ($p_i = \frac{3}{2} y_i$) gives a SMAPE of 40 %, whereas an undershoot of 50 % ($p_i = \frac{1}{2} y_i$) gives a SMAPE of 67 %. Second, due to its unbounded nature, MAPE can be severely influenced by outliers with very high prediction errors. If one out of one hundred predictions is off by a factor of one thousand and all other predictions are spot-on, MAPE is 999 %, whereas SMAPE is just 2 %. This is most relevant when working with models that attempt to extrapolate system behaviour (e.g. regression analysis) rather than just interpolating between measurements (e.g. CART).

The majority of performance model evaluations in the product line engineering literature uses MAE and MAPE[5] to assess prediction accuracy [Per+21]. As this thesis focuses on machine learning models that work reliably on more than just a single product line, it uses

5 Note that the referenced survey uses different terms than this thesis: "Mean Relative Error" subsumes MAE and MAPE, whereas "Mean Absolute Error" refers to the sum of deviations rather than the mean of deviations.

relative error metrics. Evaluation results report MAPE where possible, and fall back to SMAPE if MAPE cannot be visualized properly.

Cross Validation

A product line with n boolean features has up to 2^n valid configurations, making exhaustive benchmarks of product lines with hundreds or thousands of features impractical. Hence, predicting the performance of configurations that were not part of the training set is a key application of performance models, and error metrics must acknowledge this fact. Otherwise, there would be little need for machine learning research, as the LUT model would already provide optimal prediction results for SSR and MSE, and a LUT model that uses median rather than mean would provide optimal prediction accuracy for MAE, MAPE, and SMAPE. For performance attributes with no measurement uncertainty (e.g. binary size), all LUT error metrics would be zero.

This can also happen unintentionally. Complex models combined with sparse and possibly noisy training data can behave like a LUT model rather than learning underlying characteristics, making them unsuitable for prediction of previously unseen configurations. This is known as *overfitting* [DS98].

Hence, evaluating models on configurations that were not present in the training data set is crucial for obtaining reliable error metrics. At the same time, it is desirable to use as many benchmark results as possible for training and validation. The configuration space is often too large for exhaustive benchmarks, and distinct training and validation sets further reduce the usable amount of configurations.

Cross validation manages to do both: it uses all available benchmark results for training and validation, while at the same time ensuring that training and validation set are distinct. The key idea is to generate several pairs of training and validation sets from the available benchmark results, train a separate model on each pair, and use the average prediction error of all models for evaluation [Koh95]. This way, it uses each benchmark result for training and validation, while ensuring that each model is only evaluated on data points it was not trained on.

As before, let $S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\}$ be the set of observations, using an arbitrary (but fixed) order. *10-fold cross validation* generates ten deterministic splits, or *folds*, each of which uses 90 % of observations as training set $S_{t,k}$ and the remaining 10 % as validation set $S_{v,k}$. For $k \in \{1, \ldots, 10\}$, training and validation set are defined as follows.

$$S_{t,k} = \{(\vec{x}_i, y_i) \in S \mid i \mod 10 \neq k - 1\} \tag{2.18}$$

$$T_{v,k} = \{(\vec{x}_i, y_i) \in S \mid i \mod 10 = k - 1\} \tag{2.19}$$

10-fold cross validation uses each training set to train a separate model, and stores its predictions on the corresponding validation set.

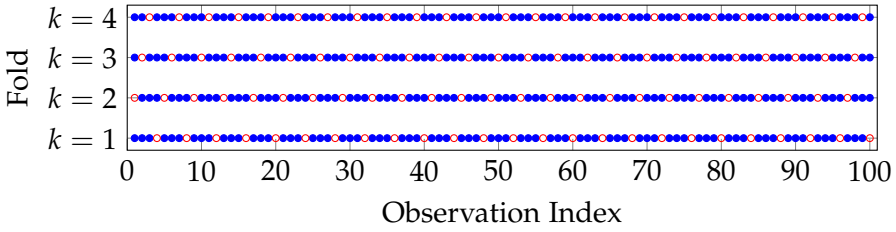


Figure 2.7: Training and validation sets in 4-fold cross validation with 100 samples. Each line shows a separate model with training set (blue, filled circles) and validation set (red, empty circles).

The first one uses $S_{t,1}$ and $S_{v,1}$, the second one $S_{t,2}$ and $S_{v,2}$, and so on. In all cases, $S_{t,k} \cup S_{v,k} = S$ and $S_{t,k} \cap S_{v,k} = \emptyset$. This results in a ground truth Y that covers the entirety of available benchmark results – however, each prediction p_i from the set P has been made by a model that did not observe the corresponding ground truth entry y_i during training.

Fig. 2.7 illustrates the training and validation partitions for a simplified variant with $k \in \{1, \ldots, 4\}$ (so, 4-fold cross validation). The first model ($k = 1$) uses observations $4, 8, 12, \ldots, 100$ for validation and all others for training. The second model uses observations $1, 5, 9, \ldots, 97$ for validation and all others for training, and so on.

Feature-Aware Cross Validation

This cross validation method works if all benchmarks measure distinct configurations. For properties such as binary size, this is a sound assumption: no matter how often the program is compiled, binary size is always the same, so a single compilation benchmark for each configuration is sufficient.

Other performance attributes, such as latency of a Linux application or energy usage of an embedded board, are influenced by uncontrolled environmental effects (*noise*). Here, repeated benchmarks with the same configuration may yield different results, and it is important that machine learning algorithms can distinguish between deviations due to noise and deviations due to configuration changes. Therefore, it is good practice to benchmark each configuration several times.

When using cross validation in this case, the training and validation sets still contain distinct configuration/observation pairs. However, a configuration in the validation set may already have been present in the training set with a slightly different benchmark result, thus defeating the purpose of cross validaton.

Feature-aware 10-fold cross validation works around this by building partitions not based on the order of observations, but on distinct feature vectors $\vec{x}$. In each split, 90 % of configurations (and corresponding observations) are part of the training set, and 10 % are part of the validation set. With the set of unique configurations $X_{\text{uniq}} = \{\vec{x}_1, \ldots, \vec{x}_j\}$,

using an arbitrary but fixed order, training and validation sets are defined as follows.

$$S_{t,k} = \bigcup_{\{\vec{x}_i \in X_{\text{uniq}} | i \mod 10 \neq k-1\}} S_{\vec{x}_i} \tag{2.20}$$

$$T_{v,k} = \bigcup_{\{\vec{x}_i \in X_{\text{uniq}} | i \mod 10 = k-1\}} S_{\vec{x}_i} \tag{2.21}$$

Thus, for each unique configuration $\vec{x}$, either all corresponding observations are part of the training set or all observations are part of the validation set. Unless noted otherwise, all evaluations in this thesis use feature-aware 10-fold cross validation.

2.6.2 *Complexity*

The interpretability of a model depends on the number of questions a user must consider to determine model output for a specific product line configuration, and the difficulty of determining how changing individual features would affect it. In contrast to prediction error, I am not aware of a common method that allows for quantitative interpretability evaluation of least-squares regression and regression tree-based models. While there are quantitative metrics that allow for comparing arbitrary machine learning models (e.g. Rademacher complexity [BM01]), they focus on learning ability rather than the understandability of concrete models.

Therefore, this thesis uses model complexity as a proxy variable for interpretability. For least-squares regression, the number of function terms (i.e., the number of weights $\left|\vec{\beta}\right|$) appears to be a common complexity metric [McC11]. For a tree or forest f, it is the number of leaves (also known as terminals or weights) $\#w_f$ [Ste09; LCG12].

While the number of leaves captures the number of product configurations that a regression tree-based model has learned, it does not capture the work that users have to invest when answering "what if?" questions. For these, they also need to look at non-leaf nodes within the tree, and determine which branch they want to take. In this context, it makes sense to consider the total number of nodes $\#n_f$.

In binary trees and forests, this is a function of the number of leaves: $\#n_f = 2\#w_f - 1$. So, when only dealing with binary trees, it does not matter whether complexity metrics use the number of leaves or the total number of nodes, so long as the complexity calculation method is consistent. However, in this thesis, we will also examine non-binary tree structures with $\#n_f \leq 2\#w_f - 1$. Therefore, I define tree complexity as the total number of nodes $\#n_f$.

LMT combine regression trees and least-squares regression: each leaf holds a linear regression formula f with weights $\vec{\beta}_f$. A constant leaf value y is expressed as a function $f(\vec{x}) = y$ with $\vec{\beta}_f = (y)$ and

does not require special treatment. Hence, LMT complexity depends on the number of non-leaf nodes $\#i_f$ and the total number of weights $\left|\vec{\beta}_f\right|$ in the set of leaf functions $\mathcal{F}$. With these considerations in mind, this thesis uses the following complexity metrics.

- Least-squares regression formula f: number of weights $\left|\vec{\beta}_f\right|$.
- Regression tree or forest f, e.g. CART, DECART, XGB: number of nodes $\#n_f$.
- Regression tree f with set of leaf functions $\mathcal{F}$, e.g. LMT: number of non-leaf nodes plus number of weights in leaf functions: $\#i_f + \sum_{g \in \mathcal{F}} \left|\vec{\beta}_g\right|$.
- LUT model with distinct configurations $Uniq_X(S)$: number of configurations it has learnt by heart: $|Uniq_X(S)| + 1$.

2.7 EVALUATION TARGETS

In the past decades, dozens of software product lines and SPL-like software products have been subject to performance modeling research, ranging from compression algorithms over database systems and related server applications to video encoders and other software [Per+21]. Predicted performance attributes often include latency and throughput, memory usage, and binary size. Evaluation targets typically exhibit either compile-time or run-time variability; combinations of both are rare. Evaluations of workload-dependent attributes such as latency and throughput typically consider the workload (e.g. video encoder input file or performed database queries) to be constant.

In line with these common evaluation methods, this thesis relies on five Kconfig-based software projects as evaluation targets for performance models: three with compile-time variability, one with run-time options, and one with both. Unless noted otherwise, all benchmarks use random sampling and run no more than one benchmark per product line configuration. Random sampling is used by the majority of performance modeling research works, and has been shown to be sufficient in many cases [Per+21; Guo+18]. Repeated measurements of the same configuration are helpful for performance attributes affected by run-time effects, but not required for compile-time attributes such as binary size.

One of the goals of product line engineering is that each valid configuration results in a working product. In practice, this may be violated, especially when dealing with software projects that behave like an SPL, but are not developed according to SPLE principles. While there is a growing body of research related to removing such deficiencies in the variability model and improving the relationship between feature model and source code implementation in general [Tar+11], this issue

PRODUCT LINE	BOOLEAN		CHOICE	NUMERIC	SAMPLES
Busybox	1,018	997	7	25	32,000
Kratos	198	183	5	137	30,000
Multipass	138	86	6	8	10,000
x264	9	5	2	4	16,800
resKIL	106	0	6	1	14,884
resKIL (latency)					$3.82 \cdot 10^6$

Table 2.3: Number of features by type and sample counts of evaluated product lines. The Boolean column indicates total number of features (left) and number of features that are not part of a choice (right).

is not within the scope of this thesis. Hence, all configurations examined in this thesis refer to working products; configurations that lead to compilation errors or other issues are silently discarded.

Table 2.3 shows the number of boolean and numeric features, the number of groups of alternative features (Kconfig `choice` entries), and the number of samples for each target. The left boolean column counts all boolean features, whereas the right one only considers boolean features that are not part of a `choice` entry. resKIL has two data sets for different types of performance attributes – however, the number and distribution of features is identical for both. I will now present the evaluation targets in detail.

2.7.1 *Compile-Time Variability*

The *busybox* project[6] provides a collection of common UNIX utilities within a single multi-call binary. It is tailored towards resource-constrained embedded Linux appliances. As such, it is extensively configurable: each utility can be separately enabled or enabled with only a sub-set of features, and many global attributes such as buffer sizes are configurable as well. Specifically, we are looking at busybox version 1.35.0, with binary size and static RAM usage (combined data and bss segment size) as performance attributes.

I am not aware of related works that use busybox for evaluation. While Queiroz, Berger, and Czarnecki aim to predict defects in busybox feature implementations, their approach is not applicable to performance modeling [QBC16].

Kratos is a research operating system developed at the former Embedded System Software group at TU Dortmund [Bus19]. It has been designed from the ground up with variability in mind, and hence offers a wide range of configuration options, including several dozen integer and string features used for fine-grained control over hardware

6 https://busybox.net

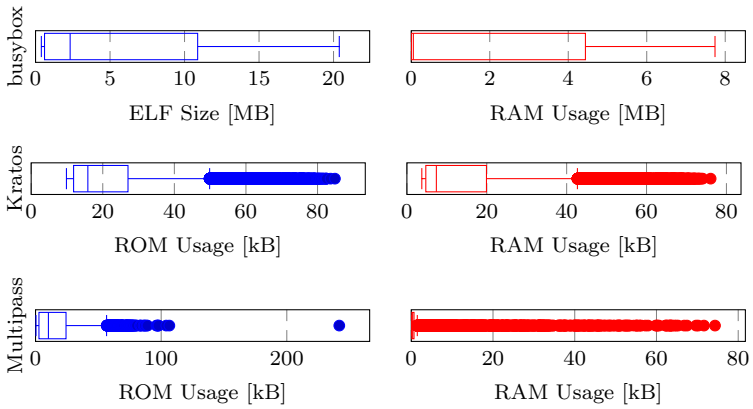


Figure 2.8: Observed performance attributes in compile-time variability benchmarks. Box segments indicate quartile boundaries.

components. It supports two MSP430-based microcontroller families (MSP430FR and CC430) and ARM (Raspberry Pi). Performance attributes are similar to busybox: ROM usage (text and data segment size) and static RAM usage (data and bss segment size).

Multipass[7] is a library operating system that I developed over the past years to help with my research. It is designed with hardware evaluation and energy measurements in mind, and hence deliberately leaves out sources of non-determinism such as preemptive multitasking. Still, it supports a variety of architectures (AVR, MSP430FR, ARM Cortex R and M, TriCore), peripherals, and test applications, all of which are exposed as configurable features. Again, performance attributes are ROM usage (text and data segment size) and static RAM usage (data and bss segment size).

Fig. 2.8 shows the distribution of observed performance attributes in these three evaluation targets as box plots that indicate quartile boundaries and outliers. Note that the individual plots use different X axis scales for better visualization. Busybox exhibits significant variance, with ELF size and RAM usage ranging from a few kB to several MB, and is therefore a prime candidate for performance prediction models. While the majority of Kratos and Multipass resource requirements falls in a range less than 10 kB wide, the high amount of outliers indicates that they, too, will likely benefit from prediction models.

2.7.2 *Run-Time Variability*

The *x264* software library[8] provides a well-known open source H.264 video encoder. It offers numerous boolean and numeric tunables that affect the trade-off between video quality, encoder speed, and output

7 https://ess.cs.uos.de/git/bf/multipass
8 https://www.videolan.org/developers/x264.html

file size. Here, the goal is to predict how a sub-set of encoder options affects encoding duration and output file size, using ffmpeg version 4.3.4 and libx264 revision 3011 (commit `cde9a93`).

The input file for encoding is constant, which is in line with other works that use x264 for evaluation purposes [Zha+15; Guo+18; Sie+15; Sie+13]. Three of those only examine the influence of boolean feature toggles, whereas Siegmund et al. consider eight boolean and 13 numeric features, though without documenting the precise feature set [Sie+15]. In this thesis, the configuration set also covers boolean and numeric features, including tunables such as output resolution, bit rate, and the number of encoding threads.

I built a hand-crafted Kconfig file to capture the variability and constraints in x264's run-time options, and implemented tooling that explores the configuration space by means of random sampling with neighbourhood exploration. This is an extension of random sampling that, for each random sample, also performs benchmarks where a single feature has been changed. For a boolean feature, this means benchmarking a configuration where this specific feature has been toggled. For a numeric feature, it benchmarks five configurations where this specific feature has been configured to one of five values taken at equidistant points between the feature's minimum and maximum value. Additionally, as encoding duration is affected by the background load of the benchmark machine in addition to encoder configuration, each x264 benchmark runs three times.

resKIL is an agricultural AI product line that will be explained in detail in Section 8.1. It models a computer vision product for image classification (e.g. determining which type of fruit is present in an image) and semantic segmentation (locating and identifying image components, e.g. obstacles and wheat). The product line uses variable hardware platforms, AI models, model optimizations, inference frameworks, and batch size.

Hardware platforms range from low-power Cortex-A single-board computers to GPU-accelerated Jetson Nano and Xavier boards. Batch size indicates the number of images that is processed in a single operation – depending on AI model and framework, increasing the batch size can significantly increase throughput with only small changes to latency. Performance attributes are model size, inference latency, inference throughput, and memory usage.

resKIL data acquisition also systematically explores the configuration space, but achieves this by iterating over an exhaustive list of feature configurations. Each benchmark of a specific configuration populates two data sets. First, it measures the per-image latency of running inference on several dozen images, ignoring the first operation to account for warm-up effects such as memory allocation. The results end up in the latency row of Table 2.3; the generic resKIL row covers the remaining performance attributes.

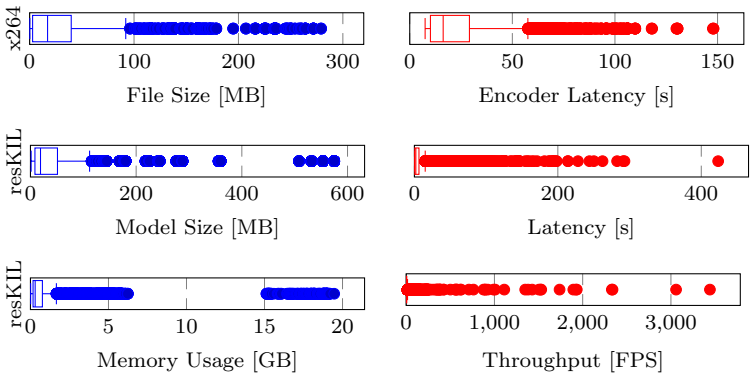


Figure 2.9: Observed performance attributes in run-time variability benchmarks.

It also measures system memory usage before loading the AI model and after the last inference operation, and treats the difference between those as memory overhead of the inference operation – so, for each set of several dozen latency measurements, there is a single memory usage data point. This crude method is necessary to account for the memory usage of implementations using offloading, e.g. by moving operations to a GPU or to specialized tensor processing units (TPUs). The memory used for these offloading engines may be allocated by system components other than the benchmark script, and hence be invisible to fine-grained approaches such as malloc tracing.

Afterwards, for each benchmark run, it calculates inference throughput from batch size and median latency, and stores throughput, memory usage, and serialized model size in the list of benchmark results. In the end, this results in two sets of measurements: one for latency, with several dozen measurements for each distinct configuration, and one for everything else. Due to time constraints, I only benchmarked each configuration in the latter set twice. Hence, memory usage data may be subject to interference by other operating system tasks.

Fig. 2.9 shows the distribution of the observed x264 and resKIL performance attributes. We see that all attributes have a high variance, often over several orders of magnitude. So, resKIL and x264 will likely benefit from performance prediction models as well.

2.8 CHAPTER SUMMARY

At this point, we know that software product lines (and, in many cases, configurable software projects) build upon features that describe individual configuration options. Feature models capture the relationship and dependencies between features in a human- and machine-readable manner; the Kconfig language is a common method of specifying them in open-source software projects. Practitioners can extend feature models with feature- or variant-wise annotations for

performance prediction, or rely on machine learning methods to automatically build a performance model. For the latter, we have examined least-squares regression analysis, regression trees (CART, DECART), model trees (LMT), and regression forests (XGB). We have also seen methods and metrics that researchers can use to assess model accuracy and interpretability. Finally, I have presented five product lines that serve as evaluation targets in this thesis.

However, we will not look into ways of improving the accuracy and interpretability of performance models right away. Instead, the next section introduces the other focus area of this thesis: energy models for configurable hardware peripherals.

3

ENERGY MODELS

When designing an embedded system, memory is not the only resource that engineers need to care about. Energy is at least as important, especially so since it is often a consumable resource. In battery-powered applications, energy usage determines how long a single charge lasts. With energy harvesting, where e.g. miniature solar cells are the only power source, it decides whether the system can operate at all. Even on grid power, energy is often relevant; for instance, in CPUs, energy usage relates to heat dissipation and must not exceed the available cooling capacity.

Just like the performance attributes we have examined in the previous chapter, energy usage in embedded systems is far from static. On the one hand, embedded systems consist of configurable and – depending on use case – interchangeable components, including microcontrollers, sensors for measuring environmental data, actuators for reacting to it, and wireless radio transceivers. Just like software product lines, their configuration can affect attributes such as energy usage or latency [ZV16]. On the other hand, whereas e.g. the text segment size of a device driver is constant after compilation, this is not the case for energy usage. Even for a fixed configuration, it depends on the executed workload [HHS19].

To illustrate this, consider an engineer who is designing a battery-powered wireless sensor network that measures environmental data and regularly transmits it to a central hub. Apart from the sensors, which we will not examine here, each node consists of at least two components: a microcontroller and a radio chip. The microcontroller can run at various clock speeds that affect processing latency, which in turn dictates how long it must remain active before it can go back to a low-power sleep mode. The radio chip uses configurable transmit power (affecting packet loss and thus the risk of spending energy to re-transmit data) and data rate (affecting the time it must remain in transmit mode). It can enter a low-power sleep mode between transmissions, but needs time and energy to wake up from it. So, the most energy-efficient configuration depends on factors such as transmission frequency, expected packet loss, and CPU load.

Energy models predict how workload and device configuration affect latency and energy usage, and thus allow engineers to minimize them and reason about multi-objective optimization problems such as energy usage versus expected packet loss. Hence, they are wide-spread in the *Cyber-Physical Systems* (CPS) and IoT communities [Sna+16].

My own experience with energy models started with my Master's thesis at TU Dortmund, where I developed work-in-progress algorithms for automated benchmark generation, energy measurements, and energy model generation with Markus Buschhoff [Fri17; BFS18]. Parts of this thesis build upon those foundations. While I have improved all of them since then, many of those changes are incremental, and thus best considered as related work rather than contributions in the context of this thesis. Sections 5.2 and 6.2 as well as Chapter 7 will focus on the latter.

This chapter covers state-machine models that describe the latency and energy requirements of individual hardware components depending on workload (function calls and hardware states) and configuration (e.g. data rate, transmit power, clock speed). It starts with a simple energy model that employs a state machine for energy usage prediction without respecting device configuration. In addition to a gentle introduction to state machines, this allows for a refresher on the relation between energy, power, and latency (i.e., time) when dealing with workload specifications and corresponding energy usage predictions.

We will then look into applications of energy models in the literature; this overview also serves as motivation as to why state machines are a good fit for the goals of this thesis. After that, we will examine the *Parameterized Priced Timed Automata* state machine extension and see how it can describe the run-time variability of hardware components and its influence on energy attributes, thus allowing energy models to support configurable devices.

The remainder of this chapter covers automated generation of benchmarks for energy measurements and the *Unsupervised Least-Squares Regression* machine learning algorithm for extending finite automata to parameterized priced timed automata. It concludes with an overview of peripheral components used as evaluation targets within this thesis.

3.1 STATE MACHINES

In general, hardware components have distinct operating modes. For instance, a radio transceiver may be in transmit or receive mode, idle, or in a low-power sleep state. Each of these has a distinct average power consumption, and switching between modes is not necessarily instantaneous.

So, from a modeling point of view, we are looking at a state machine. Each state corresponds to a hardware operating mode (hardware state), and each transition changes the operating mode. The change may be induced by an explicit driver function call from an application, e.g. `radio.transmit(data, len)`, or by the hardware itself. In the latter case, it is typically still known to the driver thanks to interrupts or timeout mechanisms. For instance, most radio chips will automatically revert from transmit to idle mode upon a completed transmission,

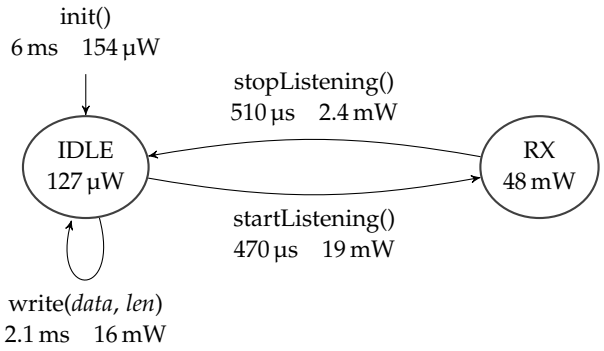


Figure 3.1: State machine model excerpt for an nRF24L01+ radio transceiver without run-time configuration, annotated with energy attributes.

and indicate that by asserting a *transmitDone* interrupt. This results in a one-to-one relation between driver functions and state transitions: *transmit* indicates a transition from the idle state to the transmit state, and *transmitDone* the inverse.

Fig. 3.1 shows a state machine model excerpt for an nRF24L01+ radio transceiver that might be responsible for communication handling in a battery-powered wireless sensor network. In this excerpt, there are two states: the default IDLE mode, and the RX (receive) mode that listens for incoming radio transmissions. There is no distinct TX (transmit) state: the write function blocks until transmission is complete, so from the model's point of view the entire transmission happens during the function call (i.e., during a transition from IDLE to IDLE). The model annotates each state with average power usage, and each function call (transition) with duration and average power.

With these annotations, engineers can determine workload-specific energy usage. To do so, they can exploit the relation between time $t[s]$, power $P[W]$, and energy $E[J]$. Energy is power over time, so the energy used during a time frame (t_1, t_2) is

$$E = \int_{t_1}^{t_2} P(t)\,\mathrm{d}t \tag{3.1}$$

If power is constant, or only average power P is known, this resolves to $E = Pt$. So, with a model such as the one shown in Fig. 3.1, engineers can determine the total energy usage for arbitrary workloads, and also the mean power usage during arbitrary workload segments.

Consider an application that calls init followed by startListening, and then remains in receive mode indefinitely. Using the model annotations and (3.1), we see that the total energy used in the first hour of operation is $154\,\mu\mathrm{W} \cdot 6\,\mathrm{ms} + 19\,\mathrm{mW} \cdot 470\,\mu\mathrm{s} + 48\,\mathrm{mW} \cdot (3600\,\mathrm{s} - 6470\,\mu\mathrm{s})$. This comes up to about 173 J, or 7 % of the energy stored in a CR2032 coin cell battery. By comparison, an application that keeps the transceiver in its IDLE state and only spends one second per minute in RX mode needs just 4.5 J in the same time frame.

As another example, assume that the application runs a main loop that keeps the radio idle and transmits a radio packet every eight minutes. The energy used during this eight-minute frame is $127\,\mu W \cdot (480\,s - 2.1\,ms) + 16\,mW \cdot 2.1\,ms \approx 61\,mJ$ and the average power draw in the main loop is $\frac{61\,mJ}{480\,s} \approx 127.1\,\mu W$.

So, the energy model has already given us important advice: when keeping the radio transceiver in idle mode, occasional data transmissions have a negligible effect on average power usage. Keeping the transceiver in continuous receive mode, however, will quickly drain the battery.

Section 3.3 provides formal definitions for this energy model and also presents models that consider run-time configuration variables. Before that, let us put this approach into perspective by examining modeling approaches from the literature.

3.2 RELATED WORK

The term *energy model* applies to a wide range of model granularities. Approaches in the literature range from fine-grained simulators that examine dynamic and static power consumption of individual transistors to component-level models that work with driver APIs or sensor node states [KZ08]. In the latter case, components can be anything between peripherals of an embedded device and entire devices in a wireless sensor network.

Fine-grained models often focus on CPUs, as their energy usage depends not just on clock speed and operating mode, but also on the type of executed instructions and operand values [Pal+17]. This is helpful for energy-efficient software development: simulators or energy models can annotate the energy usage of individual code blocks, and thus identify inefficient code paths [HHS19]. However, it is not within the scope of this thesis.

Component-level energy models distinguish between *online* and *offline* modeling. Online models are evaluated at runtime and often rely on performance counters provided by the operating system to calculate energy usage [DZ11]. Offline models are evaluated beforehand and work with user-specified workloads rather than operating system features. As this thesis focuses on energy-aware hardware selection and configuration rather than run-time energy optimizations, the remainder of this section will only cover offline models.

3.2.1 *Modeling Methods*

A straightforward approach is using least-squares regression to fit a user-provided power prediction function to each hardware state, ignoring function calls [Zha+10]. This is similar to feature-wise performance models for software product lines (see Section 2.5.1). Examples

include $P_{\text{display,on}}(\vec{x}) = \beta_0 + \beta_1 \cdot x_{\text{brightness}}$ (linear influence of brightness configuration on display power) and $P_{\text{sensor,on}}(\vec{x}) = \beta_0$ (constant power if a sensor is enabled). Then, with online performance counters or offline estimates that indicate the amount of time spent in each hardware state and its configuration, users can determine total energy E and average power $P = \frac{E}{t}$ [Mur+12].

This approach assumes that the time and energy cost of function calls is negligible and that the time spent in each hardware state can be tracked or estimated. For peripherals such as radio chips, this is not the case. Here, entering the transmit or receive state is not instantaneous, so five 10 ms transmit bursts are not the same as a single 50 ms transmit operation [Hur+11].

An opposite approach is removing hardware states from the equation and only looking at the energy cost of function calls. If each hardware component automatically falls back to a low-power idle state after a while, as is often the case in smartphone peripheral APIs, function calls are all that is needed for energy modeling [KB11]. However, this does not work if state transitions are part of the driver API, e.g. by allowing applications to toggle a radio's receive mode or to switch between an ultra-low-power sleep and a low-power idle state.

Hence, in general, a model must at the very least accommodate hardware states (with average per-state power consumption) and function calls (with expected duration) [Zho+11]. Depending on assumptions about hardware behaviour and model granularity, it may predict the power or energy cost of function calls as well, or assume that power usage during a function call is the average power of origin and destination state [Zho+11; BFS18]. This thesis associates hardware states with average power and function calls with average power and duration.

At this point, it makes sense to have an explicit state machine model that maps driver function calls to transitions between hardware states and provides power and duration annotations [McC+11; Pat+11]. The state machine defines which function call sequences are legal, which hardware state a function call switches to, and how long it takes to end up there. It can even be learned automatically, e.g. by associating each hardware register configuration with a distinct hardware state [ZV16], or by means of changepoint detection and clustering [Che+17]. Now, each state and transition corresponds to a single, configurable aspect of the peripheral device and its driver.

3.2.2 *Model Attributes*

This flavour of energy model annotates device states and driver functions with energy attributes. For device states, it uses average power and, in case the peripheral automatically leaves the state after a while, duration. Function calls always have average power and duration.

While device datasheets often specify some of these attributes, those are rarely complete or accurate and therefore only suitable for rough estimates [ZO13]. For higher accuracy, engineers must perform energy measurements that exercise all hardware states and function calls with a suitable sub-set of configurations, and use these to build an energy model. They can do so by hand, or feed a variability model (including hardware states, function calls, and run-time configuration variables) into an automated benchmarking toolchain.

My own work relies on manually specified configuration-aware state machines as variability models. A machine learning algorithm adds configuration-dependent power and duration annotations to each state and transition, and thus transforms the variability model into an energy model: a parameterized priced timed automaton. This method builds upon ideas first presented by Robert Falkenberg and was partially co-developed with Markus Buschhoff [Fal14; Fri17].

3.3 PARAMETERIZED PRICED TIMED AUTOMATA

Before looking into configuration variables and energy annotations, let us start with the underlying automata structure. Each pair of hardware component and device driver has states and transitions. States are typically defined in the device datasheet, and may be documented alongside driver implementations as well – for instance, many peripheral devices need to be initialized before proper operation and can only execute a limited sub-set of functions in low-power deep sleep modes, and drivers need to keep track of that. Transitions are either caused by driver function calls, or signalled using interrupts that are handled by driver functions. Thus, given a driver and hardware documentation, anyone can build a model that describes hardware states and transitions.

Definition 3.3.1 In this thesis, a *Deterministic Finite Automaton* (DFA) is a tuple (Q, Σ, δ) consisting of a set of hardware states Q, a set of driver functions and interrupts Σ, and a transition function $\delta : Q \times \Sigma \rightharpoonup Q$. The initial state is always $q_0 = \text{UNINITIALIZED} \in Q$. We write state names in uppercase, and transition names in camelCase.

In contrast to the common DFA definition, this variant leaves out the set of accepting states F. As there is no limit on the amount of function calls an application may make, the concept of accepting states does not make sense here. From a satisfiability perspective, $F = Q$. Technically, that makes it a *Labelled Transition System* rather than an automaton. However, as automata and state machine terminology is widely used in the energy modeling literature, DFA is a better fit in this context.

When the operating system starts, the initial hardware state is unknown, hence $q_0 = \text{UNINITIALIZED}$. Adding configurable features

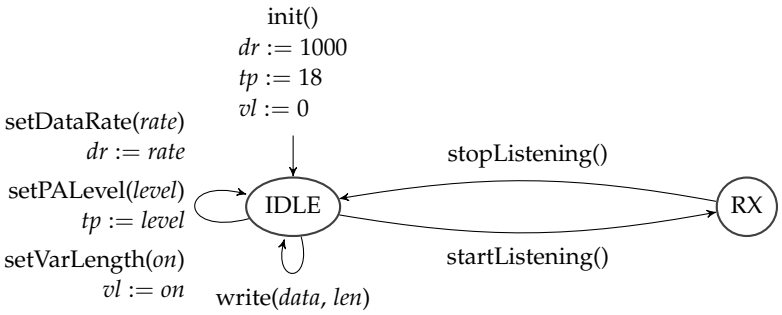


Figure 3.2: PFA model excerpt for an nRF24L01+ radio transceiver with configurable data rate (*dr*), transmit power (*tp*), and variable length packets (*vl*).

(also known as *parameters* in this context) turns the DFA into a parameterized finite automaton that is able to track how function calls affect the run-time configuration of the peripheral and its driver.

Definition 3.3.2 A *Parameterized Finite Automaton* (PFA) extends a DFA (Q, Σ, δ) to a tuple $(Q, \Sigma, \delta, V, \Delta)$. The set V describes the configurable variables (*features* or *parameters*) of the device driver, and the relation $\Delta : \Sigma \times V \rightharpoonup \{\text{const}, \text{arg}\} \times \mathbb{R}$ describes how function calls affect them. Given a function $\sigma \in \Sigma$ and a feature $v \in V$, if $\Delta(\sigma, v) = (\text{const}, x)$, then calling σ sets the value of variable v to x. If $\Delta(\sigma, v) = (\text{arg}, i)$, calling σ sets the value of variable v to the value of the i-th argument of the function σ.

Note that this model draws a clear line between device state and device configuration. The DFA component is responsible for keeping track of device states, and each state change (i.e., the destination state q' of a transition $q \to q'$) is independent of function arguments and run-time configuration settings. The PFA component keeps track of device configuration and updates it as specified by Δ whenever a driver function is executed.

This way, there is no ambiguity when using a device driver and datasheet to come up with the structure of the corresponding PFA. The cost for this is that I deliberately disregard cases where one might argue that a function should lead to different hardware states depending on its arguments and run-time feature configuration. We will see an example for such a case in the BME680's *setPowerMode* and *setSensorMode* functions in Section 3.6.2. Later on, this thesis will show that disregarding whether function arguments affect destination states does not adversely affect energy model accuracy when using appropriate modeling methods.

Fig. 3.2 shows an excerpt of a PFA model for an nRF24L01+ radio chip, leaving out features and functions related to radio channel selection and automatic retransmission for brevity. This leaves it with $Q = \{\text{UNINITIALIZED, IDLE, RX}\}$, $\Sigma = \{\text{init, idle, setDataRate,}$

setPALevel, setVarLength, write, startListening, stopListening$\}$, $V = \{dr, tp, vl\}$, and functions δ and Δ as shown.

The init function sets a default data rate and transmit power and disables variable length packets. setDataRate, setPALevel and setVarLength change data rate, transmit power, and variable length support, respectively. The write function transmits a packet with a specific payload length and blocks until transmission is complete, whereas startListening and stopListening change between IDLE and RX. Functions other than stopListening cannot be called in the RX state.

A PFA expresses the driver's run-time variability, and is sufficient for automatic generation of benchmark programs. However, it has no notion of power or energy. Adding those requires incorporating the concepts of priced timed automata [VS08]. The resulting model is a parameterized priced timed automaton [BFS18].

Definition 3.3.3 A *Parameterized Priced Timed Automaton* (PPTA) extends a PFA $(Q, \Sigma, \delta, V, \Delta)$ to a tuple $(Q, \Sigma, \delta, V, \Delta, P, t)$. The power function P and duration function t associate each state in Q and function call in Σ with a performance prediction model that maps driver configuration and function call arguments to predicted power and duration values. The function signatures are $P : (Q \cup \Sigma) \to ((\mathbb{R} \cup \{\bot\})^n \to \mathbb{R}_{\geq 0})$ and $t : (Q \cup \Sigma) \rightharpoonup ((\mathbb{R} \cup \{\bot\})^n \to \mathbb{R}_{\geq 0})$, with n referring to a state- and function-dependent number of configurable variables. For states, $n = |V|$, and for functions with n' numeric arguments, $n = |V| + n'$.

The model function t is defined for all function calls, and for states that are left automatically by the hardware, such as a TX state once transmission is complete. These state changes are typically signalled using interrupts and therefore accessible for automated benchmarks.

Note that this PPTA definition is far from feature-complete: it deliberately leaves out aspects such as polymorphic functions in C++ drivers, non-deterministic behaviour (e.g. successful transmission versus transmission timeout), and non-numeric arguments. While I have successfully experimented with modeling non-deterministic behaviour and non-numeric arguments, I do not consider these features to be interesting from a performance modeling point of view. Hence, for the sake of concise PFA and PPTA definitions, I leave them out here.

With a PPTA, developers can simulate application interactions with the device driver by means of *timed words*, and calculate energy or average power. For instance, a valid timed word for the model excerpt shown in Fig. 3.2 is $(\text{init}, 0) \cdot (\text{startListening}, 10) \cdot (\text{stopListening}, 12)$. It indicates that the initialization function (causing a switch to the IDLE state) is called at timestamp 0, the startListening function (causing a switch to RX) at timestamp 10, and the stopListening function (back to IDLE) at timestamp 12. The feature vector $\vec{x}$ starts out undefined ($\vec{x} = \vec{\bot}$) and is changed by individual function calls, as described by

Δ. The energy spent in this twelve-second snapshot, excluding the stopListening function call, is calculated as follows.

$$\begin{aligned} E =& P(\text{init})(\{\bot,\bot,\bot\}) \cdot t(\text{init})(\{\bot,\bot,\bot\}) \\ &+ P(\text{IDLE})(\{1000,18,0\}) \cdot (10 - t(\text{init})(\{\bot,\bot,\bot\})) \\ &+ P(\text{startListening})(\{1000,18,0\}) \cdot t(\text{startListening})(\{1000,18,0\}) \\ &+ P(\text{RX})(\{1000,18,0\}) \cdot (2 - t(\text{startListening})(\{1000,18,0\})) \end{aligned}$$

This thesis focuses on automatic generation of performance prediction models P and t for each state and transition. Automatic model generation relies on energy measurements that exercise a suitable subset of all states, transitions, and configurations. Energy measurement automation, in turn, relies on automatic generation and execution of energy benchmarks.

3.4 BENCHMARK GENERATION

An *energy benchmark* is an application that runs on an embedded microcontroller and interfaces with the peripheral under test. It may run as a stand-alone binary image or as part of an embedded operating system; in the latter case, the operating system should not introduce noise due to unrelated timer interrupts or execution of concurrent tasks. At its core, an energy benchmark must contain a valid sequence of driver function calls that exercises each hardware state and driver function with a suitable amount of different configurations.

For a PFA $\mathcal{A}$, the language $L(\mathcal{A})$ contains almost that: a set of all words accepted by $\mathcal{A}$, which by construction is the set of all valid function call sequences. For example, the PFA in Fig. 3.2 accepts the word init() · setDataRate(*rate*) · write(*data*, *len*) · startListening(). However, $L(\mathcal{A})$ is infinite and unaware of legal function arguments.

To obtain a finite set of function call sequences, let $L_k(\mathcal{A})$ be the prefix-free subset of $L(\mathcal{A})$ where each word visits each state no more than k times. With $\#_q(w)$ defined as the number of times a word w visits the state q, the formal definition is:

$$\begin{aligned} L_k(\mathcal{A}) = \{ & w \in L(\mathcal{A}) \mid \forall q \in Q : \\ & (\#_q(w) \leq k \wedge \nexists w' : (w \cdot w') \in L(\mathcal{A}) \wedge \#_q(w \cdot w') \leq k)\} \end{aligned}$$

For a user-specified k, the language $L_k(\mathcal{A})$ contains a finite, prefix-free set of function call sequences that do not visit any hardware state more than k times. While $L_k(\mathcal{A})$ being prefix-free is not strictly required, it helps reduce benchmark runtime by eliminating redundancy. For example, both the word shown above and the word init() · setDataRate(*rate*) · write(*data*, *len*) exercise the states IDLE and TX –

however, the longer word above also covers RX and is therefore more helpful for model learning.

As each function call sequence in $L_k(\mathcal{A})$ starts in UNINITIALIZED, benchmarks can execute them one after another without regard for the end state of the previous sequence. Thus, a computer program can automatically generate a benchmark application by extending each word $w \in L_k(\mathcal{A})$ with user-provided function arguments, delays between transitions, and measurement equipment-specific synchronization signals that allow for mapping energy measurements to hardware states, function calls, and configuration values. For instance, given $rate \in \{250, 1000, 2000\}$ and a `delay_ms` function to ensure 400 ms of measurement time per state, the word init() · setDataRate(*rate*) results in three benchmark sequences.

- `init(); delay_ms(400); setDataRate(250);`
- `init(); delay_ms(400); setDataRate(1000);`
- `init(); delay_ms(400); setDataRate(2000);`

The original work built in cooperation with Markus Buschhoff relied on XML definitions and Perl scripts for benchmark generation and energy measurement. For this thesis, I define state machines and configuration values in a YAML format that is both human- and machine-readable, and use a set of Python scripts and modules. In both cases, a custom-built device (MIMOSA) performs energy measurements and thus dictates the synchronization method [BGS13]. The implementation details of benchmark execution, synchronization, and post-processing are not relevant here. An in-depth discussion of automated energy measurement methods follows in Section 5.2.

The most important aspect is that at the end, the benchmark associates each state and driver function in each benchmark run with an average power and a duration, and aggregates those and the corresponding configurations into state- and function-specific datasets. Each dataset consists of a set $Y = \{y_1, \ldots, y_m\}$ of power or duration values and corresponding configurations $X = \{\vec{x}_1, \ldots, \vec{x}_m\}$. As energy measurements tend to be noisy, the benchmark measures each configuration $\vec{x}$ several times.

3.5 MODEL LEARNING

The idea for extending a PFA to a PPTA is to process each hardware state, each driver function, and each attribute (power and duration) separately, resulting in one performance model for each state/function and energy attribute. The PPTA then combines the PFA structure and the individual performance models into a single energy model.

In principle, any of the algorithms presented in Section 2.5 is capable of generating performance models for energy attributes. However,

neither least-squares regression nor the regression tree learning algorithms presented there handle undefined values ($\perp$) in feature vectors. To work around this, pre-processing algorithms can remove features that hold undefined values entirely or map undefined values to zero.

Whether one should apply performance prediction models from the product line engineering community to energy model generation is one of the core questions of this thesis (**RQ3**), and we will come back to that in Chapter 6. At this point, let us instead consider learning methods from the energy modeling community.

A traditional energy modeling approach, as examined in Section 3.2.1, would either predict energy attributes using the mean of all associated benchmark observations (ignoring hardware configuration altogether), require a user-specified function template, or be limited to linear regression. In contrast to that, we will now examine the *Unsupervised Least-Squares Regression* (ULS) algorithm that is capable of automatically finding and fitting functions that describe how numeric configuration variables affect hardware behaviour [Fri17; BFS18]. It is specifically tailored towards numeric (and possibly undefined) configuration variables and does not support boolean variables (feature toggles).

We start with a pre-processing step that determines which features affect hardware behaviour (i.e., which features are relevant for modeling), and discards those that do not. This reduces the risk of overfitting and improves model interpretability. Next, ULS performs two steps to build and fit a suitable function template using least-squares regression. It first examines how each relevant feature affects hardware behaviour, i.e., whether the relationship is linear, exponential, or similar. It then builds a function template that expresses the combined effect of these features, and fits it using least-squares regression.

3.5.1 *Identification of Relevant Features*

At its core, determining whether a feature affects hardware behaviour is simple: build two LUT models, one of which ignores the feature in question – if the LUT model that ignores the feature has a higher error than the one that knows about it, the feature is likely relevant. The formal description of this approach builds upon the definitions from Sections 2.5.2 and 2.5.3.

Again, let $S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\}$. We utilize $\mu(S)$ (arithmetic mean, 2.3), $\sigma(S)$ (standard deviation, 2.4) and $Val_i(S)$ (unique values of feature i, 2.6). We also rely on the set of unique configurations $Uniq_X(S)$ (2.10) and training data partitioned by configuration $S_{\vec{x}}$ (2.11), both of which are duplicated below for context. Finally, given a feature vector $\vec{x} = (x_1, \ldots, x_n)$, $\vec{x}_{\setminus i} = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ describes the feature vector that leaves out the i-th feature.

Algorithm 6 Identify the set $I \subseteq \{1, \ldots, n\}$ of relevant features in S.

function RELEVANTFEATURES(S, n)
  $I \leftarrow \emptyset$
  for $i \in \{1, \ldots, n\}$ **do**
    if $\frac{\overline{\sigma}(\mathcal{S}_{\text{const}})}{\overline{\sigma}(\mathcal{S}_{\text{const}\setminus i})} < \frac{1}{2} \wedge |Val_i(S) \cap \mathbb{R}| \geq 3$ **then**
      $I \leftarrow I \cup \{i\}$
  return I

same as (2.10)	$Uniq_X(S) = \{\vec{x} \mid (\vec{x}, y) \in S\}$	(3.2)
without i-th feature	$Uniq_{X\setminus i}(S) = \{\vec{x}_{\setminus i} \mid (\vec{x}, y) \in S\}$	(3.3)
same as (2.11)	$S_{\vec{x}} = \{(\vec{v}, y) \in S \mid \vec{v} = \vec{x}\}$	(3.4)
configuration partitions	$\mathcal{S}_{\text{const}} = \{S_{\vec{x}} \mid \vec{x} \in Uniq_X(S)\}$	(3.5)
without i-th feature	$\mathcal{S}_{\text{const}\setminus i} = \{S_{\vec{x}\setminus i} \mid \vec{x}_{\setminus i} \in Uniq_{X\setminus i}(S)\}$	(3.6)
mean standard deviation	$\overline{\sigma}(\mathcal{S}) = \frac{1}{\lvert\mathcal{S}\rvert} \sum_{S \in \mathcal{S}} \sigma(S)$	(3.7)

The set $\mathcal{S}_{\text{const}}$ contains one partition $S_{\vec{x}}$ for each unique feature vector $\vec{x}$ present in the observations S. Hence, $\overline{\sigma}(\mathcal{S}_{\text{const}})$ refers to the mean standard deviation of partitions with constant configuration – i.e., the underlying noise or uncertainty of measurements. $\overline{\sigma}(\mathcal{S}_{\text{const}\setminus i})$ refers to the same, but pretends that the i-th feature does not exist, i.e., it is not constant in the partitions of $\mathcal{S}_{\text{const}\setminus i}$. So, if $\overline{\sigma}(\mathcal{S}_{\text{const}\setminus i})$ is larger than $\overline{\sigma}(\mathcal{S}_{\text{const}})$, the i-th feature likely affects the measured performance property. However, attempting to find and fit a regression function for it only makes sense if it takes a sufficient amount of unique values to distinguish between different function templates.

Algorithm 6 combines these considerations into a heuristic that builds a set $I \subseteq \{1, \ldots, n\}$ of relevant features. For each feature $i \in \{1, \ldots, n\}$, it checks if ignoring it increases the measurement uncertainty (mean standard deviation) by a factor of at least two and if the feature takes at least three different unique values. If both checks pass, the feature is likely relevant for the modeled energy attribute and suitable for least-squares regression, and thus added to the set I.

3.5.2 *Unsupervised Least-Squares Regression*

Let I be the set of relevant features identified by RELEVANTFEATURES in the previous step. If $I = \emptyset$, no feature has an effect on the energy attribute and the constant function $\vec{x} \mapsto \mu(S)$ is sufficient for modeling. Otherwise, the goal is to find and fit a function that describes how each feature in I affects the energy attribute. This consists of two steps:

Algorithm 7 Find the best-fitting function template for the influence of x_i in observations S, using a set of function templates G.

function FINDTEMPLATE(S, G, i)
  for $g \in G$ **do**
    for $\vec{x} \in S$ **do**
      if $g(x_i) = \perp$ **then** ▷ Input domain is violated
        $G \leftarrow G \setminus \{g\}$
  if $G = \emptyset$ **then**
    return $\perp$
  for $g \in G$ **do**
    $Z_g \leftarrow 0$
    $Z_\mu \leftarrow 0$
    for $S' \in \mathcal{S}_{\text{const}\setminus i}$ **do**
      fit $f(\vec{x}) = \beta_0 + \beta_1 g(x_i)$ on S'
      $Z_g \leftarrow Z_g + SSR(f, S')$
      $Z_\mu \leftarrow Z_\mu + SSR(\vec{x} \mapsto \mu(S'), S')$
  $g \leftarrow$ **argmin**(g, Z_g) ▷ g has lowest SSR
  if $Z_g \geq Z_\mu$ **then**
    return $\perp$ ▷ No suitable template in G
  return $\vec{x} \mapsto g(x_i)$

finding a suitable function for each individual feature, and building and fitting a composite function that takes all features into account.

For the first step, the algorithm relies on a built-in set of candidate functions G. These reflect the observation that the energy influence of configuration options often follows simple relations. For instance, screen brightness tends to have a linear influence on power usage, data rate and transmission time are inversely proportional, and transmit power configuration often has a square effect on actual transmit power.

If needed, users can replace or extend G in order to use ULS in domains that its built-in functions were not intended for. In this thesis, all algorithm invocations use the following set of function templates.

$$G = \{x \mapsto x, x \mapsto \ln(x), x \mapsto \ln(x+1), x \mapsto e^x, \\ x \mapsto x^2, x \mapsto x^{-1}, x \mapsto x^{\frac{1}{2}}\}$$

The ULS algorithm first looks at the influence of features in isolation. For each feature $i \in I$, it determines the function template $g_i \in G$ that best predicts how x_i affects the observed energy attribute, and returns a set $F = \{g_{i_1}, g_{i_2}, \dots\}$ with one function template per feature in I. Algorithm 7 describes the method in detail.

The set F describes how individual configuration variables affect device behaviour, but it is not clear how they interact. Consider, for instance, a radio transceiver with configurable data rate (dr) and payload length (pl). Its transmissions start with a fixed-length preamble,

Algorithm 8 Build and fit an n-dimensional prediction function on the set of observations S, using the set of function templates G.

function BUILDULS(S, G, n)
  $F \leftarrow \emptyset$
  for $i \in$ RELEVANTFEATURES(S, n) **do**
    if FINDTEMPLATE(S, G, i) $\neq \perp$ **then**
      $F \leftarrow F \cup \{$FINDTEMPLATE(S, G, i)$\}$
  if $F = \emptyset$ **then**
    return $\vec{x} \mapsto \mu(S)$
  fit $f(\vec{x}) = \sum_{F' \in \mathcal{P}(F)} \left(\beta_{F'} \cdot \prod_{f \in F'} f(\vec{x}) \right)$ on S
  return f

followed by the variable-length payload. In this case, transmit duration neither depends on data rate alone ($\frac{1}{dr}$) nor on data rate over payload length ($\frac{pl}{dr}$). Instead, it is a function of both: $\beta_0 + \beta_1 \frac{1}{dr} + \beta_2 \frac{pl}{dr}$. Hence, ULS builds a function that accounts both for individual and combined influence of configuration variables by using the power set $\mathcal{P}(F)$:

$$f(\vec{x}) = \sum_{F' \in \mathcal{P}(F)} \left(\beta_{F'} \cdot \prod_{f \in F'} f(\vec{x}) \right) \tag{3.8}$$

It fits this function template on all measurements in S (see Algorithm 8), and uses the resulting function including the regression variables $\beta_{F'}$ as model function for performance prediction. Thus, it annotates each pair of hardware state / driver function and energy attribute with a prediction function.

3.5.3 *Example*

To illustrate how the algorithm works, let us go back to the nRF24L01+ radio transceiver shown in Fig. 3.2. The write function's power consumption is likely affected by device configuration, and therefore a good candidate for demonstrating ULS. For simplicity, we will only examine variability in the data rate (dr) and transmit power (tp) variables, and disable variable-length packets (vl) in all measurements.

The learning algorithm starts out with configurations X and corresponding power measurements Y. Each feature vector $\vec{x}$ has one entry for data rate (dr), transmit power (tp), and variable length (vl). Data rate is 250, 1000, or 2000 kbit/s; transmit power is 0, 6, 12 or 18. These unit-less numbers correspond to a power amplifier output power of -18, -12, -6 and 0 dBm, but use offset (and, thus, non-negative) values within the driver API. Variable length is always 0.

Hence, $\mathcal{S}_{\text{const}\setminus dr}$ contains four partitions with variable data rate and constant transmit power, and $\mathcal{S}_{\text{const}\setminus tp}$ contains three partitions with variable transmit power and constant data rate. The set $\mathcal{S}_{\text{const}\setminus vl}$

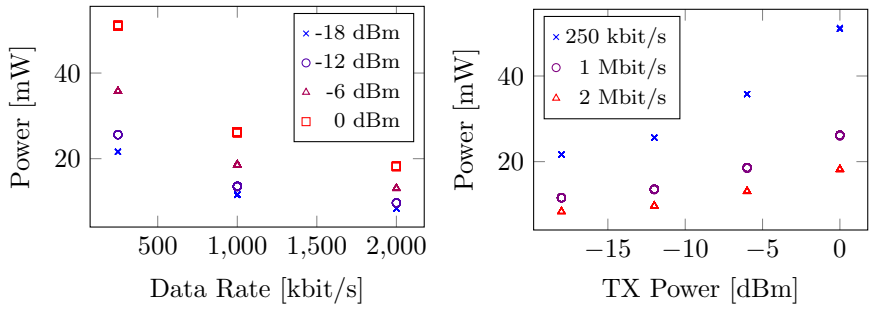


Figure 3.3: Feature configuration versus power usage in the partitions $\mathcal{S}_{\text{const}\setminus dr}$ (left) and $\mathcal{S}_{\text{const}\setminus tp}$ (right) for nRF24L01+ benchmark data.

contains twelve partitions with variable data rate and variable transmit power – so the same partitions as $\mathcal{S}_{\text{const}}$, just without the *vl* feature. Fig. 3.3 illustrates $\mathcal{S}_{\text{const}\setminus dr}$ and $\mathcal{S}_{\text{const}\setminus tp}$. Note that there are several measurements for each configuration; the measurement noise is so low that it is barely visible in the plots.

For identification of relevant features, we already see that both *dr* and *tp* affect the measured average power, whereas *vl* does not (as it is not variable to begin with). Algorithm 6 comes to the same conclusion, with $\frac{\overline{\sigma}(\mathcal{S}_{\text{const}})}{\overline{\sigma}(\mathcal{S}_{\text{const}\setminus dr})} \approx 0 < \frac{1}{2}$, $\frac{\overline{\sigma}(\mathcal{S}_{\text{const}})}{\overline{\sigma}(\mathcal{S}_{\text{const}\setminus tp})} \approx 0 < \frac{1}{2}$, and $\frac{\overline{\sigma}(\mathcal{S}_{\text{const}})}{\overline{\sigma}(\mathcal{S}_{\text{const}\setminus vl})} = 1 \geq \frac{1}{2}$. Hence, $I = \{dr, tp\}$.

Now, the task is to find functions that predict how data rate and transmit power configuration affects average power consumption. Here, Algorithm 8 returns $F = \{g_{\text{dr}}, g_{\text{tp}}\} = \{\vec{x} \mapsto \log(x_{\text{dr}}), \vec{x} \mapsto x_{\text{tp}}^2\}$. This matches the functions that a viewer of Fig. 3.3 might expect. With $\mathcal{P}(F)$, the resulting model function is:

$$f(\vec{x}) = \beta_{\varnothing} \cdot 1 + \beta_{g_{\text{dr}}} \log(x_{\text{dr}}) + \beta_{g_{\text{tp}}} x_{\text{tp}}^2 + \beta_{g_{\text{dr}}, g_{\text{tp}}} \log(x_{\text{dr}}) x_{\text{tp}}^2$$

ULS then fits the function on all observations in S using ordinary least-squares regression. The result is the following model function $f((x_{dr}, x_{tp}, x_{vl}))$ for predicting the µW power usage of the nRF24 driver's write function.

$$f(\vec{x}) = 59058 + 6846 \cdot \log(x_{dr}) + 252 \cdot x_{tp}^2 - 30 \cdot \log(x_{dr}) x_{tp}^2$$

Next, let us look at real-world peripherals and measurements for evaluation purposes.

3.6 EVALUATION TARGETS

Energy modeling evaluations in this thesis use three highly configurable peripheral components as targets: an environmental sensor (Bosch BME680) and two radio transceivers (TI CC1200 and Nordic

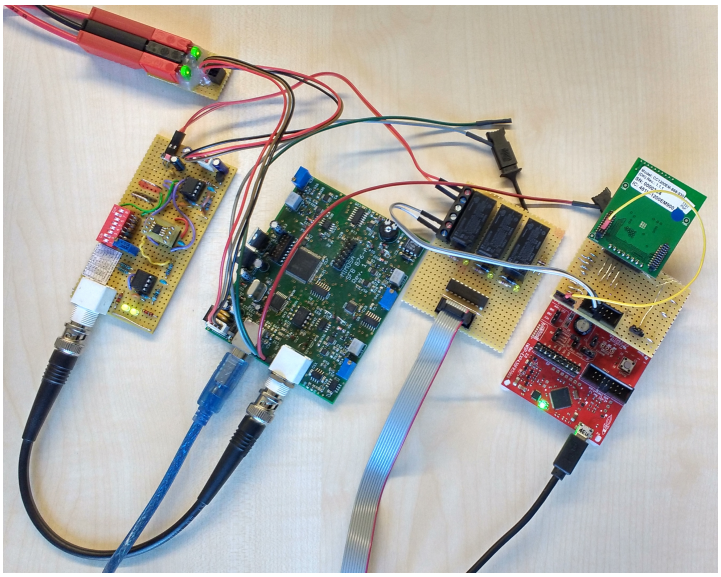


Figure 3.4: A MIMOSA energy measurement setup. Components (left to right): voltage supply and shunt board, integrator, calibration board, and microcontroller for benchmark execution with DUT.

nRF24L01+). Benchmark automation and data acquisition rely on the MIMOSA measurement device and the *dfatool* framework that will be presented in Section 5.2. We will now examine MIMOSA and the three peripherals in detail.

3.6.1 *MIMOSA*

MIMOSA[1] is a measurement device that has been specifically designed for automated measurements of low-power embedded components [BGS13]. It uses a current mirror and hardware integrators to track the current flowing through a *Device Under Test* (DUT), allowing it to observe power consumption spikes even if they are shorter than the sample rate. Built-in voltage drop compensation ensures that the voltage fed to the DUT is constant and not affected by the voltage drop in the shunt resistor used for current measurement.

MIMOSA provides 16-bit current measurements with a configurable upper limit of 2.4 to 50 mA, giving a resolution of 41 to 869 nA per bit. Its 100 kHz sample rate includes analog current readings as well as the state of a digital synchronization input that can be connected to a *General Purpose Input/Output* (GPIO) pin of a microcontroller. Combined with an automatically-generated benchmark program that toggles this GPIO pin at pre-defined points in time and logs function calls via UART, this helps with automated analysis of bench-

1 MIMOSA is a German acronym for "Messgerät zur integrativen Messung ohne Spannungsabfall", i.e., measurement device for integrative measurements without voltage drop.

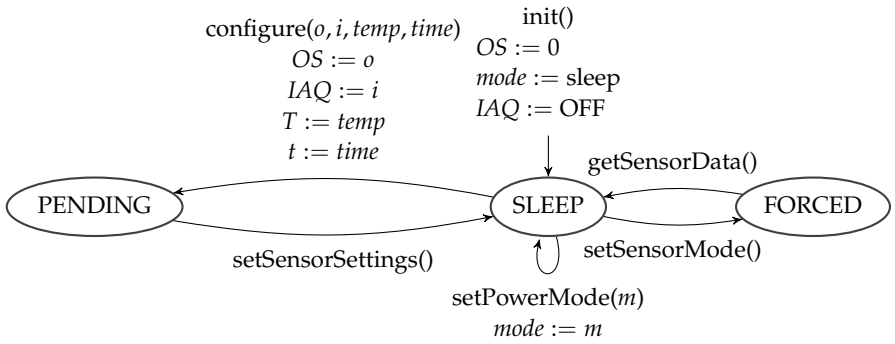


Figure 3.5: PFA model for a BME680 environmental sensor with configurable pressure oversampling (*OS*), optional air quality measurement (*IAQ*), power mode (*mode*), heater temperature (*T*), and heater time (*t*).

mark results. Fig. 3.4 shows a typical measurement setup, using a TI MSP430FR5969 microcontroller for benchmark execution and a TI CC1200 radio transceiver as device under test.

Each pair of consecutive MIMOSA samples is 10 µs apart. This has implications for short driver function calls: A transition that takes 15 µs to complete will be measured as either 10 or 20 µs depending on its microsecond timing, and a model that correctly predicts its 15 µs duration will always be off by 5 µs (so, more than 30 %). As this error is caused by measurement artifacts rather than model deficiencies, this thesis leaves out power and duration measurements for transitions with a maximum duration of 100 µs or less.

It also leaves out the UNINITIALIZED state and the init function when working with energy measurements. These operate on undefined hardware states where configuration attributes suitable for performance prediction are not available. They are also not interesting, as they typically do not occur once an embedded system has finished its startup. With this in mind, let us look at our devices under test.

3.6.2 *BME680 Environmental Sensor*

Bosch SensorTec's BME680 is a low-power sensor for measuring temperature, humidity, air pressure, and air quality. The latter works by means of a metal oxide plate whose resistance, when heated to several hundred degrees Celsius, is affected by the presence of pollutants in the air. Bosch SensorTec provides an open-source BME680 driver[2] and lists device states in the documentation; Fig. 3.5 shows the corresponding variability model. The measurements in this thesis build upon a C++ port of the driver, as this works best with my automatic benchmark generation and execution framework.

2 https://github.com/boschsensortec/BME68x_SensorAPI

The sensor starts out in a low-power SLEEP mode, and does not perform any measurements by default. Applications can configure it using *configure*, *setSensorSettings* and *setPowerMode*, call *setSensorMode* to start a single measurement (FORCED mode), and use *getSensorData* to read its results.

There is a discrepancy between driver states and hardware states. The driver does not apply configuration changes made with *configure* automatically, but only does so upon an explicit call to *setSensorSettings*. Hence, the PFA model has a PENDING state that reflects the state machine inside the driver API rather than a separate hardware state.

The driver also has special behaviour when it comes to the interaction between hardware states and configuration variables. The function *setSensorMode* sets the operating mode of the sensor to *mode*, which is either SLEEP or FORCED. On the hardware side, the sensor only transitions into FORCED mode if *mode* tells it to. However, in the PFA model, *setSensorMode* always leads to the FORCED state and it is up to the PPTA energy model to handle the influence of *mode* on the state's average power.

As soon as it has performed a measurement, the sensor automatically returns from FORCED to SLEEP mode – however, it neither signals this using an interrupt, nor does it happen after a constant time. Instead, measurement duration depends on device configuration, and the manufacturer recommends that applications using the driver estimate it, wait, and then poll the sensor state until the measurement is complete. To avoid energy-intensive polling loops, the benchmarks here instead wait for the (constant) maximum time that FORCED mode can be active. They treat the *getSensorData* function call as a return to SLEEP mode; this function transfers measurement results from the device to the driver and should be called whenever FORCED mode was active.

Configurable features include the oversampling rate of temperature, humidity and pressure measurements, heater temperature and heat duration, and whether air quality measurements are enabled. The benchmarks here do not configure temperature and humidity oversampling to allow for less time-intensive benchmark runs.

Fig. 3.6 shows the variance of measurement results, excluding the *configure* and *setPowerMode* functions due to a duration of less than 20 µs. FORCED and *getSensorData* exhibit notable variance, and the duration of *setSensorSettings* and *setSensorMode* is far from constant as well. The few dozen microwatts worth of outliers present in SLEEP and PENDING are small compared to the power deviations in other states and transitions, most notably FORCED and *getSensorData*. So, the sensor has both energy attributes that may benefit from configuration-aware performance prediction models, and attributes that are likely not affected by its configuration. For energy model evaluation, FORCED

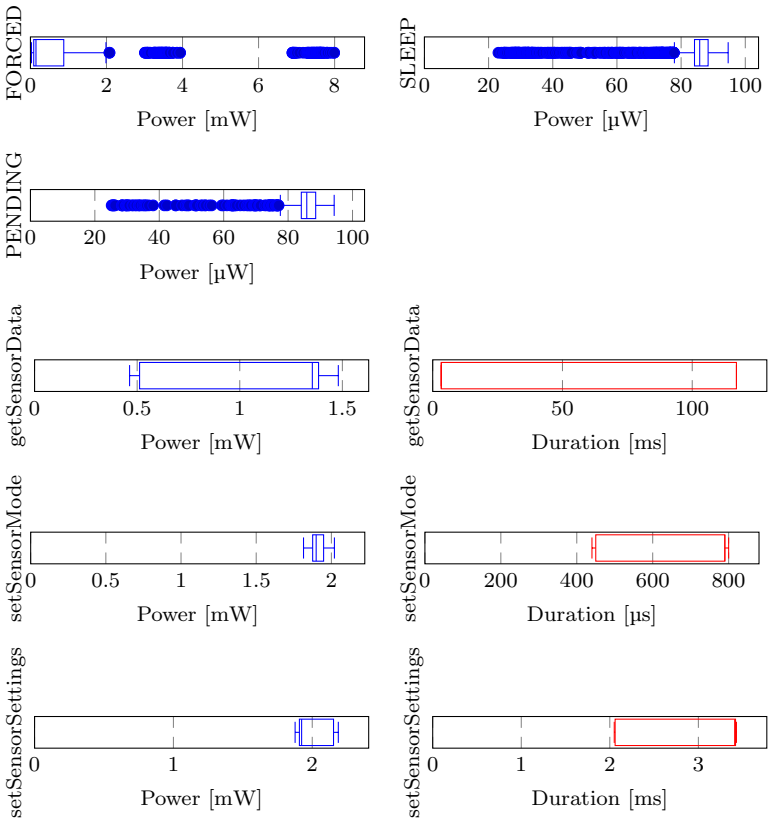


Figure 3.6: Observed power and duration of BME680 states and transitions in benchmark results.

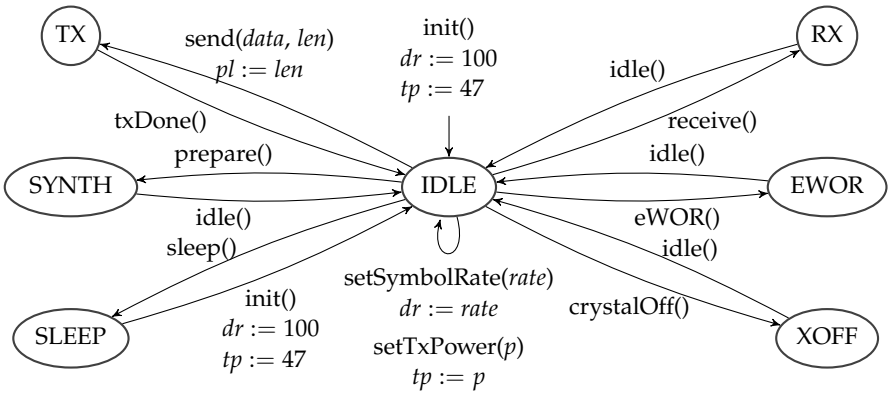


Figure 3.7: PFA model for a CC1200 radio transceiver with configurable payload length (*pl*), data rate (*dr*), and transmit power (*tp*).

power, *getSensorData* power and duration, and *setSensorSettings* duration are most interesting.

3.6.3 *CC1200 Radio Transceiver*

The Texas Instruments (TI) CC1200 is an embedded radio transceiver for the sub-GHz ISM band that was used in the first iteration of the *Solar Doorplate*, a demonstrator for a wireless network of energy harvesting-powered smart doorplates developed at TU Dortmund [Bus19]. It features a variety of sleep and intermediate power states and low-level control over modulation and other radio attributes. In addition to standard IDLE, SLEEP, RX (receive) and TX (transmit) states, it supports EWOR, SYNTH, and XOFF modes.

EWOR (wake on radio) is a sleep mode in which the transceiver regularly wakes up to check for radio traffic, and emits an interrupt if it detects any. This allows the microcontroller to remain in a low-power sleep mode rather than having to do these checks by itself. SYNTH (synthesizer standby) is an idle mode in which the frequency synthesizer is turned on in anticipation of radio transmissions, allowing the radio to switch to RX/TX mode faster than from the regular IDLE mode at the cost of higher idle power consumption. XOFF (crystal off), on the other hand, is an idle mode in which the chip's processor clock has been turned off. In contrast to SLEEP, all register values are retained, and return to idle takes less time. The CC1200 user guide contains a detailed description of these hardware states, including a state machine model [Ins13].

Fig. 3.7 shows the transitions and configuration options of the corresponding device driver. Data transmission (TX) is non-blocking; a *txDone* interrupt signals transmission completion or errors. For simplicity, the driver model only considers transitions to and from IDLE, and leaves out transitions between non-idle states. Configurable variables are *payload length*, *data rate*, and *transmit power*. The driver

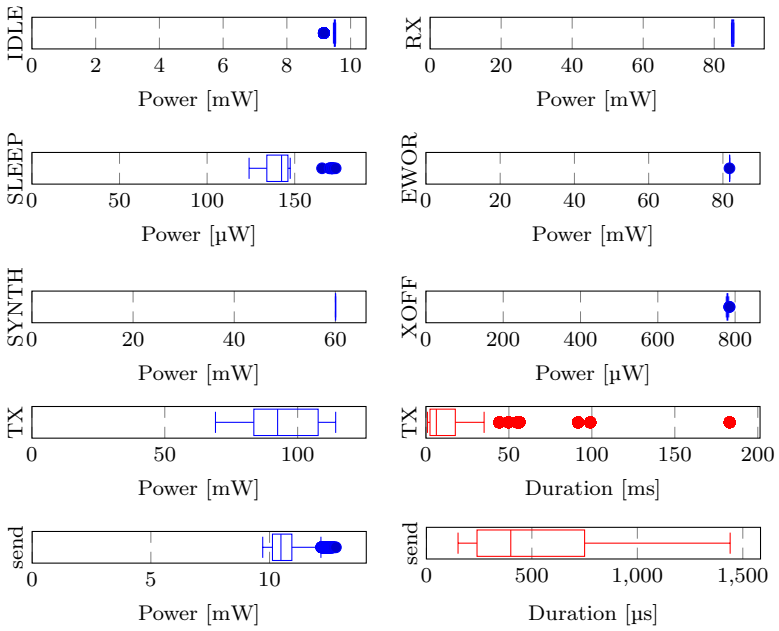


Figure 3.8: Observed power and duration of CC1200 states and transitions in benchmark results.

does not support additional configuration options, e.g. for wake-on-radio operation.

Fig. 3.8 shows the variance of measurement results. *send* is the only function call that takes more than 100 µs to complete, so it is the only one relevant for model generation. The wake-on-radio EWOR state can be much more efficient, in principle, but the driver used here does not provide the configuration options needed to achieve that. For energy model evaluation, TX power and duration as well as *send* duration are most interesting.

3.6.4 *nRF24 Radio Transceiver*

Nordic Semiconductors' nRF24L01+ is a low-cost 2.4 GHz radio transceiver with packet-based operation that was used in the second Solar Doorplate iteration [Bus19].

Similar to CC1200, it provides STANDBY and RX modes. In the driver used here[3], there is no TX state, however – instead, sending a packet is a blocking operation, so the entire transmission process is part of the *write* transition. While it does have a SLEEP mode, power consumption in that state is below the usable measurement range of MIMOSA and thus not included here.

In contrast to CC1200, the nRF24 driver supports packet operation with automatic acknowledgment of received packets and automatic retransmission of packets that were not acknowledged. For this to

3 https://github.com/nRF24/RF24

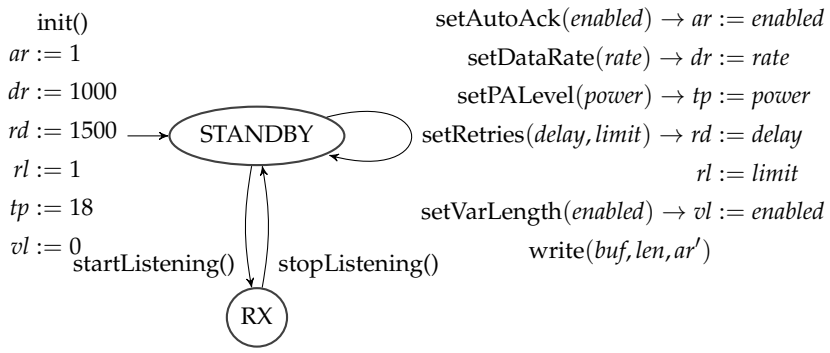


Figure 3.9: PFA model for an nRF24 radio transceiver with configurable automatic retransmission (*ar*), data rate (*dr*), retransmit delay (*rd*), retransmit limit (*rl*), transmit power (*tp*), and variable payload length support (*vl*).

work, the automatic retransmission feature must be enabled on sender and receiver, which users can do globally (*setAutoAck*) or on a per-transmission level with the ar' argument to the *write* function. Retransmission delay and limit (maximum number of retransmissions) are configurable as well.

The driver uses 32-Byte packets by default, but can be configured for variable-length payloads. So, just like BME680, it has a variety of boolean and numeric configuration options. Fig. 3.9 shows the corresponding PFA model.

I expect dynamic payload size to affect whether *write* duration depends on payload length or not, and auto-ack to determine whether related settings affect *write* duration or not. Note that payload length is a function argument and not a configurable driver feature here: as *write* is a blocking function, there is no TX state and thus no need to keep track of the payload length for TX power and duration prediction.

Fig. 3.10 shows the variance of measurement results. Most function calls take longer than their CC1200 counterparts and show little to no influence of configuration variables. The only configuration-dependent components appear to be power consumption and duration of the *write* function, and possibly RX power. Hence, only these three are interesting for energy model evaluation.

3.7 CHAPTER SUMMARY

As we have seen, peripheral components and device drivers are not unlike software product lines: they, too, expose variability that affects timing and energy attributes of hardware states and function calls. In contrast to software product lines, variability models for hardware components have to take interactions between hardware states, function calls, and run-time configuration variables into account.

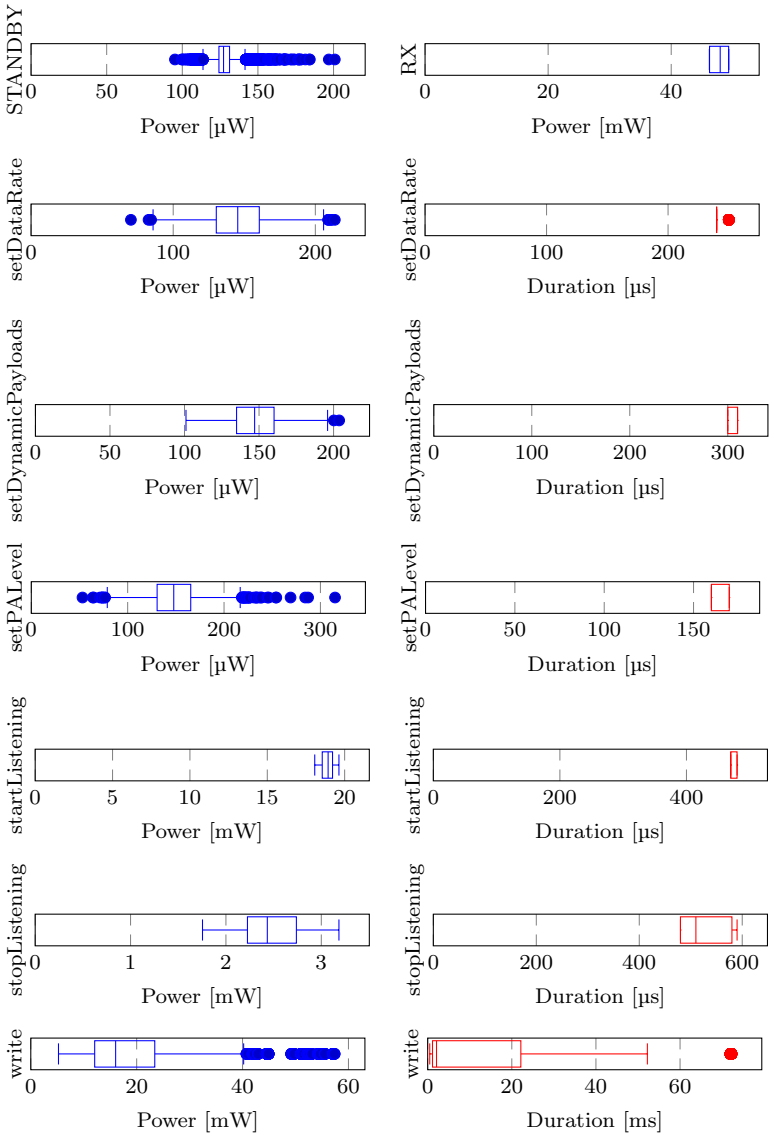


Figure 3.10: Observed power and duration of nRF24 states and transitions in benchmark results.

Even without an explicit reference to product line development, past energy modeling research has used variability models in a similar manner: state machine models capture the interaction between device states and driver functions, and regression templates describe which configuration variables or run-time performance counters affect hardware behaviour.

However, compared to SPLs, there has been little work on the performance modeling aspect. Many existing energy models use constant energy annotations and are therefore unable to predict how configuration variables affect energy requirements. Those that support configuration variables often rely on manually-specified regression templates [Zha+10], or use clustering and mean values for prediction [Che+17] rather than attempting to learn a prediction function.

Unsupervised Least-Squares Regression, a work-in-progress algorithm that I developed before starting work on this thesis, partially resolves this. It can automatically identify relevant features and build suitable function templates for least-squares regression, thus allowing engineers to automatically transform Parameterized Finite Automata into Parameterized Priced Timed Automata. However, in contrast to performance models for software product lines, it does not support boolean feature toggles.

With this in mind, let us now take a closer look at the similarities and differences between performance models for software product lines and energy models for embedded peripherals. I will also use this opportunity to revisit the research questions laid out in Section 1.3 with the background knowledge that we now have.

4

PROBLEM STATEMENT

Non-functional property models for software product lines and energy models for hardware components address the same problem: automatically learning a performance prediction model for a variable system component. However, as the two previous chapters have shown, there are notable differences between the two.

- In software product lines, the main challenge is handling the hundreds to tens of thousands of (mostly boolean) configuration options that modern software exposes [Per+21; Guo+18].
- While hardware components tend to have no more than a dozen (often numeric) run-time configuration variables, models for them must consider the interaction between hardware states and driver functions [Zha+10; HHS19].
- In contrast to software product lines, automatic data acquisition for energy models relies on suitable measurement equipment and benchmark synchronization methods.

Two different scientific communities with little interdisciplinary cooperation work with these two types of performance models: the Software Product Line Engineering (SPLE) community focuses on performance attributes of software components, and the Cyber-Physical Systems and Internet of Things (CPS and IoT) community on energy attributes of hardware components. Both communities make different assumptions about the type and importance of features they handle, and use different methods for variability and performance modeling. Table 4.1 outlines the key differences in variability modeling method, feature type and configuration space size, and typical performance model generation approach.

In my opinion, both communities can – and should – learn from each other, and machine learning algorithms for performance model

SPLE	CPS/IOT
Feature models [ES15; BSE19]	UML [SUP21] or informal + state machines [ZV16; Che+17]
$10 \dots 10^4$, boolean [Tar+11]	$1 \dots 10$, numeric [Har+16]
Regression trees [Guo+18]	Least-squares regression [BFS18]

Table 4.1: Key differences between variability and performance models in the SPLE and CPS/IoT communities.

generation can specifically benefit from both fields. In fact, showing that this is the case is the main contribution of this thesis: in Chapter 7, I will present the *Regression Model Tree* machine learning algorithm and use it to automatically build interpretable performance models for software and hardware components.

Additionally, the journey towards regression model trees provides contributions to the entire performance modeling life cycle, from energy measurements over the formal link between variability models and performance models to model generation and performance-aware product line configuration. All of them prioritize automation: tedious and error-prone manual labour should be avoided wherever possible.

RQ1: ENERGY MEASUREMENT SYNCHRONIZATION

Performance model generation relies on data sets that map software or hardware configuration to performance attributes. As noted in the previous chapter, obtaining these is especially challenging for energy models. Automatic energy measurement requires not just the ability to measure energy usage, but also a way of synchronizing energy readings to benchmark events.

Conventional automation approaches use out-of-band signals for synchronization, which may be unavailable for a variety of reasons. On the one hand, they require hardware that exposes suitable output signals. While this is often the case when working with development prototypes, it may no longer be possible when running energy measurements on a production unit that does not provide access to spare output signals. On the other hand, these automation methods can only be used with measurement equipment that is capable of logging energy readings and synchronization signals at the same time. Such equipment is often expensive, either in terms of money (commercial hardware) or time (do-it-yourself approaches).

Hence, **RQ1**: are automated and accurate CPS/IoT energy measurements feasible on hardware that lacks suitable out-of-band synchronization methods? Here, the main contribution is a generic synchronization and drift compensation algorithm that exclusively relies on on-board timers and in-band signalling. Additionally, I will provide an accuracy analysis of the embedded energy measurement circuit on the affordable ($20 apiece) commercial off-the-shelf MSP430FR5994 evaluation boards that serve as evaluation targets for the algorithm. Both of these topics follow in Chapter 5.

RQ2: LINKING VARIABILITY MODELS AND PERFORMANCE MODELS

Once an engineer has obtained a set $S = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \dots\}$ of configurations and performance measurements, they can employ machine learning to build a performance model. Especially in the field

of software product lines, this raises the question of linking the performance model with the already-present variability model. Learning algorithms can integrate both into a single model by means of feature- and variant-wise annotations (cf. Section 2.2), or keep them separate and use feature vectors for coupling (cf. Section 2.5).

This is a fundamental decision that impacts the expressiveness and flexibility of variability and performance modeling methods, the annotation process during model generation, and also model maintenance as a product line and its implementation components evolve over time. Chapter 6 addresses the corresponding research question (**RQ2**): should performance models be integrated into variability models, or should they be separate entities?

Along this way, we will also examine the relation between hardware components and product lines. Drivers for hardware components are rarely developed as product lines, yet a key element of this thesis is that energy models for hardware components can be treated just like performance models for software product lines. Hence, I will explain the relation between configurable hardware components, state-machine based variability models for them, and product lines.

Finally, we will take a look at the features that variability models work with. As stated earlier, SPLE often focuses on boolean-only features, whereas energy models typically work with numeric (non-boolean) configuration variables. Using the evaluation targets presented in Sections 2.7 and 3.6, we will examine whether this makes sense, or whether variability and performance models should consider both kinds of features.

RQ3: INTERPRETABLE MACHINE LEARNING

After that, we can finally address the core question of this thesis (**RQ3**): can a common machine learning algorithm for SPLE and CPS/IoT performance models provide lower prediction error and model complexity than conventional approaches, without requiring manually provided domain information or model structure? If that is the case, any configurable software or hardware system whose configurations can be formally expressed is suitable for performance model generation. It does not matter how closely it resembles a product line and whether it has been engineered according to SPLE principles.

The answer to this question revolves around my *Regression Model Tree* machine learning algorithm and data structure. In Chapter 7, we will first examine two methods for improving model interpretability. I will then present the Regression Model Tree data structure and the corresponding machine learning algorithm, including a method for detecting co-dependent variables in regression analysis. Regression Model Trees combine the aforementioned methods in a way that improves model interpretability and accuracy, allowing them to achieve

better performance than the sum of their parts. The chapter closes with an evaluation against related machine learning methods.

RQ4: PERFORMANCE MODELS FOR HYBRID PRODUCT LINES

Once variability and feature modeling are taken care of, it is time for a detailed look into case studies and model applications. On the one hand, this provides real-world context for querying and interpreting Regression Model Tree-based performance models. After all, a performance model is not a goal unto itself, but most useful when reasoning about real-world products. On the other hand, so far, we have examined software and hardware components largely in isolation. The case studies contain hybrid product lines with variability in both aspects, and thus serve to answer **RQ4**: are product line engineering and performance modeling techniques also applicable to product lines that cover soft- and hardware variability?

Chapter 8 addresses this by presenting two product lines and analyzing performance models for those. The *resKIL* product line for resource-efficient AI in agricultural machinery combines configurable software with interchangeable hardware components; its performance attributes include AI latency, throughput, and accuracy. The second product line describes the configuration space of data transfer between wireless IoT nodes, with the analysis focusing on how hardware and data serialization format selection affect the energy cost and memory requirements of data transfers.

resKIL also serves as a real-world case study for using and interpreting Regression Model Trees that goes beyond the quantitative evaluation in Chapter 7. This includes manual model analysis as well as performance-aware product line configuration within the *kconfig-webconf* utility that I designed as part of this thesis.

5 DATA ACQUISITION

Before engineers can use machine learning algorithms for performance model generation, they must run benchmarks to obtain sets $S = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \dots\}$ of configurations and corresponding performance attributes. For software product lines, this typically entails building specific configurations and then measuring attributes such as file size or processing throughput. This is an established method; I will give a brief overview of my implementation for automatic performance measurements of Kconfig-based software product lines in Section 5.1.

When working with hardware components, data acquisition becomes more challenging. Here, measurement campaigns rely on benchmark applications to exercise a suitable subset of all hardware states, function calls, and configurations. They must observe energy attributes during benchmark execution, and synchronize them with benchmark events so that they can associate each state and function call with configurations and corresponding average power or duration values. Section 5.2 presents my method for doing so even when out-of-band synchronization signals are unavailable, and thus answers **RQ1**: are automated and accurate CPS/IoT energy measurements feasible on hardware that lacks suitable out-of-band synchronization methods? It includes an accuracy analysis of a $20 apiece commercial off-the-shelf energy measurement circuit that serves as evaluation target.

5.1 BENCHMARKING KCONFIG-BASED SOFTWARE PRODUCT LINES

Obtaining benchmark data for learning performance models of Kconfig-based SPLs or SPL-like software projects consists of four steps for each training sample: Configure the project according to an appropriate configuration space sampling strategy, build a product, measure performance attributes, and store configuration and performance attributes in a database. I have implemented this as part of my *dfatool* performance modeling toolchain[1]. With it, given a Kconfig-based product line or software project, any engineer can automatically obtain benchmark data with a minimum amount of manual work.

The name *dfatool* stems from the toolchain's origin: its first application, and the reason for its inception, was energy measurement automation built upon DFA and PFA models. It uses the resulting benchmark data to extend those into PPTA energy models (see Section

1 https://ess.cs.uos.de/git/software/dfatool

3.3). By now, its feature scope has extended and dfatool has become a name without underlying meaning.

5.1.1 *User-Provided Commands*

The engineer starts with a release or development version of the software project whose performance they want to measure, such as busybox. Before the benchmark process can start, they must implement three commands. For simplicity, I will assume that the project comes with a Makefile and that the engineer implements the benchmark-related commands as make targets. This is not a requirement: all commands are configurable, and the engineer may also use shell scripts or other build, configuration and measurement commands instead.

- `make nfpkeys` must output a JSON dictionary that lists the relevant performance attributes with human-readable descriptions, units, and optimization goals (i.e., whether an attribute should be minimized or maximized).
- `make randconfig` must generate a random configuration and write it to the `.config` file. In many cases, it is sufficient to call `kconfig-conf --randconfig Kconfig` for this.
- `make nfpvalues` must output a JSON dictionary that maps each performance attribute to a measurement that corresponds to the current system configuration.

Examples are available in the dfatool documentation.

5.1.2 *Sampling*

Once that is done, the engineer can start exploring the configuration space using dfatool's `explore-kconfig.py` command. Like many other performance modeling approaches for SPLs, dfatool uses random sampling [Per+21]. Optionally, it can randomize numeric variables on its own, and explore the neighbourhood of random or user-provided configurations. For each configuration, neighbourhood exploration additionally benchmarks all configurations in which only a single variable has a different value, using five distinct values for numeric variables (see Section 2.7.2 for details). Randomization of numeric variables and neighbourhood exploration rely on the `Kconfiglib` parser[2] to determine the type and (if numeric) range of configuration variables and to read/write `.config` files.

At runtime, explore-kconfig first stores performance attribute descriptions (`make nfpkeys`) in a benchmark database. Next, it generates

2 https://github.com/ulfalizer/Kconfiglib

(`make randconfig`, if random sampling is enabled) and builds (`make`) configurations, and stores each configuration (`.config` file contents) and the corresponding performance attributes (`make nfpvalues`) in the benchmark database.

For example, the command `explore-kconfig.py --random 2000 --random-int .../Kconfig` benchmarks 2,000 random configurations, while ensuring that numeric features are randomized as well. The latter is important when implementing `randconfig` with commands such as `kconfig-conf --randconfig`, which only randomizes boolean features.

In `--random` mode, explore-kconfig also checks for build success and generates a new random configuration when encountering a build failure. This way, it always generates the requested number of training samples. While well-designed product lines should not have invalid configurations that do not resemble a working product, dfatool also supports software projects that are not developed according to SPLE principles. There, it may encounter invalid configurations and other variability modeling issues [Tar+11].

dfatool's `analyze-kconfig.py` command transforms this data set into performance models. Engineers can interpret those manually or pass them to kconfig-webconf for performance-aware product line configuration. Details follow in Chapter 7 and Section 8.2, respectively.

5.2 ENERGY BENCHMARK SYNCHRONIZATION

> *Related publication*: Birte Friesel, Lennart Kaiser, and Olaf Spinczyk. "Automatic Energy Model Generation with MSP430 EnergyTrace". In: *Proceedings of the Workshop on Benchmarking Cyber-Physical Systems and Internet of Things*. CPS-IoTBench '21. Nashville, TN, USA: Association for Computing Machinery, May 2021, pp. 26–31. ISBN: 978-1-4503-8439-1. DOI: 10.1145/3458473.3458822 [FKS21]

While performing energy measurements can be as simple as using an oscilloscope and a shunt resistor, manual measurement analysis is a tedious and error-prone task [YF09]. Automation makes it less tedious, but more complex: it relies on test fixtures that measure energy attributes of the target peripheral while keeping track of the active function call or hardware state, thus synchronizing energy measurements to benchmark events. Many approaches use GPIO pins or the *Universal Asynchronous Receiver/Transmitter* (UART) interface to achieve this task.

For the GPIO variant, in the simplest case, the benchmark program toggles a GPIO pin whenever entering or leaving a hardware state. The pin is connected to a digital input of the measurement device, which keeps track of the input level and thus is able to detect state and

transition boundaries. Some approaches use multiple GPIO pins to encode state or transitions identifiers, making the system more robust against non-deterministic behaviour [AH09].

In the UART variant, the benchmark program logs the start and end of transitions between device states on the serial interface, and the measurement device keeps track of those [Lim+13]. However, UART output affects energy usage and timing behaviour of the device under test. While using on-board timers to measure the duration of states and transitions is a partial remedy, UART-based synchronization is still more challenging than using GPIO pins.

Both of these methods rely on the availability of GPIO or UART signals for *out-of-band* synchronization. They cannot be used when the DUT does not expose them or when the measurement equipment cannot capture them. A factor contributing to the latter is that professional measurement equipment that supports out-of-band signalling is often expensive (thousands or even tens of thousands of euros) or has to be built in a do-it-yourself (DIY) manner.

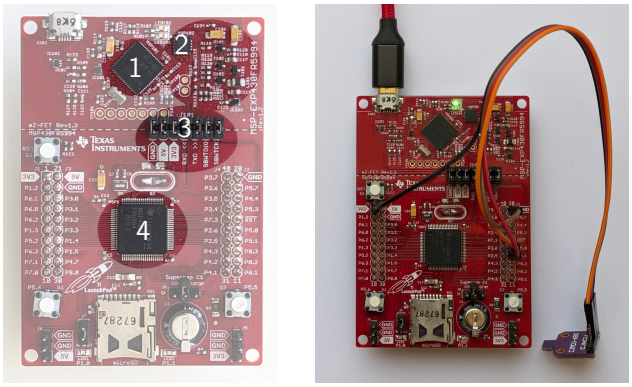
While a DIY solution may be very affordable (tens to hundreds of euros), it is still expensive in terms of the time users have to invest in order to obtain a working and sufficiently accurate measurement device. External factors that require researchers and university students to work from home, such as a global pandemic, further exacerbate this issue: now, everyone must have their own device for performing energy measurements, and accidents could lead to a multi-thousand dollar manufacturer service bill or multi-hour manual repair.

I took this as an opportunity to look into methods for automated energy measurements that exclusively rely on in-band synchronization methods, and are thus compatible with commercial off-the-shelf equipment that is affordable in terms of money and time. The *EnergyTrace* technology embedded in $20 apiece MSP430FR5994 LaunchPad evaluation boards serves as case study and evaluation target.

The next sub-sections explain what EnergyTrace is, how it works, how accurate it is out of the box, and how calibration and drift compensation can further improve its accuracy and automation capabilities. Apart from an in-depth analysis of EnergyTrace, the key contribution is a generic synchronization and drift compensation method that is applicable to any kind of measurement equipment that does not support (or have access to) out-of-band synchronization signals.

5.2.1 *EnergyTrace*

MSP430 LaunchPads are a family of evaluation boards for 16-bit ultra-low-power TI MSP430 microcontrollers. They are commercially available for less than $20 and augment the microcontroller with a programming and debugging interface and, in many cases, the EnergyTrace energy measurement system. All of these components



(a) Host MCU ①, EnergyTrace circuit ②, jumper links ③, and target MCU ④.

(b) Test fixture for automated energy benchmarks on a BME680 air quality sensor.

Figure 5.1: An MSP430FR5994 LaunchPad with its most relevant components annotated (left) and a typical energy measurement setup (right).

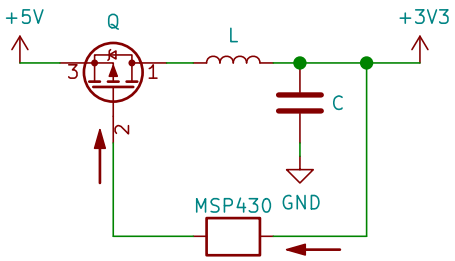


Figure 5.2: Simplified EnergyTrace DC-DC converter schematic [TI16]. The EnergyTrace MCU monitors the output voltage (+3V3) and regulates the inductor L (via transistor Q) in response to it.

are built into the LaunchPad. From a user perspective, EnergyTrace consists of two parts: the control and measurement circuitry and firmware on the LaunchPad, and the client library.

Hardware

An MSP430FR5994 LaunchPad contains three MSP430 *Microcontroller Units* (MCUs): the MSP430FR5994 itself (target MCU), an MSP430F5528 (host MCU) for programming, debugging and USB-to-UART interfacing, and an MSP430G2452 (EnergyTrace MCU) connected to a DC-DC converter that powers target MCU and peripherals [TI16]. This allows it to perform energy measurements of target MCU and peripherals without any external components. Fig. 5.1 shows the location of these microcontrollers on a LaunchPad, and how to connect a BME680 air quality sensor (see Section 3.6.2) for energy measurements. The upper third of the board contains debugging and energy measurement circuitry, and can be decoupled from the target MCU using jumper links. The lower two thirds contain target MCU and peripheral connections.

The DC-DC converter powers the entire bottom part of the board: the target MCU and all peripherals that are connected to its 3.3 V pins (here: the BME680 sensor). It operates as an inductive charge pump (see Fig. 5.2) that monitors the output voltage (+3V3) of a buffer capacitor (C) and uses a switch (Q) to briefly charge an inductor (L) whenever the output voltage drops below a threshold. This allows it to convert USB voltage (4.5 to 5.2 V) to a near-constant output voltage that oscillates in a tight range around 3.3 V.

Conventional DC-DC converters use an integrated circuit for voltage monitoring and switch control. EnergyTrace replaces this IC with the EnergyTrace MCU and can therefore keep track of the frequency and duration of charge pulses (i.e., the time during which Q is connected). Each pulse transfers a known amount of energy, allowing it to calculate the energy usage of target MCU and peripherals.

Note that host MCU and EnergyTrace MCU also require a 3.3 V power supply for operation. This one is not part of the DC-DC converter, but instead provided by a conventional linear regulator that does not provide feedback signals for energy measurement.

Software

TI does not provide source code for the firmware running on host and EnergyTrace MCU; they likely intend EnergyTrace to be used only with their in-house development environments. This is helpful for developers who want to quickly assess the energy behaviour of their application and target hardware, but not so much for automated measurements.

The libmsp430 client library that runs on a desktop or laptop computer and interfaces with a connected LaunchPad is closed-source as well. Still, it provides an API to the EnergyTrace sub-system that can start and stop measurements and accepts a callback function. It calls this function periodically and passes a buffer that contains a variable-length list of EnergyTrace events. Each event i contains

- a 32-bit timestamp T_i (1 µs per least significant bit),
- a 32-bit current I_i (1 nA),
- a 16-bit voltage U_i (1 mV), and
- a 32-bit cumulative sum of energy E_i (100 nJ).

The open-source EnergyTrace CLI project[3] provides a command-line application for energy measurements built on top of this API. It starts a measurement, writes timestamp, current, voltage, and energy data to its standard output, and stops the measurement after a specified time has elapsed or once it receives a termination signal.

3 https://ess.cs.uos.de/git/software/energytrace-util

The cumulative sum of energy is the total amount of energy transferred since the start of the measurement. By default, it also refers to the total amount of energy spent during program execution, as the host MCU resets the target MCU when starting an EnergyTrace measurement. Current and voltage appear to be momentary readings taken at the corresponding timestamp instead. Considering that the EnergyTrace hardware measures voltage and energy, but not current, I_i is likely calculated by the firmware on the LaunchPad or by the libmsp430 client on the computer.

For PPTA generation, total energy is not helpful – instead, we need a series of power values that we can map to hardware states and function calls. With the provided EnergyTrace events, there are two ways of calculating those: using momentary values ($P_i = U_i \cdot I_i$), or by looking at average power in the interval $[T_{i-1}, T_i]$. The latter relies on interval duration $\Delta T_i = T_i - T_{i-1}$ and per-interval energy consumption $\Delta E_i = E_i - E_{i-1}$. As we do not know the EnergyTrace-internal relation between event contents, it will be interesting to see whether these two approaches lead to identical results.

$$P_{[i-1,i]} = \frac{\Delta E_i}{\Delta T_i} \stackrel{?}{=} U_i \cdot I_i = P_i$$

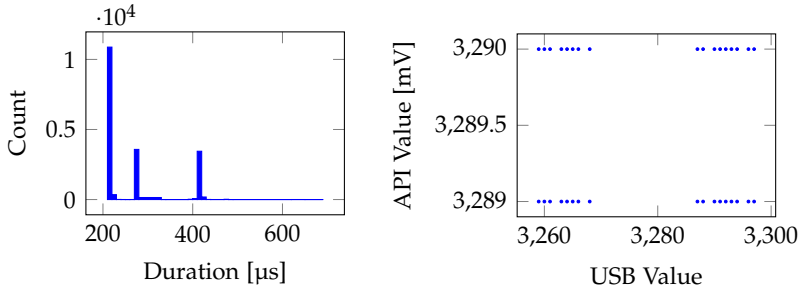
5.2.2 *Baseline Evaluation*

MSP430 LaunchPads are designed as evaluation boards rather than production equipment, hence the EnergyTrace sub-system does not come with any kind of performance or accuracy guarantees. The first evaluation questions relate to the reported energy and current data: how accurate are they, and are they credible in the first place? To answer this, I used eight identical MSP430FR5994 Rev 1.2 LaunchPads bought between 2016 and 2020, running firmware version 31200000 and libmsp430.so version 31200004.

API Data

Preliminary tests indicated that the EnergyTrace firmware and/or client library perform post-processing (low-pass filtering) of some measurement properties before providing them via the API. Additionally, UART output may be buffered by the host MCU's USB-to-UART interface rather than provided in real-time, which affects its usability for benchmark synchronization. So, before performing actual energy benchmarks using a separate, calibrated source/measure unit, let us look into the USB traffic between client library and firmware, and compare it with reported API events.

The benchmark program for this evaluation part puts the target MCU into a low-power sleep state and wakes up periodically to toggle an on-board *Light-Emitting Diode* (LED) and write data to UART.



(a) Histogram of sample durations ΔT_i reported by EnergyTrace API; bin size is 10 µs.

(b) Raw values in USB traffic versus voltages U_i reported by the EnergyTrace API.

Figure 5.3: Voltage and duration as reported by the EnergyTrace API.

Wakeup frequency and UART write size vary between benchmark runs, but are constant within each run. I used the Wireshark utility to monitor USB communication between host and LaunchPad while running the benchmark, and cross-referenced it with API output and expected benchmark behaviour later.

The first observation is that the USB interface transmits EnergyTrace readings in groups of five samples each, at a rate of about 750 Hz. Each sample contains a timestamp, voltage, and cumulative energy reading, with a mean sample rate of 3.75 kHz. The host MCU also buffers UART writes from the target MCU, and flushes them to the USB interface once 64 bytes have accumulated or a timeout of at least 60 ms has expired. USB transmissions that contain EnergyTrace and UART data use a low-priority USB bulk transfer mode that does not provide real-time guarantees.

This already provides an answer to the synchronization question: UART messages alone are only suitable for applications that can handle timing uncertainty on the order of 100 ms. Anything with higher accuracy requirements, e.g. function calls that only take a few milliseconds to execute or short intermediate states, must use a different synchronization mechanism to map energy readings to benchmark events. However, a GPIO-based synchronization method such as the one used in MIMOSA is not available here.

The client-side EnergyTrace library also appears to perform non-trivial post-processing of time, voltage, and energy data provided by the LaunchPad before passing it on to API consumers. While the benchmark shows that each USB event corresponds to exactly one EnergyTrace API event (and vice versa), the relation between a sample in USB traffic and the corresponding API reading is neither left- nor right-unique. Identical voltage, time, and energy values in USB traffic may lead to different API values, and vice versa.

On the timing side, both USB traffic and API output show that EnergyTrace samples are not equidistant. The first sample in each group of five samples typically has a timestamp that is about 400 µs

behind its predecessor, whereas the remaining four typically occur 210 or 270 µs later. Outliers go up to 690 µs; see Fig. 5.3(a) for a histogram of observed ΔT_i values. The USB interface also appears to transmit timestamps with varying resolution. While more than 99 % of duration values (ΔT_i) use a resolution of 650 ns per least significant bit, there are additional clusters around 1, 1.5, and 2.5 µs.

The USB interface likely transmits voltage readings as a millivolt number. I observed values from 3259 to 3297, whereas the API only reported 3289 and 3290 mV (see Fig. 5.3(b)). I assume that this is the result of a low-pass filter in the client library. The mean deviation of API-provided voltage readings from raw USB data (assuming that it is is indeed transmitted in millivolt) is 11 mV.

Cumulative energy values in raw USB traffic have a mean resolution of 365 nJ per least significant bit. In low-power situations, the finest granularity I observed was an API energy difference of 200 or 300 nJ per least significant bit in USB traffic. During periods with very low power consumption, individual samples (both in USB traffic and in API readings) may report $\Delta E_i = 0$, as the DC-DC switching frequency has fallen below the sample rate.

Overall, energy values in USB traffic appear to have the least direct connection to API output. Most notably, they are not monotonic, with about one in 50,000 samples being lower than its predecessor. If they represented actual cumulative energy, this would indicate a *negative* energy flow, which is highly unlikely. The corresponding API values are monotonic and consistent with expectations.

Finally, USB traffic does not contain current readings, so they must be calculated by libmsp430 from time, voltage, and energy data. They appear to be filtered and downsampled to 10 to 750 Hz, depending on average DUT power consumption: the lower its power draw, the lower the effective sample rate. So, $P_i \neq P_{[i-1,i]}$, and $P_i = U_i \cdot I_i$ is not helpful for energy measurements. The remainder of this section derives current and power from energy readings instead:

$$I_{[i-1,i]} = \frac{\Delta E_i}{U_i \cdot \Delta T_i} \qquad P_{[i-1,i]} = \frac{\Delta E_i}{\Delta T_i}$$

Energy Measurement Accuracy

Next, let us compare $I_{[i-1,i]}$ readings to an external source/measure unit that serves as calibration device. As a preliminary evaluation showed that the usable range of EnergyTrace covers 0 to about 25 mA, the following evaluation is limited to 0 to 10 mA.

BENCHMARK This series of measurements does not involve a time-sensitive benchmark application or peripherals. Instead, the benchmark program simply puts the microcontroller into a low-power sleep mode (LPM2 without wake-ups). A programmable current sink, connected in parallel to the target MCU, is responsible for benchmark

execution. Apart from 2.4 µA (8 µW) caused by the target MCU's sleep mode power consumption, all current flowing through the EnergyTrace circuit is also seen by the current sink.

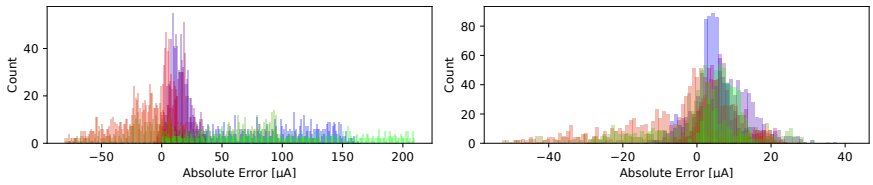
The current sink is provided by a Keysight N6785A *Source/Measure Unit* (SMU) and controlled by a Keysight N6705B DC Power Analyzer. I configured its built-in arbitrary waveform generator for a trapezoid sink current function. After ten seconds at 0 mA, it ramps up to 10 mA (33 mW) over a 90-second interval, waits 20 seconds, and takes another 90 seconds to ramp down to 0 mA. It repeats this cycle four times, resulting in 14 minutes worth of benchmark data per device, and logs voltage and actual sink current to a USB flash drive. Due to the discrete nature of the current sink, ramp-up and ramp-down consist of 99 constant-current steps. Each step increases or decreases current by 101 µA (333 µW) and lasts for about 909 ms.

To avoid interference from a ground loop between the computer connected to the LaunchPad and the power analyzer, the computer ran on battery power without wired Ethernet. I also inserted a low-dropout Schottky diode between the LaunchPad's 3V3 pin and the SMU to avoid damage to the LaunchPad – otherwise, the SMU might back-power it when configured for a sink current of 0 mA. While this decreases the voltage observed at the SMU, it has no effect on the current. I did not take measures to minimize electromagnetic interference or thermoelectric effects, as I expect EnergyTrace to be used without such measures as well.

Finally, for a verification measurement, I replaced the LaunchPad with an N6784A SMU (built into the same power analyzer) operating as a current source. In this case, the power analyzer logged sink and source currents, and I expect the readings to be nearly identical. In the configuration used in these benchmarks, the current ranges of both source/measure units have a specified accuracy of 0.025 % + 10 µA. So, leaving electromagnetic and thermoelectric interference aside, the difference between source and sink current should be no more than 25 µA at 10 mA sink current, and 20 µA in the µA range.

OBSERVATIONS I used the PELT changepoint detection algorithm (as implemented in the Python3 `ruptures` module) to automatically detect changes in sink current, and thus split benchmark results into individual constant-current segments [TOV20; KFE12]. I averaged the current in each segment to allow for a comparison of expected and actual current readings over the entire 10 mA measurement range.

First of all, the verification measurement exhibits an offset between source and sink current of up to 43 µA in the micro-ampere range, and 33 µA (2.3 %) beyond 800 µA. The micro-ampere deviation is twice the expected maximum error of 20 µA, whereas the disagreement in the milli-ampere range is only slightly higher than anticipated. With the environmental factors that can affect micro-ampere readings in



(a) On-board calibration (b) On-board and external calibration

Figure 5.4: Histogram of EnergyTrace measurement error. Each colour indicates a different LaunchPad board; bin size is 1 µA.

mind, I consider these results, and thus the measurement setup, to be acceptable. So, I expect the following EnergyTrace results to be accurate within at least 43 µA as well.

Using EnergyTrace as-is, with just the on-board calibration that it performs automatically, results in a maximum absolute measurement error of 210 µA. Relative error depends on the current range: it is up to 6.9 % below 500 µA and nearly constant (up to 2.9 %) between 500 µA and 10 mA. For some boards, the error is below 1 % in this range. Fig. 5.4(a) shows the individual error distributions.

With additional external calibration at 10 mA sink current, absolute error decreases to 53 µA. The maximum relative error now is 5.2 % below 500 µA, and 1.2 % above. Fig. 5.4(b) shows details for individual LaunchPads.

5.2.3 *Benchmark Synchronization*

At this point, we know that EnergyTrace is capable of measuring the mean current in 200- to 600-microsecond intervals with an error in the lower microampere range. We also know that UART output is not suitable for synchronizing with EnergyTrace events due to buffering and non-realtime USB transfers. I will now explain how my synchronization algorithm manages to perform automated energy measurements under these circumstances by combining in-band signals with an on-board timer and delayed (i.e., not timing-relevant) UART communication.

First off, the benchmark application must ensure that the start and end of benchmark execution are associated with a fixed-duration interval of near-constant power usage. On the LaunchPad, this works by turning on its on-board LEDs for a specific amount of time while keeping the CPU idle and having no active background tasks.

The resulting power usage spikes have a known duration and value. The first one occurs in a well-defined time range at the start of the energy benchmark, whereas the last one is the last spike that is visible in measurement data – once the benchmark is through, it puts the CPU into an idle loop with near-constant power usage. Hence, the

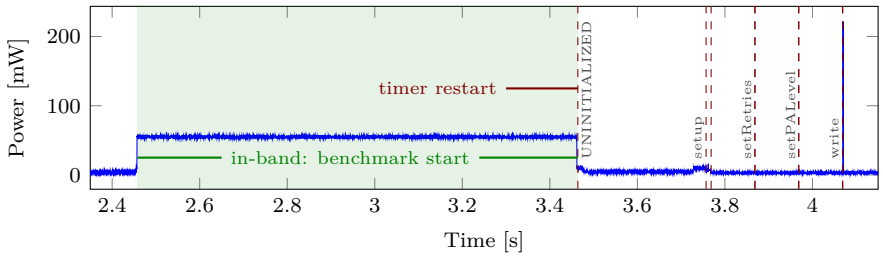


Figure 5.5: An nRF24L01+ energy benchmark with an in-band synchronization signal (tinted area) and state/transition boundaries reconstructed from on-board timer data (dashed bars). Each bar represents a timer restart; the first one coincides with the end of the in-band benchmark start signal.

synchronization algorithm can identify the start and stop timestamps of both in-band synchronization signals.

On top of this, the benchmark application must use a built-in microcontroller timer to count the number of CPU cycles spent in each driver function call (transition) and between each pair of consecutive function calls (hardware state), and log cycle counts and transition IDs to an in-memory data structure. This works by reading and then restarting the timer at the start and end of each transition, thus ensuring that timing data is available for the entire runtime of the benchmark application. Timer control and data logging has a negligible execution time and energy usage overhead, so this does not affect benchmark results.

The benchmark then logs timing data and function calls via UART or another suitable communication interface after each word $w \in L_k(\mathcal{A})$ (i.e., after each series of function calls that starts with an initialization function, see Section 3.4). By placing log output within the UNINITIALIZED state between two consecutive words $w \in L_k(\mathcal{A})$, its time and energy overhead also does not affect benchmark results.

Finally, when analyzing benchmark results, the synchronization algorithm combines the in-band signals for benchmark start and benchmark end with the UART logs that identify the duration (and, thus, relative timestamp) of each PFA state and transition.

The on-board timer starts right at the end of the in-band benchmark start signal, and stops at the beginning of the in-band benchmark end signal. Hence, benchmark and UART timestamps for these two events are known, and the synchronization algorithm can use linear interpolation to calculate benchmark timestamps for all intermediate UART timestamps. Thus, it knows the benchmark timestamps of the start and end of each function call and hardware state. Fig. 5.5 illustrates an in-band synchronization signal and state/transition boundaries reconstructed from on-board timer signals while benchmarking an nRF24L01+ radio chip using EnergyTrace.

I have implemented this in dfatool's `generate-dfa-benchmark.py` script; it can be enabled with the `--energytrace=sync=timer` switch.

Combined with a light-weight operating system, such as Multipass, this allows for automated energy measurements using just a LaunchPad and a DUT (see Fig. 5.1(b)).

5.2.4 *Timing Accuracy*

At this point, the synchronization algorithm assumes that benchmark and UART timestamps come from a perfect clock source that always has precisely the same frequency. In practice, this is not the case: clock frequency is affected by temperature and other environmental factors, and may change (*drift*) over time.

For applications with a single clock source, that is not much of an issue. However, the synchronization algorithm works with two different sources: one provided by the energy measurement device (here: EnergyTrace MCU), and one provided by the microcontroller executing the benchmark application (here: target MCU). Inaccuracies in either clock affect the synchronization between energy measurements and benchmark events. If both drift in a different manner, the calculated benchmark timestamps become less accurate the further they are away from the synchronization points at the start and end of the benchmark. We will now examine how much of an issue this is on the MSP430FR5994 LaunchPad.

The EnergyTrace MCU obtains its clock signal from an ordinary quartz crystal; these typically come with an error of $\pm$ 30 to 100 ppm. The target MCU has two clock systems that are suitable for high-frequency operation: A *Digitally Controlled Oscillator* (DCO), and an optional *High-Frequency External Crystal Oscillator* (HFXT). I am not aware of detailed DCO circuitry documentation or accuracy figures and expect that it focuses on low cost rather than high accuracy.

The HFXT crystal is not populated by default, so the HFXT variant does not strictly fall within commercial off-the-shelf hardware. Still, I was curious to find out how it compares to DCO performance and the drift compensation algorithm outlined in the next section, and added a 16 MHz HFXT crystal as well as two 22 pF capacitors to one of the evaluation boards. This is an ordinary crystal as well, so it also has an expected tolerance of $\pm$ 30 to 100 ppm, depending on manufacturer.

A benchmark event in the middle of a 200 s benchmark has a distance of 100 s to either synchronization point. Assuming conventional quartz crystals with up to 100 ppm tolerance, both EnergyTrace and target MCU time signals have an uncertainty of up to 10 ms in the middle of the benchmark. In the worst case, with -100 ppm on one and $+100$ ppm on the other clock, the synchronization error can be up to 20 ms. For accurate measurements of millisecond signals, this would be a deal-breaker. However, before looking into solutions for that, let us first examine whether it is an issue in practice.

BENCHMARK The evaluation uses a custom driver that interfaces with the LaunchPad's on-board LEDs. Each driver function flashes an LED for a specific duration, ranging from 100 µs to 10 ms. Its PFA consists of a single IDLE state, and one transition from IDLE to IDLE for each LED flash function. The driver does not do anything else.

The benchmark program logs the measured duration and name (i.e., expected duration) of each driver function call; the points at which it dumps its log over UART are also known. Apart from the benchmark's in-band start/end synchronization signals, driver function calls and UART dumps are the only reasons for energy usage beyond the MSP430's idle power consumption in this setup. Hence, if mean power in an EnergyTrace reading exceeds 2 mW, the benchmark is either executing a PFA transition or performing a UART dump. As the order of transitions and UART dumps is known, a computer program can filter out UART dumps and thus determine the EnergyTrace timestamps of each state and transition in the benchmark. The 2 mW threshold is low enough to reliably detect 100 µs LED flashes, even though each sample covers an interval of up to 690 µs.

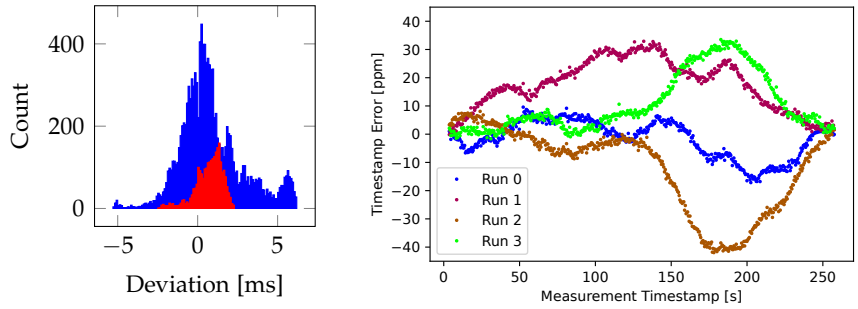
In this setup, each benchmark performs 2500 transitions, and total benchmark duration depends on the idle times between transitions. It ranges from 31 to 256 seconds. I executed each DCO benchmark on three different LaunchPads, with five benchmark runs per LaunchPad. The HFXT benchmark was limited to a single LaunchPad.

This leaves us with two series of timestamps for each benchmark run: one that has been calculated from EnergyTrace and timer data and may be affected by clock drift, and one that has been taken directly from EnergyTrace data. By comparing the timestamps of each event, we can determine clock drift over time.

OBSERVATIONS The maximum distance between a benchmark event and the next synchronization point in these benchmarks is 128 s, so the expected worst-case error for two quartz oscillators with 100 ppm deviation each is 25.6 ms. The observations do not come close to this. For the DCO variant, the maximum deviation between two consecutive readings is 83 ppm, and the maximum total error is 6.2 ms. For HFXT, it decreases to 73 ppm (2.49 ms).

Fig. 5.6 shows a histogram of DCO and HFXT error as well as DCO drift. We see that, while many timestamps are close to reality, there is a sizeable amount of timestamps with an error of more than 1 ms. For measurements of function calls with a duration on the order of one millisecond to tens of milliseconds, this is not ideal.

It is likely that benchmark synchronization faces similar challenges on hardware components other than MSP430FR5994 and EnergyTrace. However, there is an algorithmic way around this.



(a) Error distribution for DCO (blue) and HFXT (red, overlaid) timestamps.

(b) Synchronization error of individual events on a single LaunchPad, using DCO timestamps.

Figure 5.6: Synchronization error of 256-second EnergyTrace benchmarks.

5.2.5 *Drift Compensation*

Changepoint detection is a versatile, general-purpose method of identifying changes such as sudden benchmark power consumption deviations in a data series. It has already been used to identify the different current levels imposed by the source/measure unit in Section 5.2.2.

As changes between hardware states and function calls often come with an observable change in power consumption, changepoint detection may be able to identify those as well, and thus improve synchronization between benchmark events and benchmark timestamps. It is not a trivial solution, though: changepoint detection algorithms require hyper-parameter fine-tuning and may deliver false positives and false negatives. Hence, I devised the following drift compensation algorithm on top of changepoint detection and on-board timer logs.

CONCEPT Even if precise event timestamps are not known a priori, the range in which they occur is. Maximum clock error is a function of oscillator specifications and the time difference from/to the start/end of the benchmark application, whichever is lower. As the observations above show, for a few minutes worth of data, the actual benchmark event is always within a $\pm 10\,\mathrm{ms}$ window around the *uncompensated* timestamp determined from on-board timer logs.

Assuming that the start and end of each driver function call coincides with observable changes in energy usage, each *changepoint* in this window is a candidate for the actual (*compensated*) event timestamp. There may be candidates that do not correspond to a benchmark event, and there may also be benchmark events without observable changes in energy usage (and, thus, with an empty candidate list). However, we can exploit the fact that synchronization error is bounded and evolves over time. Two consecutive benchmark events must have a similar synchronization error, and if we know the deviation for event n, then the deviation of events $n-1$ and $n+1$ must be nearly identical. The task

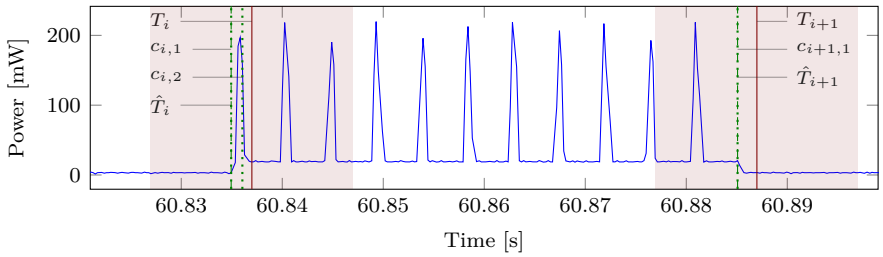


Figure 5.7: A `write` function call within an nRF24L01+ energy benchmark with uncompensated state/transition timestamps reconstructed from on-board timer data (T_i, solid bars), corresponding intervals for changepoint detection (tinted areas), changepoints ($c_{i,j}$, dotted bars), and compensated timestamps ($\hat{T}_i$, dashed bars).

of drift compensation is to solve the optimization problem defined by these considerations: "minimize the change in synchronization error between consecutive measurements."

Fig. 5.7 illustrates this approach for the `write` function provided by the nRF24L01+ radio driver. Here, the radio chip is configured for up to ten automatic retransmissions, hence the graph shows eleven transmission spikes with delays in-between. All of these spikes are part of a single `write` function call.

As we see, the uncompensated benchmark timestamps T_i, T_{i+1} are off by about 2 ms. Changepoint detection identifies two changepoints $c_{i,1}, c_{i,2}$ for the first benchmark event, and a single changepoint $c_{i+1,1}$ for the second one. As the deviation between T_i and $c_{i,1}$ is nearly identical to the deviation between T_{i+1} and $c_{i+1,1}$ (i.e., $T_i - c_{i,1} \approx T_{i+1} - c_{i+1,1}$), whereas $c_{i,2}$ is much closer to T_i, the compensated event timestamps are $\hat{T}_i := c_{i,1}$ and $\hat{T}_{i+1} := c_{i+1,1}$.

INTERPOLATION At first glance, devising a formal algorithm for this approach seems like a simple task. Let $\{T_1, \ldots, T_n\}$ be the uncompensated benchmark timestamps of events one through n, where T_1 is the start of the benchmark (i.e., the end of the in-band start signal) and T_n is the end (i.e., the start of the in-band end signal). Perform changepoint detection within a 20 ms window around each timestamp T_i, using the PELT algorithm [KFE12]. Let C_i be the set of changepoints (candidates) it has determined for each event $i \in \{2, \ldots, n-1\}$. By construction, timestamps T_1 and T_n have an error of zero, so $C_1 = \{T_1\}$ and $C_n = \{T_n\}$. Finally, select compensated event timestamps $\hat{T}_i \in C_i$ so that the following cost function is minimal.

$$\sum_{i=1}^{n-1} \left| (\hat{T}_i - T_i) - (\hat{T}_{i+1} - T_{i+1}) \right| \tag{5.1}$$

However, this function does not consider benchmark events without observable changes ($C_i = \emptyset$), and also does not account for events where changepoints have been detected, but do not contain the correct

event timestamp ($\hat{T}_i \notin C_i$). Handling these requires skipping individual changepoint sets and interpolating between neighbours instead. In this case, the synchronization error of the compensated event timestamp $\tilde{T}_i$ is the average of the synchronization error of the previous and the following event.

$$\tilde{T}_i = T_i + \frac{1}{2}\left(\hat{T}_{i-1} - T_{i-1} + \hat{T}_{i+1} - T_{i+1}\right) \tag{5.2}$$

By expressing changepoint sets C_i as nodes in a weighted directed graph, the drift compensation algorithm can interpolate between missing or incorrect changepoints while still minimizing the change in synchronization error between consecutive measurements. The key idea is to define one node for each candidate within C_i, and have an edge $u \to v$ if and only if u belongs to event i and v belongs to event $i+1$ or $i+2$. For $i \to i+1$ edges, the weight expresses the difference in synchronization error, and for $i \to i+2$ edges, the weight is the difference in synchronization error plus an interpolation penalty. This ensures that the algorithm only performs interpolation when necessary. With this setup, obtaining compensated event timestamps is as simple as using Dijkstra's algorithm to find a shortest path from benchmark start to benchmark end, and looking at the nodes it passes.

ALGORITHM Let T_i, $\hat{T}_i$, and C_i be defined as before. Build a weighted directed graph (V, E, w) consisting of nodes V, edges $E \subseteq V^2$, and weights $w : E \to \mathbb{R}_{\geq 0}$ as follows.

$$V = \bigcup_{i=1}^{n} C_i$$

$$E = \{(u, v) \mid \exists i : (u \in C_{i-1} \cup C_i \wedge v \in C_{i+1})\}$$

$$w(u,v) = \begin{cases} |(u - T_i) - (v - T_{i+1})| & \text{if } u \in C_i \wedge v \in C_{i+1} \\ |(u - T_{i-1}) - (v - T_{i+1})| + 270\,\mu s & u \in C_{i-1} \wedge v \in C_{i+1} \end{cases}$$

Determine the shortest path $\{T_1, v_1, v_2, \ldots, v_m, T_n\}$ from T_1 to T_n. Each path element (u, v) connects two candidates for compensated event timestamps: one from the set C_i (or C_{i-1}), and one from the set C_{i+1}. So, set $\hat{T}_i := u$ (or $\hat{T}_{i-1} := u$) and $\hat{T}_{i+1} := v$. If u is from the set C_{i-1}, there is no suitable changepoint in C_i: set $\hat{T}_i := \tilde{T}_i$ instead (see equation 5.2).

The 270 µs penalty for skipping a changepoint set corresponds to the average interval between EnergyTrace measurements. If needed, this method can be extended for skipping k set of changepoints. If $u \in C_{i-k} \wedge v \in C_{i+1}$, the edge weight is:

$$w(u, v) = |(u - T_{i-k}) - (v - T_{i+1})| + k \cdot 270\,\mu s \tag{5.3}$$

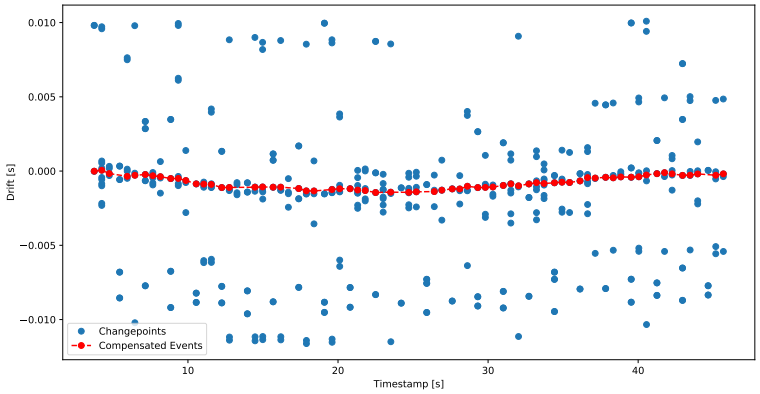
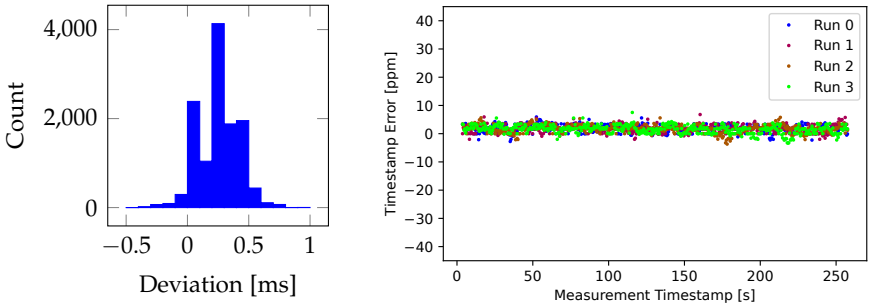


Figure 5.8: Changepoints and corresponding absolute error for each event candidate (blue), and event timestamps inferred by the drift compensation algorithm's shortest path search (red).



(a) Error distribution for DCO timestamps.

(b) Synchronization error of individual events on a single LaunchPad.

Figure 5.9: Synchronization error of 256-second EnergyTrace benchmarks with drift compensation.

Fig. 5.8 shows an example of the nodes in the event candidate graph and the compensated event timestamps selected by a shortest path search, using benchmark data from Section 5.2.4. Each column represents a changepoint set C_i. We see that the shortest path contains several interpolation edges (i.e., edges $\hat{T}_{i-1} \rightarrow \hat{T}_{i+1}$) where the algorithm did not find suitable changepoints $\hat{T}_i$.

EVALUATION Applying the drift compensation algorithm to the observations used in Section 5.2.4 yields a maximum synchronization error of just 0.95 ms (see Fig. 5.9), independent of clock source and benchmark duration. This is close to EnergyTrace's horizontal resolution of 690 µs, indicating that drift compensation will likely perform better on measurement equipment with a higher resolution.

In case of MSP430FR5994 and EnergyTrace, the drift compensation algorithm correctly annotates at least 90 % of any function call that has a duration of at least 10 ms, and more than 99 % of hardware states

that typically last at least 100 ms. The built-in DCO clock source is sufficient and users do not need to populate the HFXT crystal. An implementation of the algorithm is available as part of my open-source msp430-etv[4] and dfatool[5] projects.

While these are promising results, drift compensation is not without limitations. The algorithm relies on the majority of state and transition boundaries having observable effects when it comes to energy usage, and also assumes that those are the only cause of such effects. Systems that run other loads in parallel and function calls that do not affect energy usage can degrade its performance.

Still, I can give a positive answer to **RQ1**: automated energy measurement without access to out-of-band synchronization is feasible. We have also seen that $20 worth of hardware can go a long way – personally, I consider EnergyTrace an ideal solution for student labs (where the focus lies on handing out many devices and letting students gather hands-on experience rather than absolute accuracy of results) and quick, qualitative measurements. I also found this method to be helpful during the coronavirus pandemic that took place during my PhD studies. Carrying tens of kilograms of equipment back and forth between home and office is decidedly less convenient than just taking a LaunchPad.

Alternatives

Some LaunchPads, including the MSP430FR5994 variant, support an enhanced EnergyTrace technology known as *EnergyTrace++*. Here, EnergyTrace MCU and host MCU work together to annotate each EnergyTrace sample with the current status of the target MCU's built-in components. At first glance, this makes it an ideal utility for benchmark synchronization. The target MCU is only awake during function calls and asleep otherwise, so the status annotations in EnergyTrace++ allow for determining state and transition boundaries. As they are part of the EnergyTrace events logged by the EnergyTrace MCU, there are no clock synchronization issues.

However, in contrast to EnergyTrace, using EnergyTrace++ affects the target MCU and thus timing and energy properties of the benchmark application. The host MCU needs to use the target MCU's debug interface to read its status, which costs energy and pauses CPU execution while the interface is active [FG14]. EnergyTrace++ also operates at a lower sample rate of about 1.1 kHz compared to EnergyTrace's 3.75 kHz. I was unable to obtain satisfactory results with it.

Another approach is to use an external logic analyzer instead of the built-in cycle timer for synchronization. While this works, I found no improvement over internal DCO measurements. Neither uncompensated error (105 ppm / 6.07 ms) nor error after drift compensation

4 https://ess.cs.uos.de/git/software/msp430-etv
5 https://ess.cs.uos.de/git/software/dfatool

(0.94 ms) were notably better. Moreover, using an external logic analyzer increases cost and complexity of the measurement setup, and relies on the DUT exposing suitable GPIO pins.

5.2.6 *Related Work*

The drift compensation component of my synchronization algorithm has been inspired by an energy measurement method proposed by Cherifi et al. [Che+17]. Their method also uses changepoint detection to identify hardware state changes, and adds a clustering step to transform observed state changes into a state machine model. While this has the advantage of not requiring a user-provided PFA model, it is unsuitable for driver functions that do not cause an observable change in energy consumption. It is also incapable of expressing the influence of configuration parameters, and generates one hardware state for each configuration with distinct energy attributes instead.

Apart from that, publications that cover automatic energy model generation without out-of-band signals typically focus on smartphones and laptops. Those have built-in battery management units that provide energy measurement capabilities as a side effect. For instance, *DevScope* manages to get by with 104 µA sensing resolution and an update rate of just 0.28 Hz [Jun+12]. It performs workload- rather than state machine-based energy modeling, and is able to work around the battery management system's low update rate by automatically measuring a wide range of workloads for a sufficient amount of time.

Additionally, there is a body of research on building and testing custom measurement devices. While most of those rely on the availability of out-of-band synchronization signals, they still offer a useful combination of low cost and high accuracy, and may fill niches that commercial devices do not address.

iCount employs a trick to work with nearly unmodified off-the-shelf components. By soldering a single connection to an already-present DC-DC converter's feedback pin, the microcontroller running the benchmark application can measure the converter's switching frequency and duty cycle, and thus calculate the amount of transferred energy [Dut+08]. This is similar to EnergyTrace, but uses passive observation rather than active control of switching behaviour. At a measurement error of up to 20 %, it is far less accurate.

EnergyBucket achieves less than 2 % measurement error over 1 µA to 50 mA, at an estimated cost of $60 plus assembly and calibration time [AH09]. It uses a pair of capacitors as charge pumps by discharging a capacitor to a fixed threshold and logging the time it takes from start of discharge to reaching the threshold. As soon as the threshold is reached, EnergyBucket switches over to the other (full) capacitor and re-charges the previously discharged one. While this comes with low hardware cost, it also results in an energy consumption-dependent

measurement interval. Over its specified current range, the time resolution of energy readings ranges from 1.1 kHz to just 0.005 Hz.

The *MIMOSA* board (Section 3.6.1) also falls in this category [BGS13]. It is more expensive and sophisticated than EnergyBucket, using a set of three capacitors and a current mirror that imprints the current flowing to the device under test onto a capacitor. Each capacitor is charged, read, and then discharged in turn. With an error of less than 10 µA and a constant sample rate of 100 kHz, it is one of the most accurate do-it-yourself devices I know.

RocketLogger has a documented error of no more than 0.09 % at up to 500 mA and a maximum sample rate of 64 kHz [Sig+17]. In contrast to MIMOSA, EnergyBucket, and EnergyTrace, it performs power measurements at discrete points in time rather than measuring energy over intervals, so it is unable to identify spikes in energy usage that are shorter than its sample rate. With about $50 worth of hardware components, it is very affordable.

FlockLab focuses on testing several networked devices at once, which is helpful for energy measurements of wireless sensor networks [Lim+13]. Similar to RocketLogger, it performs discrete power measurements at a sample rate of 28 or 56 kHz. It is accurate within 10 % at 100 µA and has an error of less than 0.5 % beyond 500 µA.

Finally, *Spot* combines a shunt resistor with a voltage-to-frequency converter to catch energy usage transients independent of sample rate [Jia+07]. It consists of readily available components and achieves an error of less than 1 µA.

Overall, we see that few research works address the challenge of measurement automation without out-of-band signals. The only examples that I am aware of are smartphone- and laptop-centric solutions that rely on battery management systems, and the method proposed by Cherifi et al. None of these is suitable for the energy modeling methods used in this thesis. Apart from that, research focuses on custom measurement devices and either does not consider benchmark synchronization or uses out-of-band signals.

5.3 CHAPTER SUMMARY

We are now able to automatically obtain sets $S = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \dots\}$ of configurations and corresponding performance attributes. My open-source dfatool project can do so for both fields covered in this chapter: Kconfig-based software product lines, and states and transitions of configurable hardware components.

When it comes to hardware components, we have seen that synchronization of benchmark events and observed power readings still poses a challenge, especially when no out-of-band signalling methods are available. This issue is most prominent in DUTs that do not provide access to GPIO pins or when using affordable off-the-shelf

energy measurement technology rather than expensive professional equipment or DIY solutions. While I performed most energy measurements in this thesis with the MIMOSA platform that is capable of GPIO-based synchronization, I have answered **RQ1** by showing that a synchronization algorithm that exclusively relies on in-band signals can be a viable alternative. With external calibration and drift compensation, it achieves a maximum measurement error of 53 µA and 0.95 ms on the $20 apiece commercial off-the-shelf MSP430FR5994 LaunchPad's EnergyTrace circuit.

At this point, we are equipped to reproduce the variability results presented in Sections 2.7 and 3.6, and start thinking about performance prediction models. However, we have neither considered which configurable features are relevant for these models, nor how variability model and performance prediction model should interact. The next chapter addresses these questions.

6 VARIABILITY MODELS

So far, we have examined two kinds of variability models: feature models that capture the hierarchy and dependencies of software product line configuration options (Section 2.1), and parameterized finite automata that describe states (operating modes), transitions (function calls) and run-time configuration of embedded peripherals (Section 3.3). We have also seen examples of how engineers can extend them with performance models: some modeling languages support feature- and variant-wise performance annotations and aggregation functions (Section 2.2), and parameterized finite automata can be extended to parameterized priced timed automata (Section 3.3).

However, those are just examples. In the SPLE community, there is no consensus on a recommended variability modeling language and performance modeling approach, and the state of the art is far from homogeneous [Sun+21b]. Only a sub-set of variability modeling languages supports integrated performance models by means of feature- and variant-wise annotations, while others have to be combined with separate performance prediction models if performance modeling is desired. This reflects that product line engineering research focuses on handling variability, and performance prediction models evolved as optional add-ons.

With embedded peripherals, it is the other way round: research focuses on energy models, with variability handling being more of a necessary burden. While state machine-based energy models know about variability in the sense of different hardware operating modes, many of them do not consider run-time configuration variables. Those that do, such as parameterized finite automata, treat them as a loose collection of features rather than using a hierarchical feature model.

In addition to that, SPLE and CPS/IoT researchers do not agree on the kind of features that a variability model should consider. SPLE literature tends to focus on boolean feature toggles, and often assumes that performance prediction models need not work with numeric features [Per+21]. Energy modeling methods for hardware components, on the other hand, typically focus on state machines and numeric configuration variables. They often neglect boolean feature toggles altogether or express them as part of the state machine by having one set of states where the feature is enabled and another set where it is disabled.

It is not clear whether this means that numeric or boolean features are irrelevant in the respective domain, or whether they were simply not deemed as important and therefore not considered in the past.

Also, it is not clear how hybrid product lines that contain variable software and hardware components, such as the resKIL embedded AI product line, fit into these disparate assumptions.

The goal of this chapter is to find the common denominator of the two communities and to determine the constraints, requirements, and assumptions that regression model trees work with. It has far-reaching consequences for the remainder of this thesis, as one of my goals is to introduce regression model trees as a common performance modeling method for software product lines and embedded peripherals. The results of this chapter decide how regression model trees and existing variability models should interact, and how embedded peripherals with their loose collection of features play into this interaction.

We start with the relation between variability modeling languages and performance models in software product lines. Here, engineers can choose between integrated performance models that are part of a variability model, and separate performance models that interface with product configurations by means of feature vectors. Both approaches are present in the literature, and there is no community consensus for a recommended variability modeling language or a recommended performance modeling method [Sun+21b]. Section 6.1 presents a qualitative and quantitative analysis of integrated and separate performance prediction models, and thus addresses **RQ2**: should performance models be integrated into variability models, or should they be separate entities?

For embedded peripherals, the question comes from a different angle. Even though these expose variability in the form of run-time configuration options, they are not developed according to product line engineering principles and do not come with a formal variability model. If performance models should be integrated into variability models, engineers must define a variability model before they can extend it with performance attributes. If not, they may consider variability in hardware components as a loose collection of features (i.e., a feature vector) without an underlying hierarchy. Following up on the answer to RQ2, Section 6.2 describes how the run-time variability of hardware components relates to product line engineering principles.

Finally, we will analyze the importance of numeric features in software product lines and boolean features in hardware components. We have already seen in Sections 2.7 and 3.6 that both are present in real-world product lines and real-world embedded peripherals. Sections 6.3 and 6.4 examine whether these types of features have an effect on the modeled performance attributes, and – if so – whether the modeling methods presented in the previous chapters are equipped to handle them.

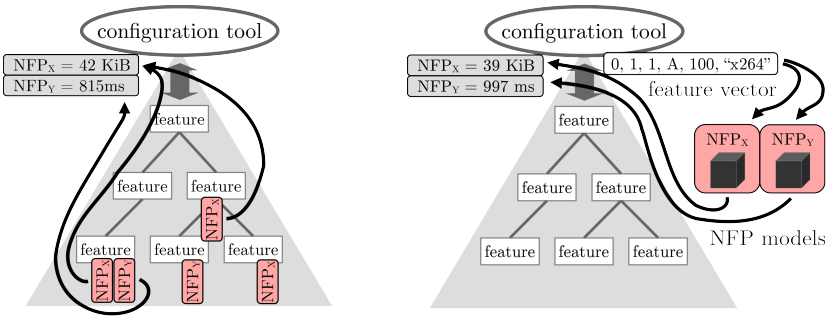


Figure 6.1: Alternative approaches for adding performance models to a feature model: integrated model (left) vs. separate model (right).

6.1 PERFORMANCE MODEL INTEGRATION

> *Related publication*: Birte Friesel et al. "On the Relation of Variability Modeling Languages and Non-Functional Properties". In: *Proceedings of the 26th International Systems and Software Product Line Conference - Volume B*. SPLC '22. Graz, Austria: Association for Computing Machinery, Sept. 2022, pp. 140–144. ISBN: 978-1-4503-9206-8. DOI: 10.1145/3503229.3547055 [Fri+22b]

Integrated performance models rely on built-in features of variability models. As we have seen in Section 2.2, these offer feature- and variant-wise annotations as well as aggregate functions. Aggregate functions may simply state how the attributes of sub-features affect parent features, or include cross-tree references that capture the interaction of feature sets. As textual variability modeling languages follow a well-defined syntax, they can be created and evaluated manually or automatically, e.g. via machine learning and performance-aware configuration tools.

Separate performance models, on the other hand, are not part of the variability model. Instead, they translate product line configurations (feature vectors) $\vec{x}$ into predicted non-functional properties y. Here, the feature vector serves as interface between variability model and performance model, and the performance model $f(\vec{x})$ itself is a black box of arbitrary complexity. Separate models are well-suited for automatic generation and evaluation; some can be generated and evaluated manually as well.

Fig. 6.1 illustrates these conceptual differences. We have already seen examples for both. The feature- and variant-wise annotations in Section 2.2 can be embedded into a textual variability model such as Clafer or SPL Conqueror [Bąk+16; Sie+12b]; the decision tree variants presented in Section 2.5.2 are external models.

At the time of writing this thesis, I am not aware of a clear trend towards either variant in the product line engineering community. The first languages with explicit support for built-in performance models

were published in 2010; since then, some languages have followed up on this idea and improved it, while others do not consider performance models as part of the language at all. A survey undertaken at the Software Product Line Conference (SPLC) 2021 also showed that there is no clear tendency towards either variant [Sun+21b]. Looking at performance modeling research does not answer this either, as many research works in this area exclusively focus on the performance model. Often, it is not clear which variability modeling language (if any) they are using to begin with [Per+21].

So, when designing a novel machine learning method for performance model generation, it is important to know how the generated model will interact with variability models: should it integrate with an existing variability modeling language, or should it be separate? To help answer this, I set out to do a qualitative and quantitative comparison of these two approaches. The qualitative analysis was supported by my colleague Matheus Ferraz.

6.1.1 *Qualitative Analysis*

We will examine six aspects of performance prediction models, starting with the *annotation process* (i.e., data acquisition). Next, we will look at *complexity* and *expressiveness* of the generated model, as well as *flexibility* and *maintenance* when it comes to updating and extending it. Finally, we will also consider whether the two approaches follow the principle of *separation of concerns*.

Annotation Process

First and foremost, the choice between integrated and separate models affects how users annotate model components with performance attributes. In principle, in both model variants, users can do so manually or use a machine learning algorithm to automate the process. We have seen examples for all four cases: manual annotation of an integrated model in Clafer (Listing 2.2), automatic generation of feature- and variant-wise annotations using least-squares regression (Section 2.5.1), manual specification of a regression forest (Fig. 2.5), and automatic generation of regression trees (Fig. 2.3).

In practice, the feasibility of manual annotation depends on model size and the availability of expert knowledge. If a domain expert already knows how features affect system performance, integrated models allow them to create variability model and performance model at the same time, thus saving time and ensuring consistency. For instance, the financial cost of features is typically known in advance rather than a result of benchmarks, and therefore a good candidate for manual annotation. However, especially when faced with a large amount of features and feature interactions, such expert knowledge is no longer reliable [Ach+22].

Hence, most research works rely on automatic model generation using benchmark data and machine learning [Per+21]. This is less tedious and error-prone, but only a sub-set of machine learning algorithms can be used to obtain integrated performance models. Others, such as regression trees, work by building their own model structure and must be kept separate.

So, while integrated models can provide benefits if the feature model is small and expert knowledge for manual annotation is available, separate models with benchmark-based model generation offer a more diverse choice of modeling methods, increasing the likelihood of finding a suitable modeling method for a specific domain. For the annotation process, this means that separate models should be preferred, but there are exceptions to the rule. At least when surveying research that focuses on performance models, separate models are also more widely used in practice [Per+21].

Model Complexity

Once annotations are present, model complexity indicates how well a human can exploit them for performance-aware system configuration. For this aspect, I assume that the user is not relying on a configuration frontend that predicts and annotates the effect of feature toggles, but instead obtaining knowledge about the performance influence of features by looking at the model.

At first glance, integrated models are ideal for this use case. They annotate each feature with performance attributes, use aggregate functions to describe how sub-features affect the performance of a parent feature, and can employ variant-wise annotations to indicate how a feature interacts with features in other parts of the variability model hierarchy. Thus, when a user is interested in the performance influence of an individual feature, they can find all relevant information as annotations related to the feature in question, without having to keep track of the complete feature model.

Separate models, on the other hand, can become quite complex and hard to grasp. A keen reader may have already observed this in Section 2.5.2, where I used hand-crafted LMT and XGB models as examples. The models generated by the corresponding machine learning algorithms were accurate, but far less interpretable.

However, considering the ever-increasing amount of feature interaction in today's product lines, there is a flip side to this [Tër+22]. Expressing feature interaction via variant-wise attributes can quickly clutter the model, making it hard to understand: users have to consider the configuration of dozens of interacting features in order to determine whether an individual feature should be enabled or not. Separate models, on the other hand, benefit from a growing research interest in interpretable machine learning [BP21]. My own regression model tree approach also aims to fit under the interpretable machine

learning umbrella. Still, on average, integrated performance models are less complex.

Model Expressiveness

A closely related subject is expressiveness. More complex models tend to be more expressive, and thus have a better-equipped toolbox for learning and predicting complex product behaviour. They must not become too complex, though, lest they suffer from degraded accuracy due to overfitting [DS98; Ray+19].

The expressiveness of integrated performance models is dictated by the textual variability modeling language they are part of, and typically limited to feature-wise annotations, feature interaction (variant-wise annotations), and aggregate functions (see Section 2.2). Separate models, on the other hand, can be as simple as feature-wise annotations, or as complex as a neural network or even a high-level hardware simulator. Hence, as long as the learning process takes care to minimize the risk of overfitting, separate models are clearly superior in this regard.

Flexibility

The same goes for flexibility. If it turns out that the current integrated performance modeling method is inadequate for the task at hand, engineers have no choice but to switch to a variability modeling language with more powerful performance modeling features. This comes at the cost of having to translate the entire variability model into a new modeling language, which can be a significant source of headache due to subtle differences between different languages [Fei+21]. With separate models, if the current approach is no longer adequate, engineers can switch to a new performance modeling method without having to change the variability model. In most cases, they do not even have to perform new benchmarks, but can use already-available benchmark data to train a new model.

Maintenance

Real-world product lines tend to evolve over time, requiring maintenance of variability and performance models. Whenever a product line's variability model changes, the previously used performance model may no longer be accurate. For instance, it might not be aware of newly added features, or might refer to feature combinations that are no longer valid. Hence, both integrated and separate performance models must be updated.

When using integrated models with manual annotation, domain engineers can update variability model and performance models in one go, thus ensuring consistency. With separate models, manual updates are possible as well, but engineers must take extra care when mapping changes in the variability model to the performance model's

structure and annotations. When relying on machine learning for model generation, or when faced with a separate model that cannot be updated manually, practitioners have no choice but to rebuild the entire performance model. This is also the case when a product line's implementation evolves, e.g. by employing more efficient algorithms or performing additional (mandatory) safety checks: the variability model remains the same, but performance attributes of individual features may be different.

In general, in these cases, engineers must run a new benchmark campaign based on the new variability model and corresponding implementation. Especially for large configuration spaces, this can be a time-consuming task. However, some types of separate models can reduce the amount of benchmark configurations by employing transfer learning to efficiently adapt to implementation and variability model changes [Jam+18]. I am not aware of integrated performance prediction models that support transfer learning, hence I consider separate models to be slightly better suited to this task.

Separation of Concerns

As mentioned above, even if the variability model remains the same, the performance model may need to be updated. The variability model focuses on product attributes rather than subtle implementation details, and hence does not know or care about code efficiency improvements or compiler updates. However, these can impact performance attributes, and in the worst case make the entire performance model worthless, requiring new annotations or benchmarks for everything.

In fact, especially when following SPLE methods, an implementation (and, thus, its effect on performance) is not even available yet when the feature model is designed. With this in mind, integrated performance models clearly violate the principle of separation of concerns: a feature model does not know or care about implementation details, and hence it should not contain performance annotations that depend on them. Separate models, on the other hand, follow this principle.

At this point, most qualitative attributes are in favour of separate models, though their severity depends on the application in question. Table 6.1 gives a summary of the six aspects we examined.

6.1.2 *Quantitative Analysis*

For the quantitative analysis, we will compare model accuracy and complexity of integrated Feature-Wise Annotation (FW) models to separate Classification and Regression Trees (CART). Feature-wise annotations are one of the most simple, easy to understand and easy to annotate integrated performance modeling methods. They can be generated manually or automatically, e.g. by means of least-squares regression [Ros+11; Sie+11]. Classification and regression trees are also

	INTEGRATED	SEPARATE
Annotation Process		(✓)
Model Complexity	✓	
Expressiveness		✓
Flexibility		✓
Maintenance		(✓)
Separation of Concerns		✓

Table 6.1: Qualitative comparison of integrated and separate performance models. A check mark ✓ indicates the better approach in the respective category. Parentheses (✓) indicate that an approach is only slightly better.

relatively simple, and often used as an external modeling method in the literature [Guo+18]. See Sections 2.2 and 2.5 for details.

This analysis relies on the product lines and benchmark results introduced in Section 2.7. To ensure a fair comparison, I used identical feature vectors for FW and CART model generation by mapping disabled boolean features and undefined numeric features to 0 and enabled boolean features to 1. This way, least-squares regression and regression tree generation algorithms can digest all system configurations as feature vectors $\vec{x} \in \mathbb{N}^n$.

Feature-wise annotation associates each feature with a weight that is relevant if and only if the feature is enabled (for boolean features) or that acts as a scaling factor (for numeric features). When employing machine learning to generate feature-wise annotation models, it is sufficient to fit the formula $\beta_0 + \sum_{i=1}^{n} \beta_i x_i$ on training data using least-squares regression. Each regression weight β_i (with $i > 0$) corresponds to a constant feature-wise annotation for feature x_i. The special weight β_0 expresses the base cost or performance of the product line without optional features. For CART, I used the learning algorithm presented in Section 2.5.2. I left its hyper-parameters at their default values ($T_m = T_\sigma = 0, T_d = \infty$), hence it favours accuracy over compactness.

Fig. 6.2 provides accuracy and complexity data for both modeling methods. The top half shows the prediction error of integrated (FW) and separate (CART) models after 10-fold cross validation, and the prediction error of a LUT model (without cross validation) that serves as lower error bound. Due to a wide range in relative error (up to 800 % MAPE for resKIL latency), it reports accuracy using SMAPE, which is easier to graph thanks to its limited range (0 to 200 %). The bottom half shows model complexity (see Section 2.6.2) in logarithmic scale. In both cases, lower values are better.

The results confirm the qualitative analysis in the previous section. Thanks to its expressiveness, the separate CART model is more accu-

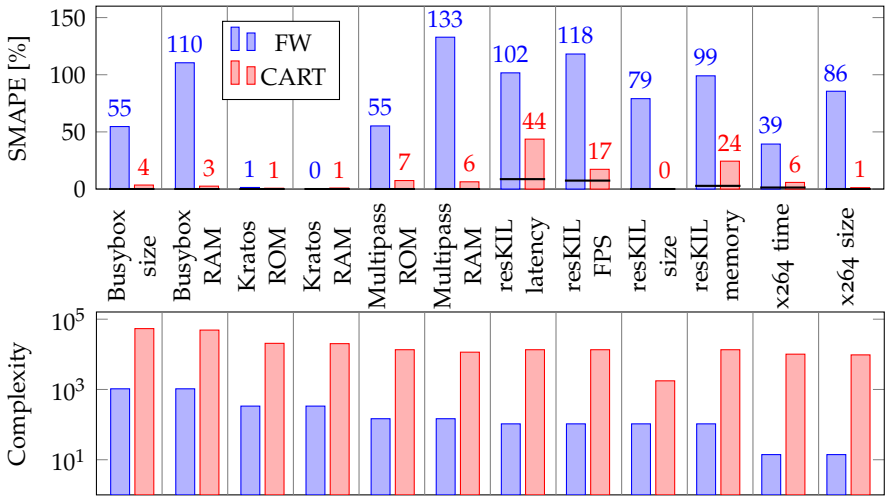


Figure 6.2: Symmetric Mean Absolute Percentage Error (SMAPE) and complexity score (log scale) of integrated Feature-Wise Annotation (FW) and separate Classification and Regression Tree (CART) models after cross validation. Solid lines indicate lower error bound (LUT model, no cross validation).

rate than the integrated feature-wise annotation model in all cases; it is often close to the lower error bound. The feature-wise model often has a prediction error of more than 50 %, and Kratos ROM and RAM usage are the only cases where it is less than 2 %. This indicates that Kratos has less feature interaction than the other product lines, thus allowing a simple feature-wise annotation model to accurately predict its performance attributes.

However, as anticipated, the CART model is also one to two orders of magnitude more complex. At the same time, as the cross-validated results show, it often achieves a prediction error of less than 10 %. It is therefore likely that this complexity is not caused by overfitting, but due to complex feature interactions in the product lines. So, while the integrated model is easier to understand for humans, this comes at the cost of reduced accuracy. The separate CART model is clearly better as soon as automation or tooling for performance model evaluation are involved.

Additionally, even if regression trees are more complex, their structure gives insights into product behaviour. An important side-effect of the greedy decision tree learning algorithm is that influential (and, thus, important) features are located close to the root, whereas less important features end up close to leaves or are left out entirely, depending on training hyper-parameters. For instance, Fig. 6.3 shows the top of the decision tree for predicting busybox RAM usage. Unsurprisingly, it indicates that cross-cutting concerns related to security (`SANITIZE`), debugging (`DEBUG`), and dynamic linking (`STATIC`) have a notable effect on memory usage.

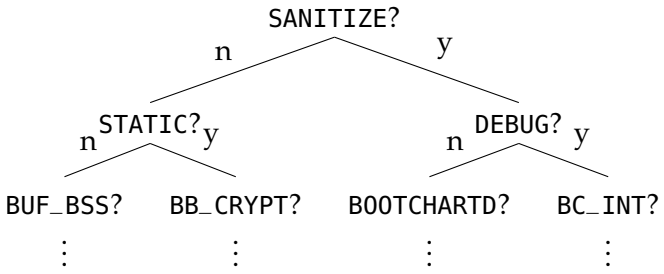


Figure 6.3: Excerpt of a Classification and Regression Tree (CART) model for busybox RAM usage.

As a follow-up conclusion, we see that even if no formal variability model is available, engineers can use performance models to build a feature hierarchy and gain insights about the importance of individual features. This is helpful when working with performance models for components that are not developed according to product line engineering principles and thus do not have a formal variability model, such as hardware components. Next, we will examine whether we can indeed leave out the variability model for those, and solely rely on performance prediction models for performance analysis. I will also explain how this thesis relates hardware components to product lines.

6.2 RUN-TIME FEATURES IN PERIPHERALS

Even though embedded peripherals and device drivers are typically not developed according to product line engineering principles and do not use feature models for variability modeling, I consider them to be similar to product lines.

First of all, many modern peripheral components have a wide range of run-time configuration options such as radio bit rate and transmit power settings, sensor resolution and measurement mode, or display update method [Che+17]. So they, too, expose variability.

Additionally, in my experience, a single application rarely uses all valid run-time configurations of a specific device. Instead, system designers use domain knowledge and energy measurements to determine suitable hardware choices and configurations, and then build their product to use only those. For example, they may configure a wireless radio chip to always perform low-power transmissions with a high bit rate, as they have decided that packet loss is not critical in their application. Or, when selecting hardware components, they may choose an E-Paper display (zero-power standby, costly updates) for a rarely updated electronic price tag, and an LC display (low-power standby, low-power updates) for a frequently updated info terminal. So, each peripheral, combined with the configuration options (features) exposed by its driver, behaves like a product line that system designers can turn into specific products to suit their needs.

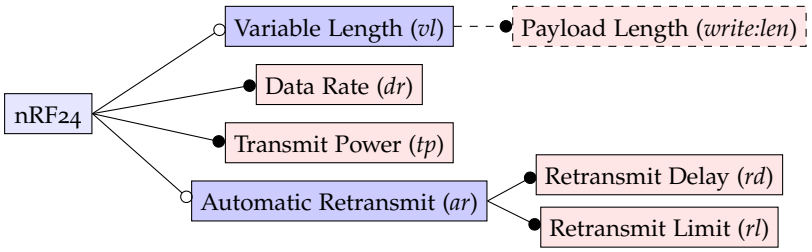


Figure 6.4: Feature model for an nRF24L01+ radio transceiver. Payload Length (dashed) is an argument of the *write* function that does not affect other states/functions. Other function arguments left out for brevity.

For instance, Fig. 6.4 shows a feature model for the nRF24L01+ radio transceiver that I created manually from driver source code and hardware documentation. The parenthesized identifiers in each feature refer to the run-time parameters in the corresponding PFA model in Fig. 3.9.

However, from a performance attribute perspective, there is a notable difference between embedded peripherals and software product lines. In contrast to properties such as hardware cost or ROM usage, energy and timing attributes are runtime-dependent and thus must be viewed in the context of a workload. For example, whether a radio chip is transmitting one or ten packages per second has significant impact on its energy usage and must not be ignored.

This has two consequences when applying performance modeling methods from the SPLE domain. First, data acquisition relies on a benchmark application that dictates the frequency of radio transmissions, sensor readings, display updates, or similar. Second, a performance model generated this way is only valid for this specific benchmark workload. It can predict how configuration changes affect the benchmark application, but cannot determine the effect of different workloads (e.g. less frequent radio transmission or deep sleep instead of idle). Engineers have to re-do the entire benchmark campaign to assess the effect of workload changes, which quickly becomes tedious and cannot easily answer "what if?" questions.

Energy models address this by breaking the model down into components such as hardware states and driver functions, and predicting energy attributes for each of those separately. Thus, the energy model becomes independent of application and workload. Users can now calculate the energy requirements of any application or workload that they can express in a model-compatible manner, e.g. as a series of function calls or as the amount of time spent in each hardware state (see Section 3.3).

Going back to the product line perspective, this means that each hardware component has a single feature model that is associated with *sets* of performance models: one for each hardware or driver

state (PFA state) and one for each driver function (PFA transition). All performance models refer to the same set of global device driver configuration variables, and driver functions may additionally be affected by function arguments. Hence, all PFA states and transitions share the same feature model, which may include function-specific features that relate to individual function arguments.

The performance influence and relevance of individual features varies between different states and functions, and thus between different performance models. For instance, a radio transceiver's sleep mode is typically not affected by transmission-related settings, but its transmit state is. As vendor documentation and device drivers generally lack a formal model that describes these interactions, it is up to the engineer or machine learning algorithm to determine the relevant features and arguments for each state and function.

In fact, available documentation typically does not provide a formal variability model and focuses on configuration functions or registers instead. So, in most cases, engineers only have feature vectors (i.e., an unstructured bag of features) to work with. If they need a feature model, they have to come up with it by themselves.

In general, they do not need one, though. As we have concluded in Section 6.1, separate performance models are often more accurate than models that are integrated into a variability model, and some are able to provide human-readable information about the hierarchy and relevance of features. Hence, a loose collection of features is sufficient for performance prediction, and we can work with embedded peripherals without manual specification of a feature model.

For the remainder of this thesis, this means that we do not have to limit ourselves to proper product lines with formal variability models. All of the following findings and algorithms apply to any product line or application whose features or configuration options can be expressed as feature vectors $\vec{x} \in \{\mathbb{R} \cup \{\bot\}\}^n$. Notably, this includes software product lines, embedded peripherals, and hybrid applications such as the resKIL agricultural AI product line.

Now, the next question is which kind of features a performance model should consider: are numeric features relevant for performance attributes of software product lines, and are boolean features relevant for energy models of embedded peripherals?

6.3 NUMERIC FEATURES IN SOFTWARE PRODUCT LINES

Related publication: Birte Friesel and Olaf Spinczyk. "Performance is not Boolean: Supporting Scalar Configuration Variables in NFP Models". In: *Tagungsband des FG-BS Frühjahrstreffens 2022*. Hamburg, Germany: Gesellschaft für Informatik e.V., Mar. 2022. DOI: 10.18420/fgbs2022f-03 [FS22b]

When presenting the x264 use case in Section 2.7, and at several later points in this document, I mentioned that most SPL research only considers boolean features when building performance prediction models. This is not limited to x264, but a common theme in the literature [Per+21].

Of course, the reason for this might be that engineers deliberately leave out numeric features because they know that those features have no influence on performance attributes of software product lines, and hence do not need to be considered for configuration space exploration and model generation. However, a desire to avoid the complexity of handling numeric features in sampling and machine learning algorithms is also a plausible explanation. In my opinion, two arguments suggest that the latter is the case.

First, the effect of numeric features on product performance may be non-linear, and thus require more than just two samples from its configuration range for learning. So, while each boolean feature doubles the configuration space, numeric features increase it by a factor of three or more, depending on sampling strategy.

Second, the common regression tree / CART performance modeling approach expresses piecewise constant functions. This makes it incapable of extrapolating beyond the data range present during model learning, and limits its accuracy when interpolating between data points. While more sophisticated algorithms like linear model trees exist, they are rarely used in practice.

This section examines two questions related to these arguments: do numeric features affect performance attributes of software product lines, and are CART able to predict their influence? To answer these, we use two sets of observations that build upon benchmark data from Section 2.7: one that takes numeric features into account, and one that does not.

The set with numeric features uses all benchmark data (observations and feature vectors) as-is. The set without numeric features uses the same observations, but its feature vectors leave out numeric (or, more precisely, non-boolean) features. For example, if the set with numeric features is $S = \{((\mathrm{y}, \mathrm{n}, 10), 1.1), ((\mathrm{y}, \mathrm{n}, 20), 1.2)\}$, the set without numeric features is $S' = \{((\mathrm{y}, \mathrm{n}), 1.1), ((\mathrm{y}, \mathrm{n}), 1.2)\}$.

This way, even though the latter set of benchmark data pretends that there are no numeric features, the underlying observations belong to configurations with variable numeric features. This reflects real-world conditions: even if a performance model does not know or care about a product line's numeric features, users may change those at any time and will still expect reasonable prediction accuracy.

To answer the first question – whether numeric features are relevant for performance prediction to begin with – we can examine the prediction error of LUT models for both sets of benchmark data. These models capture the underlying measurement uncertainty when

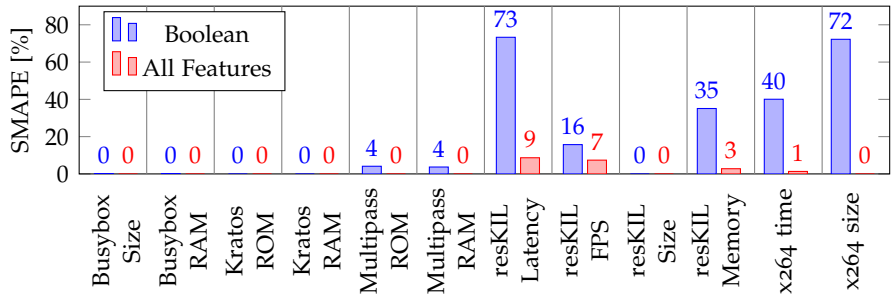


Figure 6.5: Symmetric Mean Absolute Percentage Error (SMAPE) of LUT models with boolean features only (left bars) and with all features (right bars).

all parameters are kept constant. Hence, if numeric features have no influence at all, the prediction error of a LUT model that ignores numeric features must be identical to the prediction error of a model that knows about numeric features. If numeric features are relevant, on the other hand, the error of a LUT model that ignores them will be significantly higher.

Once that is settled, we will examine the prediction error of proper machine learning models (DECART for boolean-only features and CART for all features) after cross validation. This way, we can find out whether these machine learning models can capture the behaviour we observed in the LUT model. If it turns out that respecting numeric features decreases LUT error, this should also be the case for a CART model – otherwise, it may be inappropriate for the task at hand.

6.3.1 *Relevance*

As Busybox, Kratos, and Multipass measurements do not exhibit run-time variability, an ideal model has a prediction error of zero. For Busybox and Kratos, with 25 and 137 numeric features, respectively, the LUT model achieves a near-zero error both with and without numeric features (see Fig. 6.5). So, for these product lines, numeric features are indeed not relevant for performance prediction and may safely be ignored in favour of simple, boolean-only sampling and modeling algorithms.

Multipass, on the other hand, has an error of 4 % when ignoring its eight numeric features. When including them, the error drops to zero. Here, numeric features clearly influence system performance and must not be ignored.

The resKIL embedded AI product line only has a single numeric feature (batch size). It is only relevant at run-time, and hence should not influence neural network size. The results confirm this: resKIL size has a LUT error of zero in both cases. The other resKIL attributes as well as all x264 attributes show significant deviations when ignoring

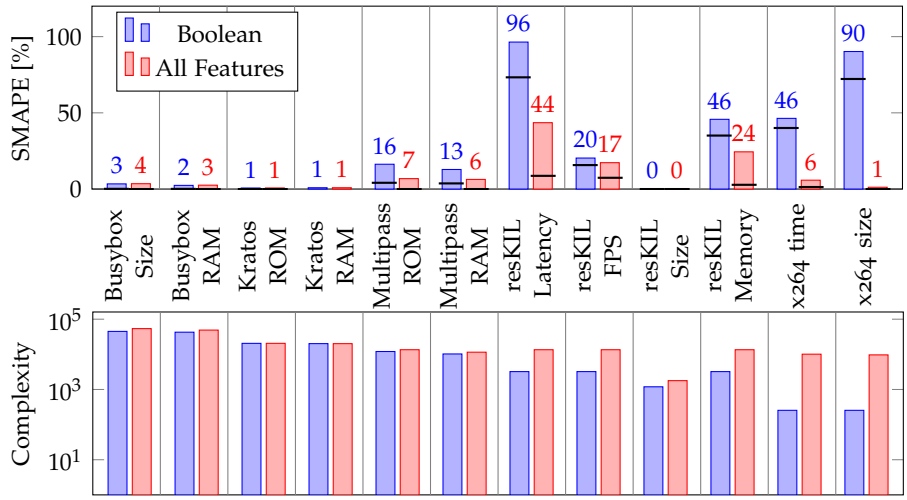


Figure 6.6: Symmetric Mean Absolute Percentage Error (SMAPE) and complexity score (log scale) of models with boolean features only (DECART, left bars) and with all features (CART, right bars) after cross validation. Solid lines indicate lower error bound (LUT model error without cross validation, see Fig. 6.5).

numeric features. In all cases, boolean-only SMAPE ranges from 15 to 80 %, whereas including numeric features decreases it to below 10 %.

These findings allow for two conclusions. First, in general, numeric features affect product behaviour and respecting them during sampling and data acquisition can significantly improve the quality of model learning datasets. Second, there is no relation between the number of numeric features in a product line and their relevance for performance prediction. There are both real-world product lines whose hundreds of numeric features have nearly no effect on performance attributes (e.g. Kratos), and real-world product lines where less than a dozen numeric features are highly influential (e.g. x264, resKIL). Now, let us examine whether CART models are able to transform these improved datasets into better performance prediction models.

6.3.2 *Prediction*

The comparison between DECART (boolean features only, left / blue bars) and CART (boolean and numeric features, right / red bars) in Fig. 6.6 shows that regression trees are capable of reaping this potential. In all cases where numeric features are relevant, CART are more accurate than DECART, often by a factor of two or more. In cases without influential numeric features, CART and DECART error is nearly identical, with no more than 0.2 % difference in prediction error. At the same time, the presence of irrelevant numeric features does not make the models more complex: DECART and CART models

for busybox and Kratos have nearly the same complexity score; only resKIL and x264 show a notably higher CART complexity due to interactions between boolean and numeric features.

In conclusion, we see that numeric features can have significant influence on performance attributes of software product lines. Including them in the sampling strategy and during model generation can significantly reduce the prediction error of performance models without degrading model complexity. In general, the cost of supporting numeric features is low: existing sampling strategies such as systematic configuration space exploration (resKIL, x264) and random sampling (Multipass) can be adapted easily, and the difference in algorithmic complexity between DECART and CART construction is small.

6.4 BOOLEAN FEATURES IN PERIPHERALS

As we have seen in Section 3.6, some embedded peripherals have boolean run-time feature toggles in addition to numeric parameters. Examples include air quality measurements and overall operating mode (BME680), and whether automatic retransmit and variable length payloads are enabled (nRF24, see also Fig. 6.4).

While some energy models in the literature already use boolean variables, these typically refer to hardware states rather than expressing run-time configuration of product line-like features. For instance, given a variable x_{GPS} that indicates whether a smartphone's GPS chip is operating ($x_{\mathrm{GPS}} = 1$) or in sleep mode ($x_{\mathrm{GPs}} = 0$), an energy model can use the regression function $\beta_{\mathrm{GPS}} \cdot x_{\mathrm{GPS}}$ for GPS power consumption [Zha+10]. When using PPTA energy models, x_{GPs} should be part of the state machine structure and not part of the feature vector.

I am not aware of an energy modeling approach that combines state machine models with boolean feature toggles. While PFA and PPTA support them in principle, so far the corresponding learning algorithms have been limited to numeric features [BFS18]. Apart from that, I am only aware of models that work with boolean and numeric features without state machines [Zha+10], and models that work with state machines but do not express run-time configuration as feature vectors [Che+17]. Neither of these fit into my product line approach.

Of course, one might argue that state machine models that only support numeric features are already sufficient. After all, any configuration variable can be expressed as part of a state machine structure by duplicating PFA or PPTA states and transitions, and some approaches in the literature even handle numeric variables this way [Che+17]. Fig. 6.7 shows this elimination of boolean feature vector components for the nRF24 radio transceiver. The model is compatible with existing PPTA benchmark and model generation methods, and can be extended to an energy model without consideration for boolean variables.

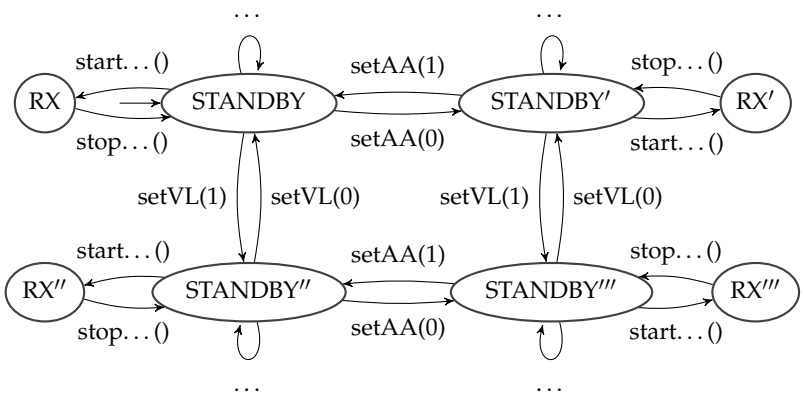


Figure 6.7: PFA model for an nRF24 radio transceiver, with the boolean *ar* and *vl* variables expressed as part of the state machine model rather than as parameters. Parameter assignments and some functions left out or abbreviated for readability.

However, this approach is inadequate. First, Fig. 6.7 already hints towards a complexity issue: each boolean feature that is expressed this way doubles the amount of states and transitions, and with just two boolean features and two hardware states the expanded model is already harder to grasp than its counterpart in Fig. 3.9. Second, this way of handling boolean features violates the relation between peripherals and product lines outlined in Section 6.2: *ar* and *vl* must be part of the feature vector.

With this in mind, let us look into machine learning methods for PFA-based energy models that support boolean configuration variables. Just like in the previous section, the analysis comes in two parts. First, we will examine whether boolean variables affect energy attributes of embedded peripherals, i.e., whether it makes sense to include them in energy models. Second, if it turns out that they do have an influence on energy attributes, we will examine whether machine learning methods are equipped to handle boolean variables in the context of energy models.

Again, the analysis uses two sets of energy measurements. The first one uses the observations from Section 3.6 as-is, and the second one leaves out boolean variables in the feature vector. For example, if the set with boolean features is $S = \{((\mathrm{y}, 5, 7), 1.1), ((\mathrm{n}, 5, 7), 2.2)\}$, the numeric-only set is $S' = \{((5, 7), 1.1), ((5, 7), 2.2)\}$. So, S' pretends that boolean features are irrelevant for the energy model, even though they exist and are changed at runtime. As the CC1200 variability model does not contain any boolean features, the evaluation set is limited to BME680 and nRF24 states and functions that exhibit variability in the observed energy attributes.

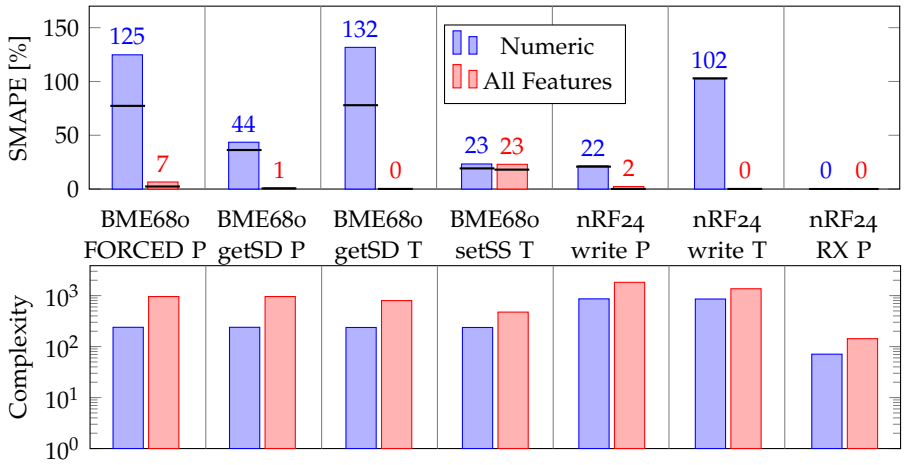


Figure 6.8: Symmetric Mean Absolute Percentage Error (SMAPE) and complexity score (log scale) of CART energy models with numeric features only (left bars) and with all features (right bars) after cross validation. Solid lines indicate lower error bound (LUT model error without cross validation).

6.4.1 *Relevance*

The solid lines in Fig. 6.8 show the LUT error of observations that only consider numeric features (left) and observations that consider all features (right). We see that boolean features indeed have a significant influence on hardware behaviour: in five of seven cases, ignoring them increases LUT error from near-zero (boolean and numeric) to 20 to 100 % (numeric only), indicating significant variability in measurements with identical numeric feature configurations.

In one case (BME680 setSensorSettings duration, abbreviated as setSS T), both observation sets have a high error, indicating configuration-independent variance in hardware behaviour. In another case (nRF24 RX power), numeric features are sufficient. So, in most cases, boolean features have significant influence on hardware behaviour and should be considered for data acquisition and model learning.

6.4.2 *Prediction*

To be in line with the SPL analysis in Section 6.3.2, we would now have to compare least squares regression variants with and without boolean parameters. However, the ULS regression method presented in Section 3.2.1 does not support boolean parameters: a boolean variable has just two distinct values, whereas ULS requires at least three to determine a function type. Linear regression, on the other hand, supports boolean variables but is incapable of expressing non-linear relations between parameters and energy attributes.

While I will present a method for combining ULS and boolean features in Section 7.2, we will resort to regression trees for now. We have already seen that they support boolean and numeric features, and therefore they are sufficient for finding out whether respecting boolean parameters in embedded peripherals can improve the accuracy of machine learning models.

The bars in Fig. 6.8 show prediction error and model complexity of CART models without and with boolean features. They confirm that respecting boolean features significantly improves model accuracy. The prediction error of CART models that respect all features is close to LUT error, whereas the prediction error of numeric-only CART models ranges from 20 to more than 100 %.

However, we also see that respecting boolean features increases model complexity by a factor of two to five in all cases, regardless of whether they influence hardware behaviour (and, thus, model accuracy) or not. This is in contrast to numeric features in software product lines (Fig. 6.6), where only influential numeric features led to a notable complexity increase. So, while respecting boolean parameters is mandatory for achieving reasonable model accuracy, they negatively affect the goal of obtaining understandable performance models. I will present a method for dealing with this in Section 7.4.

In summary, we have seen that boolean parameters are an important aspect of embedded device configuration that should not be ignored during data acquisition and energy model generation. While existing sampling strategies can be adapted easily, only a subset of conventional energy modeling methods supports boolean features.

We have also seen that regression trees may be a useful addition to existing energy modeling methods. They can learn to predict energy attributes that are influenced by boolean and numeric configuration variables with a low cross validation error, but suffer from high complexity, especially when boolean variables are present.

6.5 CHAPTER SUMMARY

We have looked into the relation between variability models and performance models in software product lines and embedded peripherals, and examined whether boolean and numeric features are relevant for performance prediction in both domains.

First off, we have seen that a loose coupling between variability model and performance model is sufficient and often beneficial. This applies to software product lines, where Section 6.1 showed that separate performance models are better than integrated ones in several regards, and to embedded peripherals, where Section 6.2 showed that we can treat variability as an unstructured bag of features (i.e., a feature vector). With this in mind, it is sufficient to use feature vectors as the sole interface for machine learning and performance

modeling methods. Engineers need not specify a variability model if the software project or hardware component they are working with does not provide one.

When it comes to feature selection for performance models, we have seen that boolean and numeric features are relevant in both domains. Overall, this shows that performance prediction for software product lines and energy modeling for embedded peripherals have more similarities than a literature review might suggest. Both work well with a loose coupling between variability model and performance model. In both cases, even though the literature focuses on one type of feature (boolean for SPLs, numeric for peripherals), boolean and numeric features affect performance attributes.

Finally, in both cases, engineers have to consider a trade-off between model accuracy and model complexity. For software product lines, the trade-off lies within the choice between feature- and variant-wise annotations and separate models such as regression trees, examined in Section 6.1. For energy models, it is the choice between user-provided regression functions, which are understandable by construction, and models such as regression trees.

This directly leads to the goal of the next chapter: designing a machine learning algorithm that manages to balance model complexity and prediction error both in the SPLE and CPS/IoT domains. Considering the similarities we have seen so far, and the way different existing approaches reduce complexity or prediction error, such an algorithm may well be within reach.

7 REGRESSION MODEL TREES

We now come to another main contribution of this thesis: Regression Model Trees (RMT) provide a novel data structure and machine learning method specifically tailored towards interpretable and accurate performance models for real-world product lines. With their help, this chapter will answer **RQ3**: can a common machine learning algorithm for SPLE and CPS/IoT performance models provide lower prediction error and model complexity than conventional approaches, without requiring manually provided domain information or model structure?

RMT address this by supporting both boolean and numeric features, and by generating understandable performance models without relying on an already-present variability model. Feature vectors serve as the only interface between product line and performance model.

Before diving into the machine learning algorithm and its evaluation, we will examine its two building blocks. Section 7.1 presents my CART extension that utilizes non-binary nodes to express the performance influence of groups of alternative features in a shallow and thus easier to interpret tree structure. Section 7.2 shows that ULS achieves less complex models than CART and LMT on real-world product lines with all-numeric features, and thus motivates why RMT, unlike CART and their siblings, do not handle numeric features as part of the tree structure. Afterwards, Sections 7.3 and 7.4 present the RMT data structure and learning algorithm. This is followed by an evaluation of model accuracy, complexity and learning time, and an overview over related approaches.

7.1 NON-BINARY REGRESSION TREES

> *Related publication*: Birte Friesel and Olaf Spinczyk. "Black-Box Models for Non-Functional Properties of AI Software Systems". In: *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. CAIN '22. Pittsburgh, PA, USA: Association for Computing Machinery, May 2022, pp. 170–180. ISBN: 978-1-4503-9275-4. DOI: 10.1145/3522664.3528602 [FS22a]

Feature models distinguish between abstract and concrete features (cf. Section 2.1). Abstract features serve as parents for groups of related features, e.g. different scheduler implementations or peripheral drivers, and cannot be enabled or disabled individually. Hence, most feature extraction methods for performance model generation only consider concrete features, typically over the domain $\{0, 1\}$ or $\mathbb{R}$.

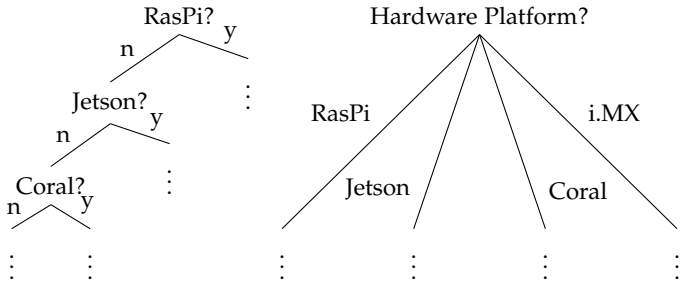


Figure 7.1: Binary (left) and non-binary (right) decision tree for expressing the effect of alternative hardware platforms when predicting resKIL performance attributes.

Boolean sub-features of an abstract feature are either mandatory or optional, and their cardinality (i.e., the number of enabled sub-features) can be specified as well. Common cardinality constraints include "at least one" (a logical-or relation between sub-features) and "exactly one" (an exclusive-or relation). In the latter case, mutually exclusive sub-features are also referred to as *alternative* features. For instance, the architectures supported by the sample operating system product line introduced in Fig. 2.1 are alternatives: each product is compiled for a single hardware architecture.

This idiom is so common that many variability modeling languages provide specific syntax for expressing groups of alternative features. For instance, in Kconfig, features listed within a `choice` block are mutually exclusive: exactly one of them must be enabled if the dependencies of the abstract feature defined by the choice block are satisfied. As shown in Table 2.3, all software product lines in this thesis use choice blocks, and the amount of alternative features can make up a sizeable part of their overall variability.

7.1.1 *Data Structure*

Utilizing this knowledge when learning a decision tree can prove beneficial. Consider a set of k mutually exclusive boolean features $\{\text{feat}_1, \ldots, \text{feat}_k\}$ below an abstract parent feature feat_0. For simplicity, let us assume that feat_0 through feat_k are the only features in the product line, that all of them are relevant for performance prediction, and that exactly one of $\text{feat}_1, \ldots, \text{feat}_k$ is enabled in each configuration.

A binary decision tree, such as one learned by the CART or DECART algorithms, is by design unaware of feat_0 and only works with the k boolean features $x_1, \ldots, x_k \in \{0, 1\}$. The resulting tree has k leaves (one per feature), $k - 1$ non-leaf nodes, and a depth of $k - 2$. One feature does not have a corresponding decision node and is instead implicitly expressed by a path where all other features are disabled. While this does not affect prediction accuracy, it hinders interpretability.

A non-binary decision tree, on the other hand, can use a single *categorical* variable $x_0 \in \{\text{feat}_1, \ldots, \text{feat}_k\}$ to encode the selected feature. It consists of a single decision node that queries x_0 and has one sub-tree (in this case: one leaf) for each of its values. Just like the binary variant, it has k leaves, but only a single non-leaf node. Its depth is 1 rather than $k - 2$. Moreover, the mutually exclusive relationship of feat_1 through feat_k has become part of the decision tree structure, providing important semantic information when visualizing it. Fig. 7.1 illustrates these two approaches, and the simplicity of the non-binary variant, for a part of the resKIL product line.

In principle, this is not a novel realization: the notion of non-binary decision trees that utilize multi-way splits on categorical variables is almost as old as the concept of decision trees itself [Kep96]. However, past research has focused on optimizing accuracy rather than interpretability, e.g. by grouping several features into the same sub-tree [Kep96]. This approach requires careful hyper-parameter tuning and is thus not ideal for automated model generation.

For the *Non-Binary Classification and Regression Trees* (NBCART) used within this thesis, I decided to favour interpretability over accuracy. Hence, they neither group several alternative features into the same sub-tree, nor do they perform pre-processing of groups of alternative features into boolean pseudo-features. Instead, they treat each group of alternative features as a single categorical variable, and extend DECART's handling of boolean features towards multi-way splits with one sub-tree for each alternative feature. The resulting data structure is defined as follows.

Definition 7.1.1 An NBCART is a tree that expresses a piecewise constant function $f : \Sigma^n \to \mathbb{R}$. Each non-leaf node holds a query "x_i?" for some index $i \in \{1, \ldots, n\}$, and each leaf node holds an output value $y \in \mathbb{R}$.

Definition 7.1.2 For an NBCART f, $\textsc{decision}(f) = \langle i, \bot \rangle$ indicates that its root node holds the query "x_i?". $\textsc{children}(f) = \{z_1, \ldots, z_k\}$ returns the set of feature values z for which the node f has a child. $\textsc{child}(f, z) = f_z$ refers to the sub-tree f_z that holds the model for $x_i = z$. If the root node is a leaf node annotated with the value y, $\textsc{decision}(f) = \bot$ and $\textsc{value}(f) = y$.

7.1.2 *Algorithms*

Using this idea in practice requires changes to feature extraction, regression tree generation, and the regression tree query algorithm. Feature extraction now ignores all boolean features $\text{feat}_1, \ldots, \text{feat}_k$ that are mutually exclusive children of a parent feature feat_0, and instead generates a single variable $x_0 \in \{\bot, \text{feat}_1, \ldots, \text{feat}_k\}$. The special value $\bot$ indicates that no alternative feature has been selected, either due to

Algorithm 9 Build an NBCART from observations S.

function BUILDNBCART(S)
  $f \leftarrow$ new NBCART
  if stop criterion satisfied **then**
    DECISION(f) $\leftarrow \bot$
    VALUE(f) $\leftarrow \mu(S)$
    return f
  for $i \in \{1, \ldots, n\}$ **do**
    for $z \in Val_i(S)$ **do**
      $S_{i,z} \leftarrow \{(\vec{x}, y) \in S | x_i = z\}$
    $SSR_i \leftarrow \sum_{z \in Val_i(S)} SSR(\vec{x} \mapsto \mu(S_{i,z}), S_{i,z})$
  $i \leftarrow$ **argmin**(i, SSR_i)
  DECISION(f) $\leftarrow \langle i, \bot \rangle$
  for $z \in Val_i(S)$ **do**
    CHILD(f, z) $\leftarrow$ BUILDNBCART($S_{i,z}$)
  return f

unsatisfied dependencies or because the feature group is optional (i.e., feat_0 is a concrete feature) and has not been enabled by the user.

Given a set of observations $S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\}$ and the usual definitions for $\mu(S)$ and SSR, I adjust the DECART learning algorithm (cf. Section 2.5.2 and Algorithm 3) for NBCART as described in Algorithm 9. The following list outlines the model generation steps in a less formal manner.

1. If a stop criterion is satisfied: return a leaf node using the mean of observed data $\mu(S)$ as model value.

2. For each feature x_i, let $z_1, \ldots, z_k$ be the unique values of x_i in S. Split S into partitions $S_{i,z_1}, \ldots, S_{i,z_k}$ so that S_{i,z_j} only contains entries with $x_i = z_j$. Calculate the model error induced by splitting on x_i: $SSR_i = \sum_{j=1}^{k} SSR(\vec{x} \mapsto \mu(S_{i,z_j}), S_{i,z_j})$.

3. Find the variable x_i with the lowest loss SSR_i and transform it into a decision node "x_i?".

4. Repeat recursively with S_{i,z_j} for $j \in \{1, \ldots, k\}$, adding one subtree to "x_i?" for each set S_{i,z_j}.

Note that NBCART provide a superset of DECART capabilities: they handle boolean features by treating them as categorical features with just two categories (enabled and disabled).

Similarly, they treat each numeric feature as a categorical variable with one category for each observed numeric value. As such, NBCART do not utilize the fact that numeric features have a natural order that can be used for binary decisions on appropriate thresholds (cf. CART:

Algorithm 10 Calculate $f(\vec{x})$ for an NBCART f.

```
function QueryNBCART(f, x⃗)
    if decision(f) = ⊥ then
        return value(f)
    ⟨i, ⊥⟩ ← decision(f)
    if child(f, x_i) then
        return QueryNBCART(child(f, x_i), x⃗)
    C ← children(f)
    return |C|^{-1} · Σ_{z∈C} QueryNBCART(child(f, z), x⃗)
```

"$x_i \leq z$"). While this degrades interpretability and accuracy when dealing with numeric features, it is a deliberate simplification on the way towards RMT. We will rectify this shortcoming later.

When using the non-binary tree for performance prediction, queries may contain configurations of alternative features that were not present in the training set and are therefore not part of the tree structure. For instance, the tree shown in the right part of Fig. 7.1 may be asked to predict throughput on the "UpBoard" platform, but its training set only contained benchmark data for RasPi, Jetson, Coral, and i.MX. When faced with such a situation – a node deciding on a categorical variable x_i that does not have a sub-tree for the requested value – it queries all available sub-trees and returns the mean of their respective predictions instead. In the hardware example, this would be the mean of RasPi, Jetson, Coral and i.MX predictions.

Algorithm 10 gives a formal description of the inference process, with the last line referring to the aforementioned fall-back for queries that contain unknown categorical values.

7.1.3 *Evaluation*

To evaluate whether this approach improves model interpretability and how much it affects model accuracy, we will compare CART and NBCART performance on the product lines presented in Section 2.7 with respect to three metrics: prediction error, complexity score, and tree depth. Both models use the corresponding learning and query algorithms. CART feature extraction leaves out categorical (choice) features, i.e., only considers the left boolean column and the numeric column of Table 2.3. NBCART feature extraction only considers boolean features that are not part of a choice, i.e., the right boolean column as well as the choice and numeric columns of Table 2.3.

Considering the deliberate lack of proper support for numeric features in NBCART, we will examine two kinds of datasets. The Busybox, Kratos and x264 evaluation targets use all available observations and thus include variable numeric features that influence their respective performance attributes. Multipass and resKIL use a sub-set of ob-

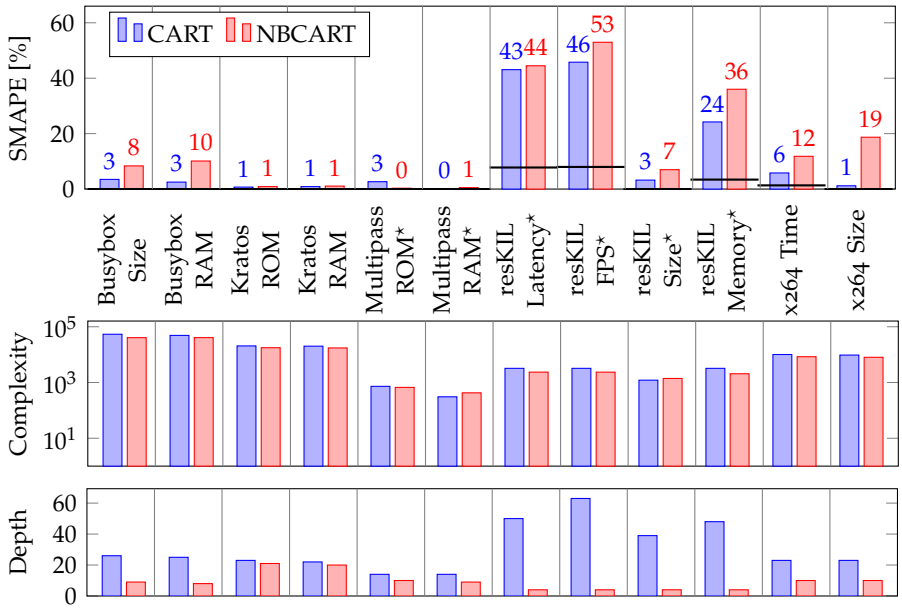


Figure 7.2: Symmetric mean absolute percentage error (SMAPE), complexity score (log scale), and depth of binary and non-binary regression trees after cross validation. Models annotated with a star ($\star$) refer to a sub-set of the configuration space without variable numeric features. Solid lines indicate lower error bound.

servations in which all numeric features are constant, i.e., boolean and categorical features are the only source of variability outside of measurement uncertainty. The Multipass sub-set holds 2,222 of 10,000 samples; resKIL uses $1.04 \cdot 10^6$ of $3.82 \cdot 10^6$ samples for latency and 3,796 of 14,884 for the other three attributes.

Fig. 7.2 shows the results. First off, we see that NBCART achieve their interpretability goal: they are consistently more shallow, with a depth of 4 to 21 compared to up to 63 for CART. The resKIL product line benefits most: as all of its boolean features are part of alternative feature groups, NBCART reduce model depth by a factor of ten.

The complexity score does not show a notable improvement. This is to be expected: While CART and NBCART come up with a different decision tree structure, they have little difference when it comes to leaf nodes. The number of leaf nodes, in turn, makes up more than half of the total complexity score.

When it comes to prediction accuracy, we see that improved interpretability is not free of charge. The NBCART and CART models for Kratos and Multipass have near-identical depth and complexity and also a near-identical prediction error. The NBCART models for Busybox, resKIL and x264, on the other hand, provide a notable reduction of tree depth at the cost of increased prediction error.

While the change in SMAPE for Busybox and resKIL is still acceptable, x264 file size prediction error is more than ten times higher when

using NBCART rather than CART. This underlines that NBCART are an incomplete building block on the way towards RMT. The x264 video encoder has a small amount of highly influential numeric configuration variables, and NBCART alone are incapable of handling them properly.

One way of resolving this is moving numeric features out of the tree structure entirely so that decision nodes exclusively deal with boolean and categorical features, and leaf nodes exclusively deal with numeric features. We will now take the next step towards this goal, and examine how functions learned by ULS compare to tree-based models when dealing with sub-sets of product line configurations whose variability is limited to numeric features.

7.2 NUMERIC FEATURES IN LEAF NODES

As we have seen in Section 3.5, ULS is capable of learning the performance influence of numeric features without user-provided annotations and without decision nodes. It has two advantages over regression tree-based models such as CART or LMT: more expressive functions and extrapolation.

Whereas regression trees express piecewise constant (or, for LMT, piecewise linear) functions by means of a decision tree structure, ULS also has non-linear functions at its disposal. Moreover, CART and DECART can only predict observations that were present during training, and they do not provide information about the relationship (linear, square, ...) between numeric features and system behaviour. ULS formulas, on the other hand, can interpolate between observations, extrapolate to predict the performance of configurations that fall outside the training range, and explicitly show the relationship and relevance of each numeric feature (e.g. βx or βx^2 for some weight β).

However, ULS is incapable of handling boolean or categorical features, and its complexity and extrapolation capabilities also increase the risk of overfitting. So, just like NBCART, ULS is a building block on the way towards RMT rather than a stand-alone solution.

The idea is to generate the model in such a way that ULS invocations only see variable numeric features. If all boolean and categorical features are constant or irrelevant, they can safely be ignored in the feature vectors passed to ULS. A learning algorithm can achieve this, for instance, by first generating a tree structure that captures the performance influence of boolean and categorical features, and then using ULS to annotate each leaf with a function that expresses the performance influence of its variable numeric features. This is similar to the linear functions in LMT leaves, but uses a top-down rather than bottom-up approach.

Assuming that we have such a learning algorithm, the question is whether ULS holds up to the claim of being more expressive, and thus

HYPER-PARAMETER	X264	BME680	CC1200	NRF24
T_d	5	5	5	5
T_l	$\frac{m}{20}$	$\frac{m}{5}$	$\frac{m}{10}$	3

Table 7.1: Hyper-parameter configurations used for LMT learning. $m = |S|$ refers to the number of training samples.

more accurate and less complex than regression tree algorithms. To answer it, we will again compare the prediction error and complexity of various modeling methods: ULS formulas (single continuous function, no tree structure), CART (piecewise constant function, static values in leaves), and LMT (piecewise linear function, using linear regression to obtain leaf functions).

In this case, the evaluation is limited to data sets that do not expose variable boolean or categorical features: all avilable CC1200 measurements as well as sub-sets of BME680 (320 samples), nRF24 (1,788 for write and 1,224 for RX), and x264 (216 samples). Each sub-set has a specific (constant) configuration of non-numeric features and thus satisfies the requirements of ULS learning. Cross validation and complexity score calculation occur as usual, and the training and validation sets are again identical for all models within an evaluation target.

CART hyper-parameters for model training use default values ($T_m = T_\sigma = 0, T_d = \infty$) and thus favour low prediction error over low complexity. ULS does not expose training hyper-parameters. For LMT, I left $T_m = 6$ at its default value, and performed an automatic configuration space exploration (also known as *hyper-parameter tuning*) to select maximum depth T_d and minimum partition size T_l for each evaluation target so that the cross-validation error is Pareto-minimal. Note that dfatool uses identical hyper-parameters for all performance attributes within an evaluation target (e.g. both x264 time and x264 size), hence the need for multi-objective optimization.

In cases where several configurations result in the same cross-validation error, complexity score minimization serves as tie-breaker. Table 7.1 lists the resulting hyper-parameter configurations.

As Fig. 7.3 shows, ULS models are consistently least complex and thus easiest to understand, with a complexity score that is often an order of magnitude lower than for CART or LMT. At the same time, prediction error is often identical, and in one instance (x264 encoding time), ULS provides both the least complex and the most accurate model. There are just three performance attributes where ULS models have a higher prediction error than CART or LMT. So, in most cases, the reduction in complexity that ULS delivers does not come at the cost of increased prediction error.

The three exceptions to ULS accuracy are x264 output file size, BME680 active mode power usage, and nRF24 transmission power

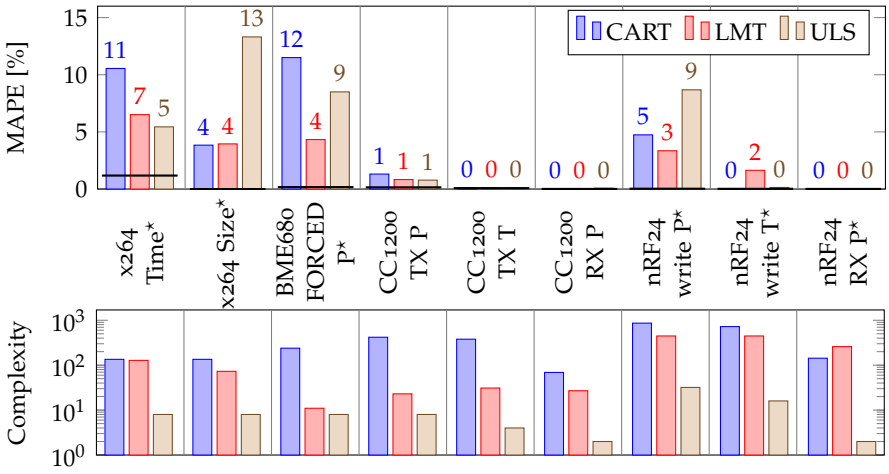


Figure 7.3: Mean absolute percentage error (MAPE) and complexity score of CART, LMT, and ULS performance models for selected product lines. Models annotated with a star ($\star$) refer to a sub-set of the configuration space. Solid lines indicate lower error bound.

usage. In all three cases, complexity – or rather the deliberate simplification employed by ULS formulas – is the root cause. In case of x264 file size, due to encoder implementation details, the number of encoding threads has a non-linear effect on output file size. The prediction error clearly shows that CART and LMT are best equipped for dealing with such performance behaviour. Similarly, the interaction between configuration variables and power consumption in BME680 active mode and during nRF24 radio transmissions is far from trivial to describe. Here, LMT gain an advantage over CART and ULS by combining several linear functions into a piecewise linear model function. Still, in all three cases, the complexity score of ULS is lower, often by an order of magnitude.

For the remaining five attributes, all three models provide accurate performance predictions with no more than 2 % mean error. Just like with x264 encoding time, ULS really shines in these cases: while CART and LMT need tens to hundreds of nodes to express the observed performance behaviour, ULS formulas have just two to eight components and are thus much easier to grasp.

Overall, we see that ULS achieves by far the lowest complexity score. While LMT models offer comparable (and, in two cases, lower) prediction error, they come with higher complexity. CART, on the other hand, are the most complex approach when it comes to models that focus on numeric features, and at the same time often less accurate.

This confirms that unsupervised least-squares regression is a suitable method for handling the numeric part of a product line's performance model. As LMT show, combining decision trees with linear regression is viable as well, though it comes at the cost of increased complexity. Now, the challenge is to combine these findings into a

single RMT data structure and learning algorithm in order to provide a useful balance between accuracy and complexity.

7.3 DATA STRUCTURE

Back in Chapter 6, we have seen that performance attributes of real-world product lines are affected by both boolean and numeric features. While existing machine learning methods such as CART are capable of handling both, the previous sections showed that they strongly favour accuracy over interpretability rather than balancing the two. We have also seen that tree structures are a good fit for expressing the interaction between boolean and categorical features, whereas least-squares regression provides a good balance between accuracy and complexity when dealing with numeric features.

While LMT take a first step towards combining the benefits of the two approaches by allowing linear functions in leaf nodes, they are not well-equipped for expressing non-linear relationships between features and performance attributes in an easily interpretable manner. Moreover, LMT do not distinguish between boolean and numeric features in the model structure, which also hinders understandability: in any given path from the root to a leaf node, a numeric feature may be referenced by more than one node, and it may be referenced both by decision nodes and by the function in the leaf node.

With Regression Model Trees, I decided to take a more radical approach towards feature handling in performance models by splitting the tree structure in two: decision nodes only reference boolean and categorical variables, and leaf nodes only reference numeric variables. This way, they combine the feature interaction handling of regression trees with the expressiveness of least-squares regression formulas.

Considering that at least three distinct numeric values are needed for ULS fitting, an RMT treats numeric features with just two distinct values in the training set as categorical variables and thus eligible for being queried in a decision node. Still, any path through a regression model tree references each boolean, categorical, or numeric feature no more than once. So, when a human analyzes the tree structure and sees that a certain node references a feature, they know that no parent or child node will reference the same feature. This leads us to the following formal definition of *Regression Model Trees* (RMT).

Definition 7.3.1 An RMT is a tree that expresses a piecewise continuous function $f : \Sigma^n \to \mathbb{R}$. Each non-leaf node holds a query "x_i?" for some variable x_i with $i \in \{1, \ldots, n\}$. Each leaf node is annotated with a function $f' : \Sigma^n \to \mathbb{R}$ that only considers numeric parts of Σ.

Boolean variables are defined over the domain $\{0, 1\}$ or $\{\mathrm{n}, \mathrm{y}\}$, with $0/\mathrm{n}$ indicating a disabled feature (either due to user configuration or due to unsatisfied dependencies), and $1/\mathrm{y}$ indicating an enabled

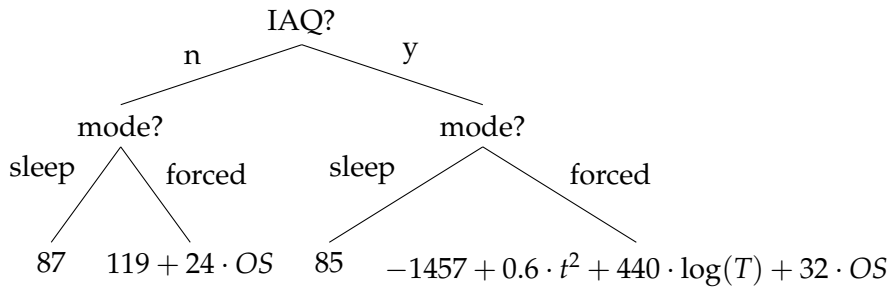


Figure 7.4: A regression model tree for prediction of BME680 active mode power usage. All values in µW, rounded for readability.

feature. The domain of a categorical variable is its set of observed configuration values (i.e., names of mutually exclusive sub-features). It may include the special value $\perp$, which indicates that the categorical feature has not been configured due to unsatisfied dependencies or because it is an optional feature. The domain of a numeric variable may include $\perp$ for the same reason.

Definition 7.3.2 For an RMT f, $\textsc{decision}(f) = \langle i, \perp \rangle$ indicates that its root node holds the query "x_i?". $\textsc{children}(f) = \{z_1, \ldots, z_k\}$ returns the set of feature values z for which the node f has a child. $\textsc{child}(f, z) = f_z$ refers to the sub-tree f_z that holds the model for $x_i = z$. Each sub-tree is an RMT as well. If the root node is a leaf node annotated with the function $f' : \Sigma^n \to \mathbb{R}$, $\textsc{decision}(f) = \perp$ and $\textsc{value}(f) = f'$.

Fig. 7.4 shows an example regression model tree for prediction of BME680 active mode power usage. The boolean *IAQ* (air quality measurement enabled) and categorical *mode* (operating mode) features determine the tree structure. The numeric *OS* (oversampling), t (heater temperature) and T (heater power-on duration) features are only present in leaves of BME680 configurations, and only in those where they are influential. We immediately see that the chip has constant power consumption in sleep mode, a minor dependence on oversampling configuration in active mode without air quality measurements, and an additional influence of heater temperature and duration in active mode with air quality measurements enabled.

In contrast to CART, LMT and XGB, RMT sub-trees refer to discrete values rather than numeric ranges. They can, however, be viewed as a superset of DECART's tree structure (extending boolean queries in decision nodes to categorical queries) and LMT's leaf handling (allowing for non-linear function templates). Note that the presence of non-binary nodes also depends on the feature extraction method: If choices (alternative features) are expressed as groups of boolean features rather than individual categorical variables, the resulting RMT will be an ordinary binary tree. We will now examine the algorithms for learning and querying regression model trees.

7.4 MACHINE LEARNING ALGORITHM

In principle, learning an RMT consists of just two steps. First, build an NBCART (Algorithm 9) while ignoring all numeric features that can be fitted by ULS (i.e., that take at least three distinct numeric values in the training set). Then, for each leaf, use ULS (Algorithm 8) to build and fit a regression formula while ignoring all non-numeric features.

However, there is a catch: least-squares regression analysis must only be used with pair-wise independent features (see Section 2.5.1). Past chapters have acknowledged this limitation and handled it manually where needed. This is not feasible for an RMT learning algorithm that may rely on hundreds or thousands of ULS invocations, each of which works with a different sub-set of training data. So, using ULS as part of RMT generation must involve a pre-processing step that detects and removes co-dependent features before invoking ULS.

We will now examine this pre-processing step in detail, adjust the ULS algorithm accordingly, and then come to the RMT learning algorithm itself. As usual, we work with a set $S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\}$ of observations, and use the notation introduced in past chapters.

7.4.1 *Co-Dependent Feature Detection*

Least-squares regression requires all input variables (i.e., all feature vector components) to be pair-wise independent. This is a well-known fact, and methods such as the correlation coefficient allow statisticians or algorithms to check variables for independence and remove co-dependent variables beforehand. However, they are tailored towards regression analysis and therefore limited to numeric values. RMT also accept features over non-numeric domains and may need to determine whether pairs of variable numeric and non-numeric features are independent[1].

While any non-numeric domain can be mapped to a numeric one in order to determine correlation coefficients or similar metrics, such a mapping imposes meaning that is not actually there. Consider a categorical feature x_1 with observed values b, A, B, and a, and a numeric feature x_2 with observed values 8, 2, 4, and 6. Assume that the benchmark only contains configurations $(x_1, x_2) \in \{(A, 2), (B, 4), (a, 6), (b, 8)\}$, i.e., x_1 and x_2 are co-dependent. A pre-processing algorithm might map $A \mapsto 1, B \mapsto 2, a \mapsto 3, b \mapsto 4$ (lexical order), $A \mapsto 2, B \mapsto 3, a \mapsto 4, b \mapsto 1$ (observed order), or use e.g. a random mapping. For the first case, the correlation coefficient between x_1 and x_2 is 1, implying that the features are co-dependent. In the second case, it is -0.2, suggesting independence – yet, the configurations that

1 Although this is not the case in the algorithm described in this chapter, RMT have done so in the past and still support experiments that rely on being able to check numeric and categorical features for independence.

Algorithm 11 Identify independent features within S.

function INDEPENDENTFEATURES(S, n)
  $D \leftarrow \emptyset$ ▷ set of co-dependent feature pairs
  for $i \in \{1, \ldots, n\}$ **do**
    for $j \in \{i+1, \ldots, n\}$ **do**
      if $|Val_i(S)| > 1 \wedge \left|Val_j(S)\right| > 1$ **then**
        $D \leftarrow D \cup \{(i, j)\}$
        for $v_i \in Val_i(S)$ **do**
          if $\left|\{v_j | \exists \vec{x} \in X : x_i = v_i \wedge x_j = v_j\}\right| > 1$ **then**
            $D \leftarrow D \setminus \{(i, j)\}$
  $I \leftarrow \{1, \ldots, n\}$ ▷ set of independent features
  for $(i, j) \in D$ **do**
    if $|Val_i(S) \cap \mathbb{R}| \geq \left|Val_j(S) \cap \mathbb{R}\right|$ **then**
      $I \leftarrow I \setminus \{j\}$
    else
      $I \leftarrow I \setminus \{i\}$
  return I

led to this outcome are the same, and the features are certainly not independent.

Due to this limitation, I decided not to use the correlation coefficient or similar numeric methods for detecting and removing co-dependent features. I also opted against analyzing the feature model to identify those: feature models are only available in SPLs and SPL-like software projects, but RMT are a general-purpose approach that also supports configurable hardware and similar.

Instead, the RMT learning algorithm employs a heuristic that relies on comparing partitions of observed configurations. While this offers room for future improvement, I have found it to provide usable results in practice, and did not observe drawbacks compared to a heuristic that relies on the correlation coefficient.

The idea is as follows: A feature pair (x_i, x_j) is co-dependent if

- both x_i and x_j take more than one distinct value, and
- For each distinct value v_i of x_i, x_j is identical in all benchmark configurations with $x_i = v_i$.

For each co-dependent pair, the RMT learning algorithm removes the feature that takes the lower amount of distinct numeric values. The idea behind this is that numeric features are generally more helpful for performance prediction than non-numeric ones. Algorithm 11 describes the check and removal steps in detail.

Going back to the example, we see that x_2 is identical in all benchmark configurations with $x_1 = v_1$ for $v_1 \in Val_1(S) = \{A, B, a, b\}$: $x_1 = A \Rightarrow x_2 = 2$, $x_1 = B \Rightarrow x_2 = 4$, and so on. Moreover, x_1 takes

Algorithm 12 Build and fit an n-dimensional prediction function on the set of observations S, using the set of function templates G.

function BUILDULS′(S, G, n)
  $F \leftarrow \emptyset$
  $Feat \leftarrow$ RELEVANTFEATURES$(S, n) \cap$ INDEPENDENTFEATURES(S, n)
  for $i \in Feat$ **do**
    if FINDTEMPLATE$(S, G, i) \neq \bot$ **then**
      $F \leftarrow F \cup \{$FINDTEMPLATE$(S, G, i)\}$
  if $F = \emptyset$ **then**
    return $\vec{x} \mapsto \mu(S)$
  fit $f(\vec{x}) = \sum_{F' \in \mathcal{P}(F)} \left(\beta_{F'} \cdot \prod_{f \in F'} f(\vec{x}) \right)$ on S
  return f

no numeric values ($|Val_1(S) \cap \mathbb{R}| = 0$) whereas x_2 takes four numeric values ($|Val_2(S) \cap \mathbb{R}| = 4$). So, Algorithm 11 removes feature x_1.

In order to add this pre-processing step to the ULS learning algorithm, it is sufficient to replace the RELEVANTFEATURES check with a check for both RELEVANTFEATURES and INDEPENDENTFEATURES. Algorithm 12 shows this adjustment.

7.4.2 *Tree Generation*

Just like all regression tree learners examined thus far, RMT generation uses a greedy algorithm that recursively adds child nodes until a stop criterion is satisfied. For decision nodes, it only considers features that are configured in all benchmark samples (i.e., $\bot \notin Val_i(S)$) and not suitable for ULS. For leaf nodes, it relies on ULS, which only considers numeric features and leaves out all non-numeric ones.

Note that the learning process does not use type annotations for features, and instead infers the type of feature i based on the observed values $Val_i(S)$. This increases the flexibility of the approach. As each decision node partitions S into smaller and smaller sets, individual features may become meaningless (or meaningful) in individual subtrees. For instance, if a numeric feature j depends on a boolean feature i being enabled, all observations will be either $x_i = 0 \wedge x_j = \bot$ or $x_i = 1 \wedge x_j \in \mathbb{N}$. When a tree node splits on feature i, feature j becomes meaningless in the $x_i = 0$ branch, and useful for ULS in the $x_i = 1$ branch.

Algorithm 13 describes the learning process in detail; the following list outlines its steps in a less formal manner.

1. For each feature x_i, examine the unique values that it takes in S.
   a) If $\bot \in Val_i(S)$, the feature is undefined (i.e., not configured) in some configurations: skip it.

Algorithm 13 Build an RMT from observations S using the set of function templates G.

function BUILDRMT(S, G, n)
  $f \leftarrow$ new RMT
  for $i \in \{1, \ldots, n\}$ **do**
    if $\perp \in Val_i(S)$ **then**
      $SSR_i \leftarrow \infty$ ▷ Feature is partially undefined
    else if $Val_i(S) \subset \mathbb{R} \wedge |Val_i(S)| \geq 3$ **then**
      $SSR_i \leftarrow \infty$ ▷ Feature can be handled by ULS
    else
      for $z \in Val_i(S)$ **do**
        $S_{i,z} \leftarrow \{(\vec{x}, y) \in S | x_i = z\}$
      $SSR_i \leftarrow \sum_{z \in Val_i(S)} SSR(\vec{x} \mapsto \mu(S_{i,z}), S_{i,z})$
  $i \leftarrow$ **argmin**(i, SSR_i)
  if $SSR_i \geq SSR(\vec{x} \mapsto \mu(S), S)$ **then**
    DECISION$(f) \leftarrow \perp$
    VALUE$(f) \leftarrow$ BUILDULS'(S, G, n)
    return f
  DECISION$(f) \leftarrow \langle i, \perp \rangle$
  for $z \in Val_i(S)$ **do**
    CHILD$(f, z) \leftarrow$ BUILDRMT$(S_{i,z}, G, n)$
  return f

   b) If $Val_i(S) \subset \mathbb{R}$ and x_i takes at least three distinct values, the feature can be modeled via ULS and it should not be part of the tree structure: skip it.
   c) Otherwise, let $z_1, \ldots, z_k$ be the unique values of x_i in S. Split S into partitions $s_{i,z_1}, \ldots, S_{i,z_k}$ so that S_{i,z_j} only contains entries with $x_i = z_j$. Calculate the model error induced by splitting on x_i: $SSR_i = \sum_{j=1}^{k} SSR(\vec{x} \mapsto \mu(S_{i,z_j}), S_{i,z_j})$.

2. Find the feature x_i with the lowest loss SSR_i.

3. If all features were skipped or splitting on x_i would not improve model error: return a leaf node containing a ULS function learned on S.

4. Otherwise: transform x_i into a decision node "x_i?" and repeat recursively with S_{i,z_j} for $j \in \{1, \ldots, k\}$, adding one sub-tree to "x_i?" for each set S_{i,z_j}.

In contrast to CART and CART-based algorithms, the RMT learning process deliberately does not expose user-configurable stop criteria or hyper-parameters such as $|S| < T_m$, $\sigma(S) < T_\sigma$, or maximum tree depth T_d. The idea is that RMT learning should be entirely unattended, without the need for costly hyper-parameter fine-tuning.

Algorithm 14 Calculate $f(\vec{x})$ for an RMT f.

function QUERYRMT($f, \vec{x}$)
  if DECISION(f) $= \perp$ **then**
    return VALUE(f)($\vec{x}$)
  $\langle i, \perp \rangle \leftarrow$ DECISION(f)
  if CHILD(f, x_i) **then**
    return QUERYRMT(CHILD(f, x_i), $\vec{x}$)
  $C \leftarrow$ CHILDREN(f)
  return $|C|^{-1} \cdot \sum_{z \in C}$ QUERYRMT(CHILD(f, z), $\vec{x}$)

7.4.3 *Queries*

The evolution from NBCART to RMT requires little changes in the query algorithm. Leaf nodes are now functions rather than static values and must be evaluated accordingly. Apart from that, the special case for queries which contain values of categorical features that are not part of the tree structure remains. Just like NBCART, RMT accommodate these by averaging predictions of known feature values. The recursive query function is defined in Algorithm 14; it starts at the root node.

Note that RMT may reference numeric variables both as part of the tree structure and as part of a leaf's regression formula. However, no path from the root to a leaf contains both variants. If a leaf references a numeric variable, there is no decision node referencing it on the path. If a leaf does not reference a numeric variable, there may be a such a decision node.

This allows RMT to express the influence of numeric variables even if insufficient data for ULS modeling is available, without sacrificing interpretability. When users encounter a node referencing a numeric feature during manual model analysis, they know that neither the path from this node to the root nor any path from the node to a leaf will reference the same feature. All information related to the feature is encoded in the single tree node they are currently examining.

7.5 EVALUATION

To evaluate the accuracy, complexity and learning time of Regression Model Trees, we will now compare them with related machine learning methods from the literature (see Section 2.5.2): Classification and Regression Trees, Linear Model Trees, and Extreme Gradient Boosting. The software projects and hardware components introduced in Sections 2.7 and 3.6 serve as evaluation targets.

TARGET	BOOLEAN		CATEGORICAL	NUMERIC	SAMPLES
Busybox	1,009	991	5	19	32,000
Kratos	39	36	1	2	30,000
Multipass	100	70	6	4	10,000
x264	9	7	0	4	16,800
resKIL	106	0	5	1	14,884
resKIL Latency	106	0	5	1	$3.82 \cdot 10^6$
BME680 FORCED	2	2	0	3	1,140
CC1200 TX	0	0	0	3	4,200
CC1200 RX	0	0	0	2	615
nRF4 write	3	3	1	5	3,720
nRF24 RX	1	1	0	3	1,224

Table 7.2: Number of usable features by type and sample counts of evaluation targets. The Boolean column indicates total number of features (left) and number of features that are not part of a choice (right). Features that have the same value in all observations are left out.

7.5.1 *Setup*

The CART, LMT, and XGB algorithms only support numeric input domains (i.e., they learn functions $f : \mathbb{R}^n \to \mathbb{R}$). Kconfig feature extraction for these models leaves out `choice` entries (categorical features) and `string` entries, and maps all `bool` features to 0 (disabled) or 1 (enabled). For numeric features, it maps all configurations in which the feature is not configured due to unfulfilled dependencies ($x_i = \perp$) to 0. Hardware component evaluations map all non-numeric feature vector components to numeric values using the lexical order of observed configurations, and also map undefined features to 0.

Kconfig feature extraction for RMT includes `choice` entries and thus leaves out `bool` features that are part of a choice. The remaining `bool` and numeric features are left as-is; it ignores `string` features as well. Feature vectors of hardware components are also left as-is.

Table 7.2 lists the feature counts and types that the learning algorithms work with. As in Table 2.3, the left boolean column lists all boolean features, whereas the right boolean column only lists boolean features that are not part of a choice. CART, LMT and XGB use the features listed in the left boolean column and in the numeric column; RMT models use the features listed in the right boolean column, the categorical column, and the numeric column.

TARGET	LMT		XGB		TARGET	LMT		XGB	
	T_d	T_l	T_d	K		T_d	T_l	T_d	K
Busybox	5	$\frac{m}{10}$	12	20	resKIL	5	3	14	200
Kratos	5	$\frac{m}{5}$	6	600	BME680	5	$\frac{m}{20}$	12	20
Multipass	8	$\frac{m}{20}$	14	150	CC1200	5	$\frac{m}{10}$	6	50
x264	8	3	8	100	nRF24	5	3	6	100

Table 7.3: Hyper-parameter configurations used for LMT and XGB learning.

A feature i is listed as boolean if it takes precisely two values when defined ($|Val_i(S) \setminus \{\perp\}| = 2$). If it takes at least three values when defined ($|Val_i(S) \setminus \{\perp\}| \geq 3$), it is listed as either numeric (if $Val_i(S) \subset \mathbb{R} \cup \{\perp\}$) or categorical (otherwise). Note that, in contrast to Table 2.3, this overview leaves out features that have the same value in all observed configurations. Those are not useful for performance prediction and ignored by all evaluated machine learning methods.

The discrepancy in the two x264 boolean cells of Table 7.2 stems from the fact that both choices in the x264 variability model have just two options. So, by the definition above, they are either treated as four boolean options (four Kconfig `bool` entries) or as two boolean options (two Kconfig `choice` entries that have just two distinct values each, and thus behave like boolean features).

CART model training leaves all hyper-parameters at their default values ($T_m = T_\sigma = 0, T_d = \infty$). I confirmed that deviating from those values consistently leads to increased model error (albeit with lower complexity) by performing hyper-parameter tuning. So, the only relevant stop criterion for CART is $|\{\vec{x} \mid (\vec{x}, y) \in S\}| = 1$ (i.e., there is nothing left to split on).

LMT uses the same setup as in Section 7.2. The minimum number of samples remains at its default value ($T_m = 6$). Maximupm depth T_d and minimum partition size T_l have been selected for Pareto-minimal cross-validation error (primary objective) and complexity score (tie-breaker) by means of hyper-parameter tuning.

XGB learning uses default values for most hyper-parameters: an influential shrinkage term ($\eta = 0.3$), no sub-sampling ($r = 1$), and low complexity penalties ($\gamma = 0$ and $\lambda = 1$). Here, hyper-parameter tuning adjusts XGB's maximum depth T_d and number of trees K to minimize cross-validation error (primary objective) and complexity score (tie-breaker).

Table 7.3 lists LMT and XGB hyper-parameter configurations. The RMT learning algorithm deliberately does not expose hyper-parameters, and thus does not mandate a time-intensive tuning process.

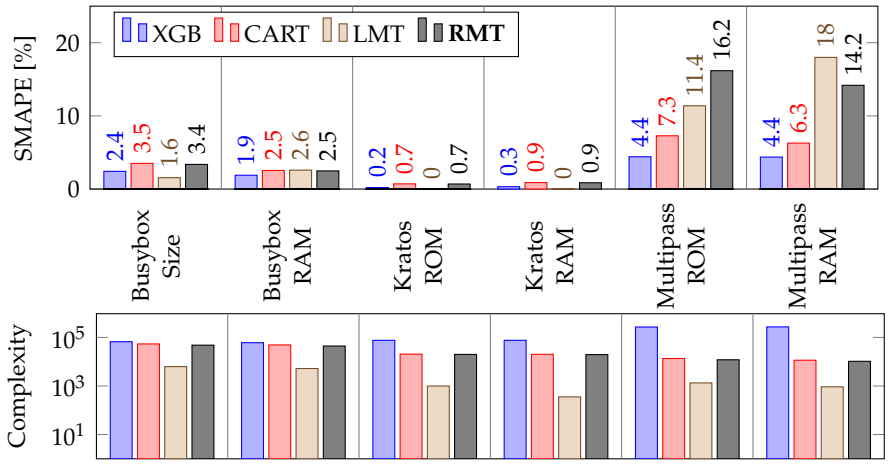


Figure 7.5: Symmetric mean absolute percentage error (SMAPE) and complexity score of Regression Model Trees (RMT) compared to other machine learning methods on conventional software product lines. Lower error bound is 0 % in all cases.

7.5.2 *Accuracy and Interpretability*

We will examine the evaluation targets in three groups: conventional software product lines where boolean feature toggles are the main contributor towards performance variability (Busybox, Kratos, Multipass), product lines with influential numeric features (resKIL, x264), and hardware components (BME680, CC1200, nRF24).

Conventional Software Product Lines

As we have seen in Section 6.3, the performance attributes of Busybox, Kratos and Multipass are largely independent of numeric feature configurations. From a product line engineering perspective, this makes them ideal candidates for modeling methods such as CART.

Fig. 7.5 confirms that CART and their sibling XGB consistently have a low prediction error on these product lines, with less than 4 % for Busybox and Kratos and less than 8 % for Multipass. However, with tens of thousands of tree nodes, their models are also the most complex ones and thus anything but interpretable.

LMT and RMT also achieve less than 4 % error for Busybox and Kratos. LMT complexity is up to an order of magnitude lower, while RMT complexity matches that of CART and XGB. Here, hyper-parameter tuning and the pruning step that is part of the LMT learning algorithm show their benefit. However, both learning algorithms come up with a sub-par prediction accuracy for Multipass.

The difficulties that LMT and RMT face when predicting Multipass ROM and RAM usage are caused by a combination of two factors. First, while the non-functional properties of Busybox and Kratos are

largely independent of their numeric features, Multipass performance is affected by them to a measurable extent (cf. Fig. 6.6). Second, the benchmarks of all three product lines (including Multipass) rely on random sampling.

In principle, when looking at a specific numeric feature, the entire data set contains a variety of configurations for it. However, each unique configuration of non-numeric features may be paired with just one unique value of the numeric feature – which is insufficient for linear or least-squares regression. For Busybox and Kratos, this is not an issue, as the few numeric features are largely irrelevant. Multipass, on the other hand, would benefit from a model that uses linear or ULS functions to describe the effect of its numeric features.

The RMT algorithm is incapable of coming up with such a model due to its strict distinction between non-leaf nodes (boolean and categorical features only) and leaf nodes (numeric features only). In fact, when examining the RMT models for Multipass, all leaves return static values. This confirms that, once all non-numeric features have been dealt with during tree structure generation, there is not a single leaf node that has numeric features with at least three distinct values left, so ULS returns a static function in all cases. It also explains why RMT complexity is close to CART: both algorithms use the same stop criterion and, due to the lack of numeric features that can be modeled via ULS, both build a tree structure that encompasses nearly all variable features.

The LMT models, on the other hand, have been pruned too aggressively: The trees for ROM and RAM contain just a dozen leaf nodes each and attempt to describe most of the product line's performance behaviour by means of linear functions. As the cross-validation results show, neither approach is suitable when using random sampling, and LMT and RMT learning would likely benefit from a more systematic data acquisition method [Nai+20; Per+21]. This is a notable difference to model learning for purely boolean product lines, where random sampling is a good match [Guo+18].

We will examine LMT and RMT performance on data sets that use systematic rather than random sampling in the resKIL and hardware component evaluations in the remainder of this section.

Hybrid Product Lines

With the hybrid resKIL product line and its exclusively categorical and numeric features, things start getting more interesting. We see in Fig. 7.6 that RMT achieve the lowest or close-to-lowest prediction error for all resKIL attributes thanks to their combination of categorical feature support and non-linear function templates. RMT complexity is also on the lower end of all evaluated modeling methods, and sufficiently low to allow for manual interpretation. We will use this to

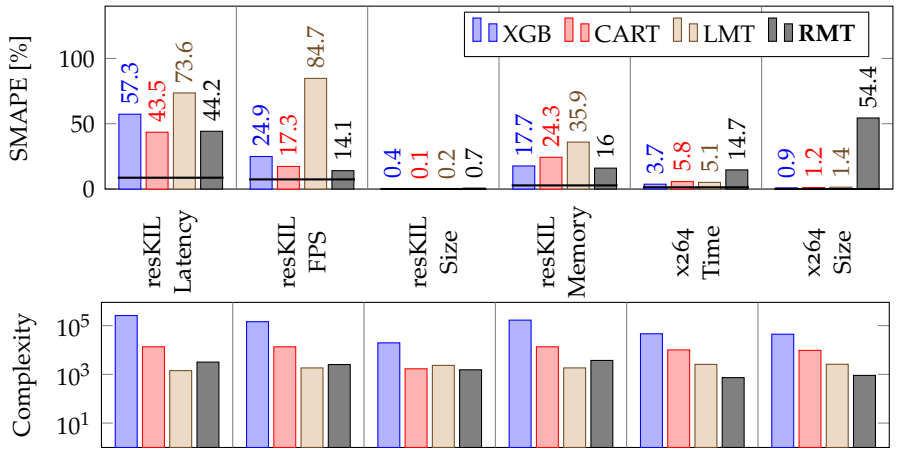


Figure 7.6: Symmetric mean absolute percentage error (SMAPE) and complexity score of Regression Model Trees (RMT) compared to other machine learning methods on hybrid product lines. Solid lines indicate lower error bound.

gain insights into the performance of neural networks on embedded systems in Section 8.1.

While CART and XGB prediction error for resKIL performance attributes is close to RMT, these models are more complex and thus less easy to understand. LMT, on the other hand, come with lower complexity but two to ten times higher prediction error. Hence, RMT are clearly the best option for prediction and analysis of resKIL performance attributes.

When it comes to x264, the CART, LMT and XGB models all have a prediction error of less than 6 % and a high complexity score of 1,000 to 100,000. Prediction error for encoding duration is in fact better than the 9 to 37 % reported in the literature [Sie+15; Zha+15; DAS21] – however, note that all listed references use different sampling strategies and sample sizes, and consider different x264 features for performance modeling. I am not aware of publications that predict output file size.

While RMT models for x264 also have a complexity score of around 1,000, their prediction error is far higher, especially for output file size. In this case, sampling is not the root cause. Instead, the x264 implementation is so complex that its performance attributes cannot be predicted adequately by tree structures that move numeric features into leaf nodes.

In all non-RMT models, the decision nodes closest to the root refer to numeric variables such as desired encoding quality (in variable bit rate mode), bit rate (in fixed bit rate mode), and output width/height. This indicates that x264's numeric features are far more influential than its boolean feature toggles. RMT do not allow decisions on numeric features and handle them in leaf nodes instead.

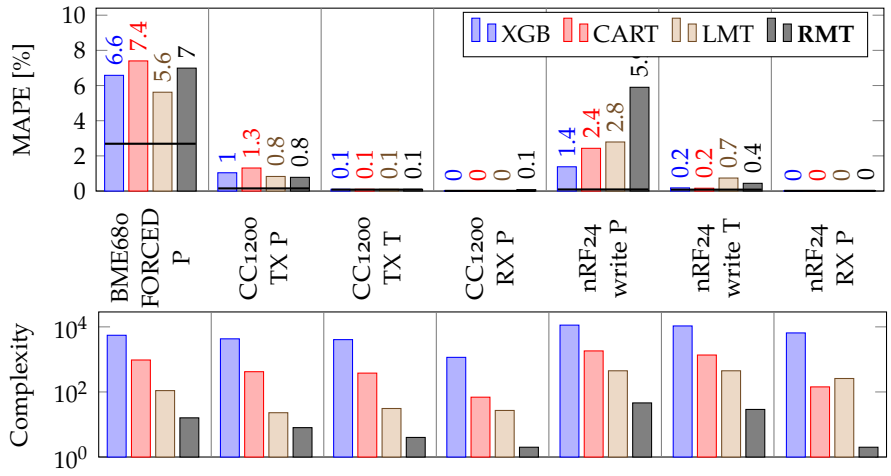


Figure 7.7: Mean absolute percentage error (MAPE) and complexity score of Regression Model Trees (RMT) compared to other machine learning methods on energy models for hardware components. Solid lines indicate lower error bound.

We have already seen in Fig. 7.3 that this does not work well for x264. All three evaluated modeling methods (CART, LMT, ULS) have difficulties when predicting the performance of x264 configurations that only vary numeric feature toggles. This applies to all benchmark data sub-sets of x264 that have constant boolean and variable numeric configurations: the cross-validated SMAPE of CART, LMT and ULS models on any of these data sets is consistently higher than the CART, LMT and XGB SMAPE values shown in Fig. 7.6. Similarly, adjusting the RMT algorithm to use CART or symbolic regression rather than ULS does not lead to noteworthy improvements in prediction error.

So, the high RMT prediction error for x264 is not an issue of ULS, but a consequence of x264's complexity and RMT's strict distinction between boolean/categorical decision nodes and numeric leaf nodes. The most accurate interpretable model that I am aware of, utilizing a regression formula with less than 20 feature- and pair-wise interaction terms to predict how eight boolean and 13 numeric features affect x264 encoding time, achieves a prediction error of 12.5 % [Sie+15]. The x264 video encoder may simply be so complex that the currently well-known modeling methods are incapable of describing its performance in an interpretable and highly accurate manner.

Hardware Components

For the third and final evaluation set, Fig. 7.7 shows how RMT and other machine learning algorithms fare when working with energy attributes of hardware components. Here, the situation is different: all four algorithms achieve a low prediction error of less than 8 %, and in all but one cases the difference between the best and worst model for

a given energy attribute is less than two percentage points. Instead, the complexity scores differ by several orders of magnitude.

XGB uses nearly 10,000 tree nodes in all cases, while CART achieve the same prediction error with 100 to 1,000. LMT complexity score ranges from 10 to 200, and RMT complexity is by far the lowest, utilizing just 1 to 20 tree nodes and regression formula components.

So, just like with the hybrid resKIL product line, RMT are clearly the best option for predicting and interpreting how run-time configuration affects the energy behaviour of hardware components. Their prediction error is on par with established methods from the SPLE community, while their complexity is lower by several orders of magnitude. We will now examine whether they are also able to transform training data into performance models in a timely manner.

7.5.3 *Learning Time*

As their name suggests, a key goal of interpretable models is to provide users with ways of gaining novel insights into how a system component's configuration affects its performance values. As such, the associated machine learning method should be able to quickly learn a performance model from benchmark data so that users can inspect it.

While we cannot evaluate this aspect in a completely fair manner – the RMT implementation is strictly a research project that favours experimentation and extendability over performance, whereas CART, LMT and XGB are optimized for speed – it is still interesting to see in which order of magnitude the algorithms operate.

Since dfatool works with entire system components (product lines or hardware components) rather than individual performance attributes, we will review the learning time for entire components (e.g. CC1200 or resKIL) rather than individual performance attributes.

From a model learning complexity point of view, the evaluation set contains two classes of targets. The first class (Busybox, Kratos, Multipass and resKIL) comes with a high-dimensional configuration space ($n \approx 100$ to $n \approx 1,000$) and thousands to millions of samples ($k \approx 10^4$ to $k \approx 10^6$). The second class (x264, BME680, CC1200 and nRF24) has a low-dimensional configuration space ($n \approx 10$) and a low amount of samples ($k \approx 10^3$). Fig. 7.8 shows the learning times.

In the first class, all evaluated algorithms struggle with the amount of data. CART are fastest, with a few seconds to nine minutes for model training. RMT come in next, with 50 seconds to 22 minutes. XGB manages to stay just below an hour, and LMT model learning can even take slightly more than six hours. For XGB and LMT, hyper-parameter tuning adds several additional weeks of processing time.

Combined with the accuracy and complexity figures shown in Figures 7.5 and 7.6, we see that RMT are the ideal choice for the hybrid resKIL product line: they are fastest, most accurate, and least

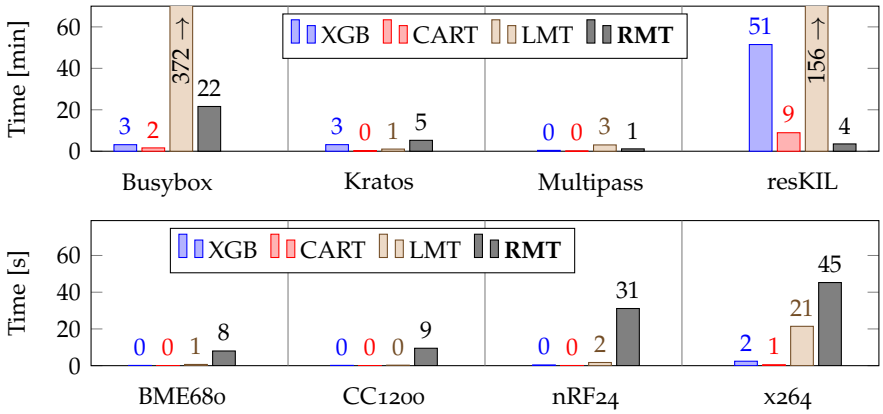


Figure 7.8: Wall-clock time for model learning on an AMD EPYC 7763 CPU.

complex. For Busybox, Kratos and Multipass, CART are a good choice, albeit less easy to interpret when dealing with alternative features (cf. Section 7.1).

In the second class, all algorithms take less than a minute to come up with a model. CART and XGB are near-instantaneous with no more than two seconds, whereas LMT take up to 21 seconds and the unoptimized RMT implementation needs up to 45. However, as we have seen in Fig. 7.7, waiting a little longer for RMT generation pays off by obtaining a much easier to understand and equally accurate model. Here, LMT and XGB come with an additional 10 to 60 minutes of hyper-parameter tuning overhead.

Overall, we see that there is no single modeling method that performs best in all application scenarios. However, we can conclude that RMT achieve a good balance of accuracy (low prediction error), interpretability (low complexity), and speed in many cases. Although CART offer comparable accuracy and faster model generation, they are more complex by up to two orders of magnitude and thus less suitable for interpretation by humans, especially when working with hybrid product lines and energy models.

We have also seen that the RMT algorithm trades accuracy for interpretability in some cases, especially when faced with unsuitable sampling techniques (Multipass) or complex performance behaviour (x264). Improved sampling, an extension with LMT-style pruning, and (in rare cases) a relaxation of the strict distinction between categorical decision nodes and numeric leaf nodes will likely improve accuracy with minimal interpretability drawbacks.

7.6 RELATED WORK

We have already examined CART, DECART, LMT, XGB, least-squares regression and ULS in Sections 2.5 and 3.5. Moreover, Sections 3.2 and

5.2.6 have covered sampling and modeling methods that specifically focus on state machines and energy models.

This section looks into additional work that relates to RMT and product lines in general. We will cover two fields: the machine learning algorithm itself (modeling methods) as well as data acquisition and feature extraction (sampling).

7.6.1 *Modeling Methods*

As mentioned previously, CART and DECART are a staple of SPLE performance modeling research that focuses on the effect of exclusively boolean feature toggles [Per+21]. For instance, given a product line with n feature toggles, DECART can often achieve less than 10 % model error with less than $10 \cdot n$ samples [Guo+18]. However, they do not support numeric or choice features, and are not tailored towards interpretability.

Linear model trees are less common. In one case, they have been used for predicting software faults from software quality attributes with a 5 to 50 % model error [RK16]. Due to their flexibility of referring to numeric features both in binary splits and in regression formulas in leaf nodes, LMT learning is generally slower than working with less expressive models such as CART or XGB [Loh14]. We have observed this in Section 7.5.3.

Zhang et al.'s *provably-optimal sparse regression trees* (SRT) divert from the greedy algorithms of the past and build upon dynamic programming with bounds instead [Zha+23]. While this approach has shown promising results both in terms of accuracy and interpretability, it does not support categorical or numeric features yet. Instead, it relies on pre-processing to turn them into boolean pseudo-features, thus hindering interpretability especially when dealing with hybrid product lines and energy models for hardware components.

Acher et al. propose a different way of reducing model complexity and prediction error: they employ a feature sub-set selection pre-processing step, thus identifying relevant features and leaving out irrelevant ones during model learning. The motivation behind this is similar to the RELEVANTFEATURES algorithm used within ULS, and incorporating it into RMT is an interesting avenue for future work. Combined with a regression forest, Acher et al. achieve a MAPE of less than 7 % when predicting Linux kernel size from its configuration options, while speeding up learning time by a factor of 5 to 48 [Ach+22]. They do not mention model complexity and instead focus on the mismatch between features that influence kernel size according to Linux kernel documentation and features that actually influence kernel size according to the model.

P4 focuses on prediction error rather than interpretability, and addresses the issue that the precise performance influence of individ-

METHOD	INPUT	OUTPUT	SPECIALTY
RMT	Σ^n	continuous	interpretable
CART [Bre+84]	$\mathbb{R}^n$	constant	
DECART [Guo+18]	$\{0,1\}^n$	(constant)	few training samples
LMT [Qui+92]	$\mathbb{R}^n$	linear	
XGB [CG16]	$\mathbb{R}^n$	linear	ensemble method
SRT [Zha+23]	$\{0,1\}^n$	(constant)	interpretable
Sub-sets [Ach+22]	$\{0,1\}^n$	(constant)	faster learning
P4 [DAS21]	$\{0,1\}^n$	(constant)	confidence intervals
NN [Her+22]	$\mathbb{R}^n$	continuous	manual model design

Table 7.4: Input domains, piecewise output functions, and distinguishing attributes of RMT and related modeling methods from the literature. Models over $\{0,1\}^n$ have piecewise constant output by design.

ual features can rarely be modeled with 100 % accuracy [DAS21]. Dorn, Apel, and Siegmund combine feature sub-set selection with feature- and pair-wise annotation in a regression model while explicitly considering model confidence. This allows them to associate each performance prediction with a confidence interval, and also identify individual features that come with a high uncertainty. Like most approaches covered here, P4 is limited to boolean feature toggles.

Finally, *BEARS* is a utility for improving the energy efficiency of configurable software systems by means of automatic, workload-dependent re-configuration [Her+22]. Herzog et al. evaluate a variety of regression tree and least squares regression models and compare them to a custom neural network (NN) architecture. While I am not aware of model-specific MAPE or SMAPE measures, their findings indicate that the neural network model achieves the highest energy efficiency improvements. They also show that tree and least squares regression models allow for energy efficiency improvements. In contrast to neural networks, those do not rely on manual model design.

Table 7.4 compares the key attributes of RMT, the modeling methods presented in Section 2.5, and the related work covered in this section.

7.6.2 *Sampling*

The work by Siegmund et al. combines accurate model generation with efficient sampling and supports both boolean and numeric features [Sie+15]. The authors utilize *Plackett-Burman* experiment design to determine the effect of numeric feature configurations [PB46], and find that it is superior to random sampling approaches. Model generation relies on linear regression with optional feature-interaction

support, and has been designed with interpretability in mind. For instance, when evaluating x264 video encoder configuration by varying eight boolean and 13 numeric features, they are able to predict encoding time with an error of 12.5 % with just 636 samples. The sampling setup is highly influential; different approaches range from 15 % error with 1046 samples to 36.7 % error with 216 samples. With less than 20 terms, the corresponding regression formulas are easily interpretable by humans.

L2S, on the other hand, follows an active sampling approach that considers environment (e.g. hardware and workload) changes in addition to conventional feature re-configuration [Jam+18]. In contrast to the resKIL product line, it does not model hardware components as explicit features. Given a model for one target environment, L2S automatically infers benchmark configurations that should be measured in order to adjust the model for a new environment, without having to repeat all benchmarks for each new environment. When modeling the latency of an AI application and adapting the model to hardware and workload changes, Jamshidi et al. report a model error of 7 to 20 %. While CART serve as underlying modeling method, the documented approach is limited to boolean feature toggles.

FLASH also relies on active sampling, but with a focus on optimization rather than performance model generation [Nai+20]. Given a set of optimization goals, it automatically determines benchmark configurations so that a CART model trained on those can be used to find Pareto-optimal system configurations for these specific goals. It supports boolean and numeric features, but relies on optimization goals that must be known beforehand. Whenever those are changed, it must perform new benchmarks and train a new model.

Finally, *Fourier Learning* is similar to P4 insofar as that it focuses on prediction error, but during sampling rather than in the generated model [Zha+21]. It allows users to specify a maximum prediction error and guarantees that any model trained with an appropriate number of samples (which is a function of the prediction error) will satisfy the specified accuracy bound. However, as the variables used by Fourier Learning models do not directly relate to product line features or configuration variables, they are anything but interpretable.

7.7 CHAPTER SUMMARY

This chapter has presented Regression Model Trees (RMT), a data structure and machine learning method that combines and extends ideas from the SPLE non-functional property modeling and CPS/IoT energy modeling communities. RMT utilize non-binary decision nodes in order to encode information about groups of alternative features in a shallow and easy-to-understand manner, and rely on ULS to automatically find and fit interpretable functions that describe the

influence of numeric features in leaf nodes. Their tree structure separates boolean/categorical features (decision nodes) from numeric features (leaf nodes) and ensures that, in each path from the root to a leaf node, each feature is associated with no more than one tree node.

As the evaluation has shown, combining ideas from these otherwise disjunct communities pays off whenever influential numeric features are present and an appropriate sampling technique for training data acquisition is used.

For the hybrid resKIL product line, RMT achieve both the lowest model error and – despite not being optimized for speed – the lowest learning time, with a model complexity that still allows for manual inspection. On the hardware side, RMT are consistently one to two order of magnitudes less complex than other methods, whereas model error is comparable in all but one cases. Additionally, in contrast to established methods such as linear regression, RMT are able to deal with hardware behaviour that depends on boolean or categorical features without having to encode this decision as part of the state machine structure and thus suffering a state space explosion [Che+17].

Of course, an algorithm is rarely ever complete, and RMT are no exception. The evaluation and literature review have shown that the current RMT design tends to sacrifice accuracy for interpretability. For instance, when it comes to the x264 video encoder, relaxing the distinction between boolean/categorical and numeric features in the tree structure by allowing decision nodes to decide on numeric features when appropriate will likely prove helpful. I also expect that incorporating the ULS RelevantFeatures heuristic and/or adapting the one utilized by Acher et al. to decrease the number of features considered for RMT generation will decrease model complexity when dealing with product lines such as Busybox [Ach+22].

Nevertheless, the RMT algorithm and its evaluation have already given a positive answer to **RQ3**: the challenges faced by the SPLE non-functional property modeling and CPS/IoT energy modeling communities are not as different as one might expect, and combining approaches from the two communities benefits both of them. Users can obtain accurate and interpretable models in a timely manner without having to provide any kind of domain information or model structure.

We will now examine three practical applications of RMT: an analysis of embedded machine learning performance, a performance-aware product line configuration frontend, and an analysis of trade-offs between data processing cost and data transfer cost. The first application includes manual interpretation of RMT models for the resKIL product line. The third one shows how engineers can combine already-available performance models for system components when building performance models for a new produtc line.

8 APPLICATIONS

While low complexity – as a proxy metric for interpretability – and low prediction error are desirable goals when designing machine learning algorithms for performance model generation, a performance model is not a cause unto itself. It is most useful when configuring or reasoning about real-world software systems and hardware components. The context in which a performance model is used may also impose (or relax) accuracy and interpretability constraints that are not necessarily captured adequately by a single average percentage error or model complexity score.

Moreover, most evaluation targets used so far – and most evaluation targets used in the literature – consider either variable software or variable hardware components, but not a combination of both. While Section 2.7.2 has already established resKIL as a hybrid product line with variability in both aspects, it has not addressed whether it makes sense to apply conventional product line engineering techniques in such a case to begin with.

This chapter presents three real-world case studies to cover these aspects:

- the *resKIL* agricultural AI product line (Section 8.1),
- the *kconfig-webconf* drop-in replacement for retrofitting performance models onto existing Kconfig-based product lines (Section 8.2), and
- a trade-off analysis of data serialization formats in wireless IoT networks (Section 8.3).

The resKIL agricultural AI product line combines configurable software with interchangeable hardware components, while the data serialization format analysis combines configurable hardware and software components with workload-dependent performance attributes. Together, these serve as an answer to **RQ4**: are product line engineering and performance modeling techniques also applicable to product lines that cover soft- and hardware variability?

kconfig-webconf, on the other hand, focuses on the machine-readable aspect of performance models. It shows that utilizing them to annotate product line features with non-functional properties is not difficult, and that adding performance models to existing product lines and other kinds of configurable software systems only requires a minimal amount of manual labour. kconfig-webconf and resKIL, which also serves as a case study for kconfig-webconf, provide context for evaluating the interpretability and accuracy of RMT models.

8.1 BLACK-BOX MODELS FOR AI SOFTWARE SYSTEMS

Related publication: Birte Friesel and Olaf Spinczyk. "Black-Box Models for Non-Functional Properties of AI Software Systems". In: *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. CAIN '22. Pittsburgh, PA, USA: Association for Computing Machinery, May 2022, pp. 170–180. ISBN: 978-1-4503-9275-4. DOI: 10.1145/3522664.3528602 [FS22a]

Artificial Intelligence (AI) applications operate under a variety of optimization goals and constraints. On the one hand, they must satisfy use case-specific minimum accuracy and maximum latency demands. On the other hand, they should have low hardware cost and – especially for batch processing – high throughput. They also come with a wide range of AI-capable machine learning methods, corresponding software platforms, optimization methods, and hardware platforms for executing them. So, selecting and configuring software and hardware components that fulfil certain functional properties and satisfy (non-functional) performance requirements is challenging.

Neural Networks (NNs) are a prominent machine learning method for AI solutions, and a core component of the resKIL agricultural AI product line that I designed as part of this thesis. They, too, come in a variety of flavours that affect the availability of software optimization methods and the ability to use hardware accelerators for NN inference. However, existing research on NN performance has focused on optimizing individual neural networks rather than selecting and configuring suitable combinations of software, hardware, and NN components. This section examines whether it is viable to apply SPLE techniques to this domain, and whether it can benefit from associated performance modeling methods such as RMT.

Before diving into the product line itself, Section 8.1.1 introduces the design space of neural network platforms that is relevant here, the motivation and context behind the resKIL product line, and approaches for NN performance modeling. Afterwards, Section 8.1.2 presents the product line and thus shows that domain engineers can formally express the variability within a real-world AI product line as a feature model. Sections 8.1.3 and 8.1.4 follow up with a performance model analysis and find that both software and hardware selection and configuration affect AI performance. Finally, Section 8.1.5 examines related work in the field of performance models for neural networks, and Section 8.1.6 concludes this excursion into AI engineering.

8.1.1 *Introduction*

With appropriate training as well as pre- and post-processing, neural networks are capable of a wide range of tasks such as classifying items for quality control or identifying the position and type of objects in front of a vehicle. The former application associates each input image with a class (e.g. "not ripe yet", "ripe", "rotten") and is commonly known as *classification*. The latter example encompasses two application domains: *semantic segmentation* networks associate each pixel with a confidence score for each known class (e.g. object type or "no known object"), whereas *object detection* networks output (possibly overlapping) rectangular boxes around identified objects, each of which is annotated with a class and a confidence score.

Developers can choose between NN inference frameworks such as *PyTorch*, *TensorFlow* or *TensorFlow Lite* in order to apply neural networks to these tasks. Each framework comes with a different set of optimization options and supported hardware accelerators, and neural network architectures may be available for all or only specific NN frameworks. For instance, TensorFlow Lite – a TensorFlow variant specifically tailored towards resource-constrained embedded platforms – supports *quantization*: replacing 32-bit floating point values and math operations inside the network with 16-bit floats or 8-bit integers to reduce model size and latency at the cost of accuracy.

Finally, at run-time, many neural networks can analyze a single image at a time or a *batch* of multiple images. Batching can help alleviate the overhead of invoking the inference framework and improve resource utilization through parallel data processing, especially when using GPUs or other hardware accelerators.

The *resKIL project*[1], funded by the Federal Ministry of Food and Agriculture (BMEL) via the Federal Office for Agriculture and Food (BLE), is directly connected to this domain. One of its goals is to obtain insights into the effects of hardware, software, and neural network architecture selection on AI performance by exploring, benchmarking, and modeling the design space of agricultural AI applications that build upon neural networks. This differs from the majority of existing AI engineering research, which focuses on the effect of neural network layout on prediction performance [Tan+19; Yu+21; LDL21].

For instance, an AI-enabled harvester may combine on-board cameras with a neural network for semantic segmentation to provide a driving aid that identifies tracks and distinguishes between harvested and non-harvested areas, or to fine-tune harvester operation for optimal yield. It might also use a less compute-intensive object detection network for obstacle detection, or a fast classification network for quality assurance of individual harvested items.

1 "resKIL" refers to "ressourceneffiziente KI für eingebettete Systeme in Landmaschinen", i.e., resource-efficient agricultural AI: ess.cs.uos.de/research/projects/resKIL.

Optimization goals and constraints depend on the use case. Harvester adjustments take several seconds, so low neural network latency is not critical and developers may be able to improve accuracy at the cost of higher latency. Obstacle detection, on the other hand, should react at least as fast as a human driver would, and have as little false-positive and false-negative findings as possible. At the same time, hardware and software selection affect the set of viable neural network architectures and their latency and throughput attributes.

In contrast to performance models for conventional software product lines or embedded peripherals, this application domain covers four interdependent aspects: hardware selection, software selection (inference framework and neural network architecture), static software configuration (model quantization), and run-time software configuration (batch size). The AI and NN performance modeling research that I am aware of only considers a sub-set of this configuration space and does not employ SPLE methodologies.

There are two approaches for building and using performance models in this domain: *black-box* and *white-box* models.

Black-box performance models apply conventional performance modeling techniques to the AI engineering domain. Each neural network architecture, hardware platform, and other tunable element of the AI application is a configuration option. The models are not aware of the internal structure of neural networks or hardware components and treat them as a black box – hence the term black-box modeling.

White-box performance models, on the other hand, take these details into account. They associate each neural network architecture with a multi-dimensional feature vector that covers, for instance, the number of floating point operations, layers, kernel sizes, and other attributes. This allows for performance prediction of unseen architectures, and is commonly used for automatic optimization of AI architectures – also known as *Neural Architecture Search* [Ban+21; Tan+19]. In some cases, they also take hardware capabilities into account, and describe the number of cores or floating point performance [LDL21]. While this makes white-box models more versatile than black-box models, it comes at the cost of requiring a larger set of training samples.

As this thesis focuses on performance modeling for product lines (and configurable systems that behave like SPLs) rather than neural network optimization or neural architecture search, the resKIL product line and associated performance models take a black-box approach.

8.1.2 *The resKIL Product Line*

The resKIL agricultural AI product line has four sources of variability:

- image processing task (classification, object detection, or semantic segmentation) and associated neural network architectures,

- inference framework (TensorFlow, TensorFlow Lite, PyTorch) and platform-specific optimization methods (e.g. quantization or GPU offloading),
- run-time settings (batch size), and
- the hardware platform used for inference.

Hardware platforms include Raspberry Pi 4, three NVIDIA Jetson variants (Nano, Xavier NX, Xavier AGX), Google's Coral EdgeTPU board, and three resKIL-specific evaluation kits. The variability model considers each hardware component as a black box with fixed (and, from the model's point of view, unknown) memory, CPU and GPU configuration. Inference frameworks are free to use all available hardware resources.

The NN architecture line-up consists of 26 KERAS classification architectures, 30 architectures downloaded from TensorFlow hub (18 for object detection, twelve for semantic segmentation), and 26 PyTorch architectures. The toolchain is also compatible with neural networks provided by resKIL project partners – however, as those are proprietary and thus not usable for reproducing results, they are not relevant here.

All KERAS and TensorFlow hub networks work with TensorFlow by default and are converted to TensorFlow Lite for benchmarking and performance model generation. Conversion uses quantization and optimization (`tf.lite.Optimization.DEFAULT`) in three modes: default (partial 8-bit integer), 16-bit float, and full 8-bit integer. Default mode quantizes all network weights to 8-bit integers and uses 32-bit floating point values and math operations for everything else – i.e., it only differs from a TensorFlow model in the way the weights are stored. 16-bit float quantization is recommended for GPU accelerators and replaces all 32-bit float values with 16-bit equivalents. Full integer quantization replaces all values and math operations with 8-bit integers and is required for special-purpose accelerators that only support integer operations.

Default and 16-bit float optimization come in two flavours: one that uses sample images to determine the range of input-dependent math operations and thus reduce the accuracy loss of the quantized network, and one that does not. For full integer quantization, access to sample images is mandatory. So, there are up to five TensorFlow Lite model flavours for each TensorFlow model.

Note that all TensorFlow Lite optimization methods only support subsets of TensorFlow neural network operators. Hence, depending on the neural network architecture in question, some TensorFlow Lite variants are unavailable. For Coral EdgeTPU, there is a sixth flavour that executes full integer models on the on-board *Tensor Processing Unit* (TPU) accelerator rather than on the CPU.

Evaluation data for PyTorch networks and NVIDIA Jetson AGX hardware has been provided by my colleague Matheus Ferraz. It is

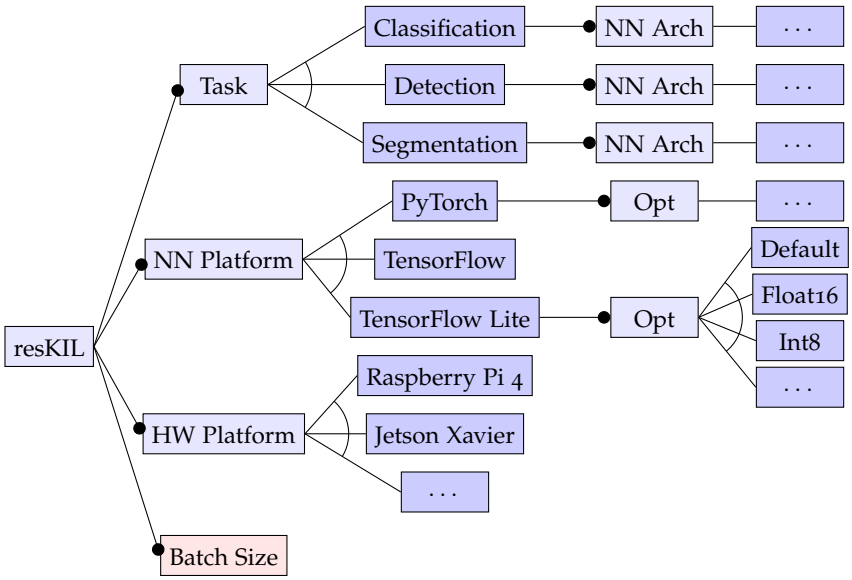


Figure 8.1: Excerpt from the *resKIL* product line's feature model.

part of training and evaluation data, and thus affects complexity and accuracy of the resKIL performance models evaluated in the previous chapter (Section 7.5). The performance model analysis in this section focuses on TensorFlow and TensorFlow Lite without Jetson AGX.

Fig. 8.1 shows the feature model for the resKIL agricultural AI product line, excluding cross-tree constraints, NN architecture names, and some optimizations and hardware platforms for brevity. Cross-tree constraints include PyTorch and TensorFlow only being available on Raspberry Pi 4 and Jetson boards, EdgeTPU-optimized TensorFlow Lite models only working on the Coral EdgeTPU hardware platform, and NN architectures without batching support that always utilize a Batch Size of one.

The resKIL project examines five performance metrics: inference accuracy, inference latency, inference throughput, model size, and peak memory usage during inference.

Latency refers to the time between providing an image or a batch of images to the inference framework and obtaining annotated results, and decides whether a product is fast enough for tasks such as obstacle detection. Model size is most relevant for updating neural networks over slow mobile data links. Memory usage determines, among others, whether the resKIL product can be used together with other applications on the same embedded device. It includes the memory usage of accelerators such as GPUs that share memory with the CPU.

Manual performance annotations provided by a domain expert are not feasible here, as performance attributes depend on complex interactions between hardware platform, NN platform, NN architecture, and batch size. There is no single, well-defined latency or throughput effect of "Raspberry Pi 4" or "Int8". Hence, we now move on

to the benchmark setup used for automated generation of resKIL performance models.

8.1.3 *Benchmark Setup*

As usual, the goal is to select configurations $X = \{\vec{x}_1, \ldots, \vec{x}_n\}$ and, for each performance attribute, obtain corresponding performance measurements $Y = \{y_1, \ldots, y_n\}$ that can be fed into the RMT learning algorithm for performance modeling.

The resKIL configuration space is small enough to allow for a systematic exploration of valid configurations, and only the numeric batch size setting mandates using a sub-set of the configuration space. TensorFlow benchmarks use batch sizes of 1, 2, 4, 8, 16, 32, and 64, if sufficient memory is available. TensorFlow Lite benchmarks are limited to batch sizes of 1, 4, 8, and 16.

Running the complete set of benchmarks for this product line takes about a week and is fully automated. While this is too long for a practical application that relies on rapid turnaround to quickly provide feedback to AI engineers, such an application is not the intended use case here. Instead, the goal within the resKIL project is to thoroughly examine neural network performance on embedded platforms. Considering that recommended sampling methods depend on the performance model type, and that data-efficient model generation is an active research field without clear guidelines [Zha+15; Nai+20], resKIL performance data acquisition errs on the side of caution.

The benchmarks use the latest Python3, TensorFlow (Lite) and PyTorch versions available for the respective hardware platform as of late 2021. Each benchmark takes a series of images, performs preprocessing (scaling and normalization) if necessary, and then uses a pre-trained neural network to perform classification, object detection, or semantic segmentation. Depending on batch size configuration, it processes images one by one or in batches of up to 64 images. In case the number of images is not a whole-numbered multiple of the batch size, it fills the last batch with random data and ignores the corresponding results when evaluating inference accuracy. While doing so, it captures resKIL performance attributes as follows.

- *Latency* (s): `model.predict` or `interpreter.invoke` wall-clock run-time, excluding the first function call (i.e., the first image or the first batch)[2].
- *Throughput* (FPS): Batch size divided by median latency of the current benchmark run.

2 In TensorFlow and PyTorch, the first function call is often slower than follow-up function calls. Benchmarks in the literature typically ignore it; after all, practical applications rarely perform just one inference.

- *Memory Usage* (MB): Memory allocated during benchmark execution, excluding background memory load caused by the operating system. Includes memory overhead for image pre-processing and a few kB of benchmark data storage.
- *Model Size* (MB): Serialized size of the neural network.
- *Accuracy*: F1 Score, defined as the harmonic mean of precision p and recall r: $F1 = 2 \cdot \frac{p \cdot r}{p+r}$. Precision is the ratio of true positives in all classification results that report a specific class; recall is the ratio of true positives in all results that belong to a specific (ground truth) class.

Note that the accuracy metric has a caveat attached. The pre-trained neural networks used in this evaluation come from different sources with, at least in some cases, unknown training data sets and possibly different intended application domains. It is not sensible to compare the accuracy of different neural network architectures under these circumstances.

Due to this, the evaluation in Section 7.5 left out accuracy entirely, and the analysis here only examines the effect of quantization on neural network accuracy. So, it only compares the accuracy of different flavours of the same neural network, making its findings robust against different training setups.

Each benchmark run with N images and batch size B generates a single throughput, memory usage, and model size result, as well as $\lceil \frac{N}{B} \rceil - 1$ latency values and N accuracy scores. It is repeated once so that there are at least two samples for each distinct product line configuration $\vec{x}$. This is relevant for latency, throughput and memory usage, which are influenced by the Linux system running on the evaluated hardware platforms. Model size and accuracy are deterministic.

8.1.4 *Findings*

We will now examine RMT models and corresponding benchmark data to determine how individual resKIL features affect its performance metrics. This also serves as an opportunity to asses RMT interpretability on a real-world product line. Although Fig. 7.6 shows a high complexity score for resKIL RMT models, their shallow tree structure and ULS-fitted leaf functions may be helpful for gaining insights into product line performance regardless.

Batch Size

When it comes to the effect of batch size on NN performance, the RMT model for inference throughput is most interesting. It contains many sub-trees with a structure as shown in Fig. 8.2, indicating that throughput is a linear function of batch size when using TensorFlow,

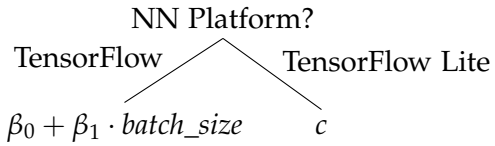


Figure 8.2: A common sub-structure in the resKIL RMT model for neural network throughput.

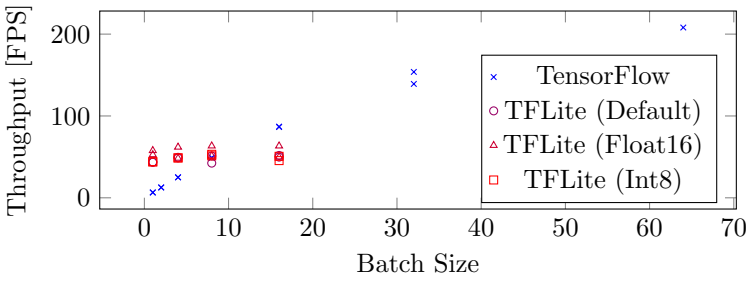


Figure 8.3: MobileNet v3 throughput for TensorFlow and TensorFlow Lite (TFLite) variants on a Jetson Xavier NX board.

and independent of batch size when using TensorFlow Lite. Also, in many cases, the constant part of TensorFlow throughput (β_0) is lower than the constant for TensorFlow Lite throughput (c). So, according to the performance model, TensorFlow Lite often provides higher throughput than TensorFlow when using low batch sizes or no batch processing at all, whereas TensorFlow is better for large batches.

A look at raw benchmark data confirms that the RMT model has correctly captured the underlying hardware behaviour. With TensorFlow, increasing the batch size often causes a linear increase in throughput, whereas it has little effect with TensorFlow Lite – the best observed case is a 50 % increase in throughput when increasing the batch size from one to 16. Also, for low batch sizes, TensorFlow Lite is indeed faster than TensorFlow; the break-even point depends on hardware and NN architecture. This effect affects both CPU-only and GPU- or TPU-accelerated neural network inference.

For example, Fig. 8.3 shows throughput versus batch size for MobileNet v3 image classification on a Jetson Xavier NX board. The three TensorFlow Lite variants achieve nearly constant throughput between 40 and 60 FPS, whereas TensorFlow starts out at 6 and grows up to 200 FPS. In this case, break-even occurs at a batch size of eight.

The effect on memory usage depends on the accelerator. With CPU-only inference, there is a linear relationship to batch size, as each image needs the same amount of memory for processing. On GPU- and TPU-accelerated platforms (Jetson Nano, Jetson Xavier, Coral EdgeTPU), memory usage is nearly constant and higher batch sizes only marginally increase it. Judging from the documentation, this is likely due to eager memory allocation strategies used on these accelerator platforms: on startup, they allocate a fixed-size chunk of

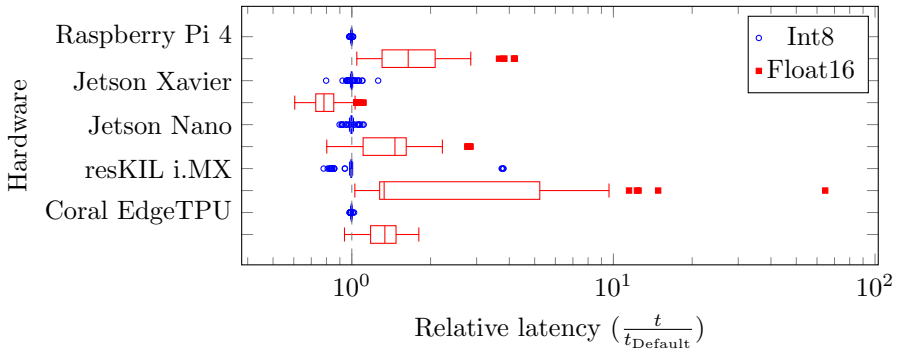


Figure 8.4: Latency distribution of TensorFlow Lite with Int8 and Float16 quantization relative to Default optimization settings. The dashed line indicates equal performance ($\frac{t}{t_{\text{Default}}} = 1$): samples on the left are faster than Default and samples on the right are slower. Each relative latency sample compares benchmark results with identical hardware, neural network architecture, and run-time configuration.

memory, and increase the allocated space if it is insufficient. While at least some of them support on-demand memory allocation, switching to it can impact latency and differs from standard behaviour. Hence, all resKIL benchmarks use the default (eager) memory allocation strategy.

Quantization

The effect of TensorFlow Lite's quantization levels on model size and run-time memory usage is close to what their operand width suggests.

Float16 TensorFlow Lite networks are about half as large as (32-bit) TensorFlow networks, and Int8 networks are again half as large as their Float16 counterparts. Similarly, full 8-bit integer quantization more than halves the memory usage compared to 16-bit float quantization. Here, the difference in memory usage is more pronounced than in model size, likely due to integer-only applications being more compact than code that contains floating-point operations.

As Fig. 8.4 indicates, the influence of quantization on latency is less straightforward. For Raspberry Pi 4, Jetson Nano, and Coral EdgeTPU, partial 8-bit quantization (Default) and full 8-bit quantization (Int8) are on par, and faster than using 16-bit floats. For the resKIL-specific i.MX hardware platform, full 8-bit integer quantization can be faster in some cases. Jetson Xavier NX inference, on the other hand, works best with Float16. Whether the quantization process has access to sample input data or not has no discernible effect on latency.

On the accuracy side, the F1 score of Float16-quantized TensorFlow Lite models is up to ten percentage points higher than that of Default or Int8 variants. This is likely due to the lower dynamic range of 8-bit integers: Float16 models use 16-bit weights and operations, whereas all weights in Default and Int8 models are stored as 8-bit numbers.

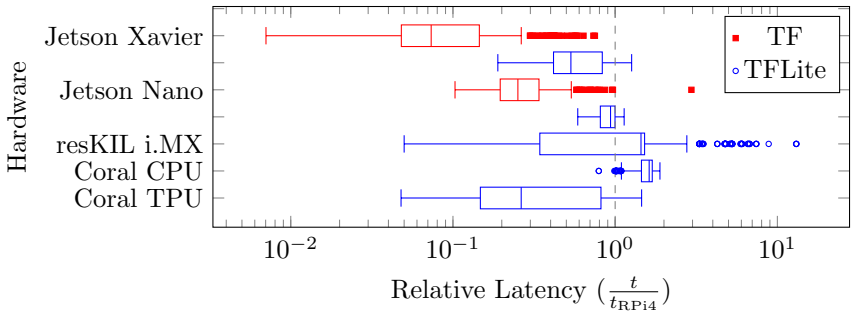


Figure 8.5: Latency distribution of TensorFlow (TF) and TensorFlow Lite (TFLite) by hardware platform relative to Raspberry Pi 4. The dashed line indicates equal performance ($\frac{t}{t_{\text{RPi4}}} = 1$): samples on the left are faster than Raspberry Pi 4 and samples on the right are slower. Each relative latency sample compares benchmark results with identical NN platform, NN architecture, quantization settings, and batch size.

Overall, TensorFlow Lite quantization is a trade-off between accuracy, latency, and memory usage. Whichever method is better depends on hardware platform, neural network architecture, optimization goals, and use-case constraints.

Hardware Choice

Hardware selection is one of the most important aspects in the resKIL product line. It affects the set of available inference platforms and all performance attributes apart from model size. Moreover, it plays a major role in product cost, and – combined with expected sales volume and margin – influences how much money (i.e., time) is available for AI engineering tasks. Fig. 8.5 illustrates its relevance by comparing the inference latency of TensorFlow and TensorFlow Lite on a Raspberry Pi 4 to other hardware platforms.

We see that Jetson Nano (TensorFlow), Jetson Xavier NX (TensorFlow and TensorFlow Lite) and Coral EdgeTPU with TPU-accelerated inference are the only hardware platforms that deliver lower latency than a Raspberry Pi 4 for nearly all evaluated NN configurations. So, these are almost always a better choice – but also more expensive. Meanwhile, Coral EdgeTPU with CPU-only inference is in fact slower than a Raspberry Pi 4.

resKIL i.MX is neither universally better nor worse – depending on software configuration, its relative latency can be anything between ten times lower and ten times higher. This is due to the i.MX hardware platform's NNAPI implementation that is meant to speed up common TensorFlow Lite operators by providing GPU-accelerated implementations. It appears to be incompatible with several modern TensorFlow operators, and has to fall back to CPU execution whenever a neural network uses them.

Considering that Raspberry Pi 4 was the most affordable platform at the time of this evaluation (early 2022), these are notable results. Depending on product line configuration, Raspberry Pi 4 can be far from the least powerful platform, and as such it may be a suitable choice for many affordable NN-based AI solutions. More expensive hardware is not necessarily faster, at least in its default software configuration. Whether switching to a more powerful hardware platform actually improves latency also depends on neural network selection and other software configuration attributes. There is an exception to these observations, though: the most expensive evaluated platform (Jetson Xavier NX) is also the most powerful one.

Summary

We have observed numerous interactions between configurable features and performance attributes of NN-based AI software systems. Virtually any feature, be it hardware selection, quantization settings or batch size, affects performance attributes of neural network inference. Hence, it is worthwhile to explore the configuration space of AI software system components before deciding on specific components and configurations for product development.

RMT models proved to be a helpful utility for finding relationships between product line features and performance attributes. They are interpretable despite their high complexity score. All of the relationships mentioned in this sub-section were confirmed by raw benchmark data.

We will now look into related work that deals with performance aspects of AI software systems.

8.1.5 *Related Work*

Most existing approaches focus on white- rather than black-box models. For instance, Li et al. devise a white-box model for the latency of convolutional neural networks (CNN) on GPUs of embedded NVIDIA Jetson TX boards [Li+21]. The model can predict the often non-linear relationship between convolution parameters and inference latency, but is limited to a specific GPU. It does not consider hardware variations.

Habitat uses transfer learning to predict how GPU hardware affects the training time of neural networks [Yu+21]. When provided with the training latency of a specific NN architecture on a specific GPU, a performance model of the GPU, and a performance model of a new GPU, it predicts the architecture's training latency on the new GPU. Its scope is limited to training latency and GPU hardware. It is not a true white-box model, and – just like resKIL models – cannot predict how changes in neural network architecture affect performance.

Oboe examines the influence of training datasets on accuracy and latency. It uses a combination of bilinear and polynomial models that predict NN performance based on meta-features that it extracts from

neural network architectures and training datasets [Yan+19]. While the performance model deliberately focuses on interpretability, it also does not consider hardware variations.

nn-Meter takes these into account by decomposing TensorFlow Lite-based neural networks into hardware-specific execution units and predicting their inference latency individually [Zha+21]. The authors use a non-linear prediction model and report a root mean square error of less than 6 % for an Adreno 640 GPU. This is typical hardware for low-power embedded devices and smartphones.

Banbury et al. examine NN performance models in the context of *TinyML*: inference on microcontrollers that uses small, 8-bit integer-quantized neural networks. They find that the type and number of arithmetic operations in a neural network architecture is a suitable predictor for model latency [Ban+21]. Their *MicroNets* approach predicts inference latency with simple linear functions that take operation counts as input. They also examine how the AI use case (and, as a consequence of that, the type of NN architecture) affects the performance model, and find that is an important parameter: neural networks for image processing need a different model for predicting latency from operation count than networks for audio processing. So, even white-box performance prediction models must take the AI application domain into account, and should provide domain-specific sub-models or consider the domain as an additional input variable.

Liberis, Dudziak, and Lane note that such a simple model is not applicable to all hardware platforms. They find that, while the number of multiply-accumulate operations is a sensible feature for predicting NN inference performance on microcontroller CPUs, neither this metric nor the number of floating point operations are suitable when using GPU accelerators [LDL21].

Finally, *MNasNet* lies outside the spectrum of white- and black-box modeling. Instead of performance models, it uses a smartphone testbed for NN latency measurements to obtain guaranteed real-world results at the cost of high round-trip times between building an NN architecture and obtaining latency data [Tan+19]. It is suitable for evaluating existing NN architectures, but not helpful for applications like neural architecture search that rely on being able to make hundreds to thousands of performance assessments. Due to its lack of underlying models, it does not provide insights into hardware or NN behaviour.

Overall, we see that while there is a large body of work on white-box AI performance models, it often focuses on individual variability aspects such as neural network selection or hardware changes. Considering that white-box performance modeling has to handle much higher-dimensional input data than the black-box models used in this thesis, this is not surprising. I am not aware of related work that supports all four aspects considered by the resKIL product line: hardware selection, software selection (inference framework and neural network

architecture), static software configuration (model quantization), and run-time software configuration (batch size).

8.1.6 *Conclusion*

This section has presented the resKIL agricultural AI product line and examined how software and hardware configuration changes affect its performance attributes: inference latency, inference throughput, model size, memory usage, and accuracy.

A systematic configuration space exploration and manual interpretation of the resulting RMT performance models have revealed numerous trade-offs between these non-functional properties. For instance, we have seen that batch processing does not improve TensorFlow Lite inference throughput, while at the same time TensorFlow Lite is faster than TensorFlow for small batch sizes. We have also seen that some hardware platforms are consistently better or worse than others, while for other platforms it depends on neural network selection and software configuration.

There are two takeaways within the context of this thesis. First, the use of product line engineering and performance modeling approaches is not limited to pure-software or pure-hardware systems: it pays off for hybrid product lines in the AI engineering field as well. Second, all RMT models for the resKIL product line are interpretable despite their high complexity score, and provide worthwhile and sufficiently accurate insights into its feature-dependent performance attributes.

Having a formal variability model for the resKIL product line gives access to a variety of tooling that builds upon feature models and performance models. For instance, when passed onto a performance-aware configuration frontend, it allows developers and product line engineers to immediately see how exchanging or configuring individual hardware and software components would affect cost and system performance. The next section presents such a frontend.

8.2 PERFORMANCE-AWARE PRODUCT LINE CONFIGURATION

> *Related publication*: Birte Friesel et al. "kconfig-webconf: Retrofitting Performance Models onto Kconfig-Based Software Product Lines". In: *Proceedings of the 26th International Systems and Software Product Line Conference - Volume B*. SPLC '22. Graz, Austria: Association for Computing Machinery, Sept. 2022, pp. 58–61. ISBN: 978-1-4503-9206-8. DOI: 10.1145/3503229.3547026 [Fri+22a]

kconfig-webconf is a configuration frontend for Kconfig-based software product lines and SPL-like software projects such as the Linux kernel or the busybox multi-call binary. It allows users to load a

`Kconfig` variability model and store feature configurations in `.config` files, which can then be turned into concrete products by the build system. In contrast to applications such as the Linux kernel's *Kconfig-qconf*, kconfig-webconf has native support for performance models and can utilize them for performance-aware configuration. This way, whenever engineers have a choice between several configurations that fulfil the same *functional* requirements, they can decide which configuration best suits their needs based on the predicted *non-functional* performance properties.

For example, developers working with resource-constrained embedded devices may choose to work with simple, less efficient algorithms rather than large libraries that are optimized for high throughput. While the latter are faster, they may require so much ROM or RAM that there is no space left for the applications that use them.

In the past, SPLE and performance modeling research has focused on data-efficient benchmarks, model accuracy, and performance-aware variability models [Guo+18; Per+21; Ros+11]. Graphical frontends for performance-aware configuration are rare; the most prominent example that I am aware of is SPL Conqueror [Sie+12b]. Open-source operating systems and other configurable software systems without SPLE background typically do not use them, instead relying on command-line switches or the Kconfig language and associated frontends for variability modeling and configuration [EKS15]. None of these provide native support for performance models.

Although some busybox and Linux kernel features come with in-line annotations that document their effect on binary size or other performance attributes, these are neither complete nor a formal performance model, and occasionally not even correct [Ach+22].

This lack of performance models is likely due to the high cost of adding them to existing projects. SPLE and performance modeling research projects come with their own assumptions about build systems and configuration workflows; these are not necessarily compatible with the way Kconfig and similar tools are used for header- and Makefile-based feature selection in the open-source ecosystem.

I designed kconfig-webconf to minimize this cost by providing a performance-aware drop-in replacement for an already-present Kconfig frontend. Using it in an existing project can be as simple as replacing a *kconfig-qconf* (or similar) call with *kconfig-webconf*. Coupled with dfatool's automatic generation of performance models from Kconfig files (see Section 5.1), this enables developers to add performance models to existing software systems with a minimal amount of work.

kconfig-webconf features include

- showing the estimated performance values of the current configuration,
- annotating feature toggles with their predicted effect on performance attributes, and

- experimental support for performance-aware resolution of feature dependencies, including Pareto optimization of selected performance attributes.

kconfig-webconf[3] and dfatool[4] are available as open-source software licensed under the terms of GNU AGPL and GNU GPL, respectively. Sections 8.2.1 through 8.2.3 explain how kconfig-webconf enables developers to retrofit performance models onto Kconfig-based software product lines. They are divided into concept, implementation, and usage workflow. Section 8.2.4 presents case studies on using kconfig-webconf with busybox and resKIL, followed by an overview over related work in Section 8.2.5 and a conclusion in Section 8.2.6.

8.2.1 *Concept*

In accordance with the findings of Section 6.1, kconfig-webconf uses a separate performance model that is not built into the feature model. It does not require changes to existing Kconfig files, and its methods are in fact independent from the Kconfig language. As long as individual configurations can be transformed into feature vectors, all kconfig-webconf concepts are applicable to any variability modeling language.

As described in Section 2.4, Kconfig supports `bool`, `tristate`, numeric (`hex`/`int`) and `string` entries. Individual features are defined within `config` blocks that contain a `prompt` – entries without a prompt are not shown by configuration frontends and thus do not define product line features. A feature can only be configured if its dependencies are satisfied; otherwise it is not *visible* and does not have a user-defined value.

kconfig-webconf transforms product line instances (`.config` files) into feature vectors as follows. Let n be the number of distinct boolean, tristate and numeric features (i.e., the number of corresponding `config` entries in the Kconfig file). Define an n-dimensional feature vector $\vec{x}$ and associate each feature with one dimension. Map

- each `bool` feature i to $x_i \in \{0, 1\}$ (0 for disabled, 1 for enabled),
- each `tristate` feature i to $x_i \in \{0, 1, 2\}$ (0 for disabled, 1 for module, 2 for enabled), and
- each numeric feature i to $x_i \in \mathbb{N} \cup \{\bot\}$.

Boolean and tristate features that are not visible cannot be enabled, hence $x_i = 0$. Numeric features do not have a well-defined value in that case, so $x_i = \bot$. In line with previous chapters, kconfig-webconf does not consider Kconfig `string` entries.

Each pair of (constant) `Kconfig` file and (product-specific) `.config` file describes a unique product. Hence, a `Kconfig` file, a repository of

3 https://ess.cs.uos.de/git/software/kconfig-webconf
4 https://ess.cs.uos.de/git/software/dfatool

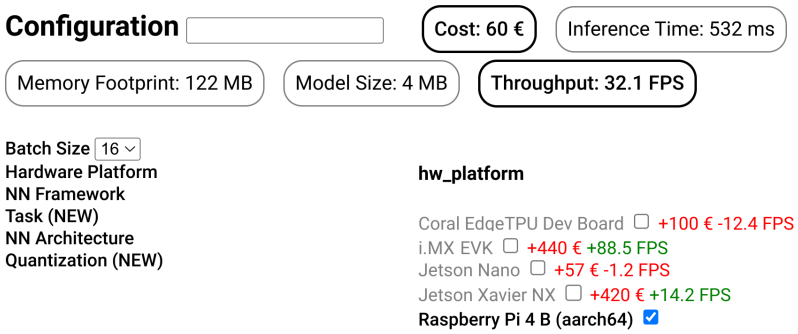


Figure 8.6: kconfig-webconf user interface excerpt with feature model (bottom), performance attributes of current configuration (top), and the effect of boolean feature toggles on the selected performance attributes (right).

`.config` files ($X = \{\vec{x}_1, \ldots, \vec{x}_n\}$), and corresponding machine-readable performance measurements ($Y = \{y_1, \ldots, y_n\}$) are sufficient for learning a performance model such as CART, RMT or XGB. Each performance model predicts a single performance attribute such as resKIL inference latency or Linux kernel size from a feature vector $\vec{x}$.

kconfig-webconf takes a feature model (`Kconfig` file) and a set of performance models as input. Whenever a user changes a feature, it determines the feature vector that corresponds to the new product line configuration, passes it on to the performance models, and displays the predicted performance attributes. It also simulates how toggling individual features would affect product performance, and displays the corresponding performance predictions next to each feature toggle. These feature-specific predictions (e.g. "ROM size +50 kB" or "Latency −1.3 ms") allow users to make informed configuration decisions.

Fig. 8.6 shows how kconfig-webconf visualizes current performance values and feature-specific predictions for the resKIL product line. The user is interested in *cost* and *throughput*, so kconfig-webconf annotates each feature with its effect on these two attributes. We immediately see that the selected platform is already the cheapest one, and that the only platforms that increase throughput come with a markup of more than 400 €.

8.2.2 *Implementation*

kconfig-webconf is a React application with optional online capabilities. It works on any device that provides a reasonably recent web browser, including Windows and Linux computers, tablets, and smartphones. It can be set up with built-in feature and performance models or work with user-provided model files.

When used locally, kconfig-webconf can offer the current configuration for download or utilize a Python helper script to directly save it as a `.config` file, just like a conventional Kconfig frontend. In an online

setup as part of a web site, it can also pass it on to an external build service. This way, users can directly obtain a ready-to-use product (e.g. a compiled application binary) from the configuration frontend.

Outside of user interface and boilerplate code, its implementation has two main components: feature model handling and performance model handling. This sub-section covers these components as well as kconfig-webconf's implementation-specific limitations.

Feature Model

With its ill-defined semantics, Konfig is far from an ideal choice for a feature modeling language [EKS15]. Still, it is widely used, and the Linux Kconfig parser and frontends effectively serve as reference implementations: if the specification is unclear, it is best to imitate their behaviour. This has already led to compatible frontends with different parsers and user interfaces, such as the Python3 *kconfiglib* package used by dfatool. For kconfig-webconf, Linux tooling and kconfiglib serve as reference implementations as well.

kconfig-webconf defines *KNode* objects to express features (`config` entries) and augments them with (conditional) dependencies, defaults, reverse dependencies, and attributes inherited from parent nodes.

Each KNode holds a value that represents the current feature configuration. This configuration may be user-specified, dependency-induced, or a default value; a flag indicates whether it has been set by the user. Just like in other Kconfig frontends, the KNodes form a tree structure that kconfig-webconf builds from explicit (`choice`, `menu`, and `menuconfig` blocks) and implicit (`depends on`) Kconfig annotations. This *menu tree* is the tree structure that users see in the user interface.

kconfig-webconf does not translate Kconfig specifications into logic formulas or extend them otherwise. While such an approach offers interesting opportunities for feature handling and configuration space exploration [Tar+09; Per+21], it is – for now – out of scope. Instead, the goal is for kconfig-webconf to behave just like any other Kconfig frontend, with the optional addition of performance model support.

Performance Models

kconfig-webconf loads performance models from JSON files that contain a Kconfig hash, feature names, and at least one serialized performance prediction model. The Kconfig hash and feature names correspond to the feature model with which the performance models' measurements were acquired. This allows kconfig-webconf to ensure that user-provided Kconfig files and performance models are compatible, and emit a warning if this is not the case.

When using dfatool, the `--export-webconf` switch is sufficient for obtaining a suitable JSON file that contains serialized models for all available performance attributes. dfatool also annotates each per-

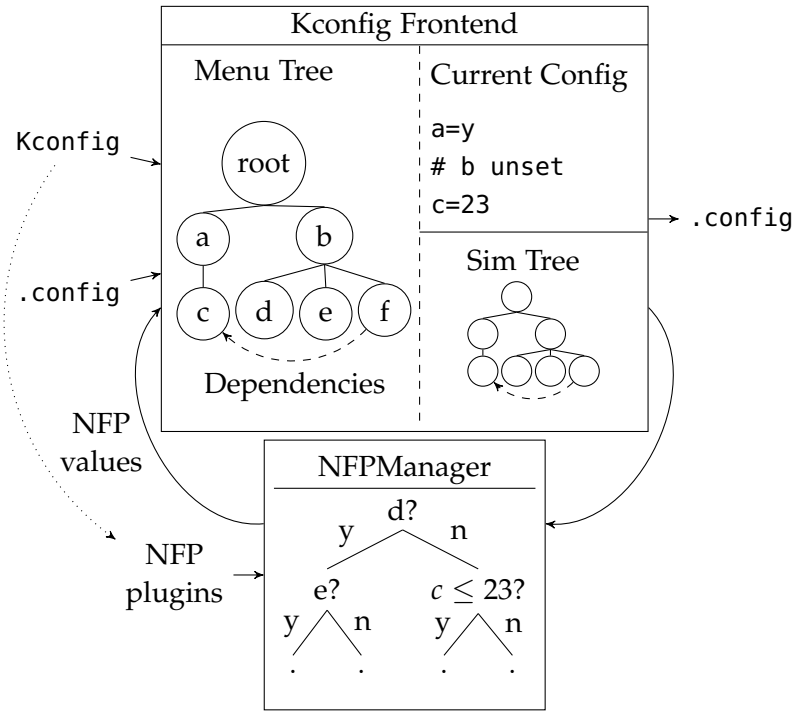


Figure 8.7: Data flow within kconfig-webconf: input (left: `Kconfig`, `.config`, NFP plugins), internal data structures (middle), and output (right: `.config`).

formance attribute with a user-provided description and a flag that indicates whether minimizing or maximizing it is desirable.

Fig. 8.7 shows how Kconfig feature model and performance models interact within kconfig-webconf. First of all, kconfig-webconf loads performance models (*NFP plugins*) into *NFPModel* instances. The NFPModel class is responsible for performance prediction, i.e., taking feature vectors as input and calculating the corresponding performance attributes. At the time of writing this thesis, it supports feature-wise annotations (linear regression), feature-interaction models (linear regression or ULS), CART, LMT, RMT, and XGB. Developers can easily extend kconfig-webconf with additional performance model types by adjusting the NFPModel class.

The *NFPManager* interfaces between user interface, feature model (KNodes), and performance models. It holds references to all loaded peformance models and to two versions of the Kconfig menu tree: *Menu Tree* represents the current configuration, and *Sim Tree* is a copy used for simulating configuration changes.

Whenever a user changes KNodes in the Menu Tree, e.g. by loading a `.config` file or configuring a feature, kconfig-webconf updates *Current Config* and triggers the NFPManager. The manager passes the configuration to all available NFPModel instances for evaluation and shows the predicted performance values in the user interface (Fig. 8.6, top). It also simulates the effect of toggling each visible boolean fea-

ture using the Sim Tree, and passes each corresponding configuration to the performance models as well. kconfig-webconf uses the results of these configuration simulations to annotate each boolean feature with its predicted relative effect on the product line's performance attributes (Fig. 8.6, right).

Limitations

At its core, kconfig-webconf is a by-product of research towards performance-aware product line configuration. It focuses on ways of visualizing performance data and evaluating performance modeling methods by using them in real-world configuration software rather than just comparing prediction errors. As such, kconfig-webconf does not offer full Kconfig compatibility, and is not as polished as other configuration frontends.

Notably, it does not support Kconfig `source` statements. These include additional Kconfig files and are helpful for disaggregating large feature models into individual components. kconfig-webconf cannot handle these statements as it does not have access to the file system. Instead, dfatool provides a `kconfig-expand-includes` utility that resolves `source` statements and combines all components into a single Kconfig file. This file is semantically equivalent, and can be used with kconfig-webconf and any other Kconfig frontend.

Additionally, kconfig-webconf does not support the Kconfig macro pre-processor. This is not a design limitation, but a deliberate omission due to time constraints. Macros within Kconfig files appear to be limited to complex software systems such as the Linux kernel; none of the evaluation targets within this thesis use them.

8.2.3 *Workflow*

In order to use kconfig-webconf for performance-aware product line configuration, an engineer first needs to obtain a suitable set of performance measurements. For software product lines such as busybox, Kratos or Multipass, dfatool provides all tools that are needed. For hybrid product lines such as resKIL, engineers can either provide dfatool-compatible wrapper scripts, or generate benchmark data by themselves and feed it directly to dfatool's model generation script.

Section 5.1 outlines how dfatool supports automated benchmarks of Kconfig-based software product lines; the resKIL analysis in Section 8.1 only used dfatool for model generation. In either case, once benchmark data is available, dfatool's `analyze-kconfig.py --export-webconf model.json` generates performance models and embeds them into a JSON file for use as a kconfig-webconf NFP plugin.

kconfig-webconf provides a `webconf.sh` helper script that allows it to be used as a performance-aware drop-in replacement for existing configuration frontends. So, where users previously called `make`

`config` for configuration and `make` to build a product (e.g. compile a busybox binary), they can now call `make webconf` and `make`. The only requirements for using `webconf.sh` are a suitable make target (or similar build system integration), and ensuring that feature model (Kconfig file) and performance model (JSON file) are available at well-defined locations.

`webconf.sh` starts a Python web server that serves the kconfig-webconf application and its model files, and opens a web browser so that users can access it. kconfig-webconf detects this environment and uses the web server to load `Kconfig`, `.config`, and performance model from the file system. Similarly, its "save configuration" button uses the web server to save `.config` to the file system rather than offering it for download. Thus, kconfig-webonf behaves just like a conventional configuration frontend with added performance model support.

As a fallback mechanism, kconfig-webconf also offers manual upload and download of `Kconfig`, `.config` and performance model files. In this case, there is no need for `webconf.sh` integration, and it does not matter whether kconfig-webconf is served from a web site or the local file system. While less convenient, this method allows developers to easily evaluate kconfig-webconf and its performance model support in arbitrary product lines, without having to add make targets or other build system commands.

Finally, in addition to these two local applications, kconfig-webconf also supports product line configuration and compilation as a web service. Output of `.config` files is not limited to local storage: they can also be passed to a build service which then offers a compiled application (i.e., a ready-to-use product) for download. By serving kconfig-webconf and all associated model files from a web site with an integrated build service, users can configure products without ever having to interact with `make webconf`, `make`, or similar commands.

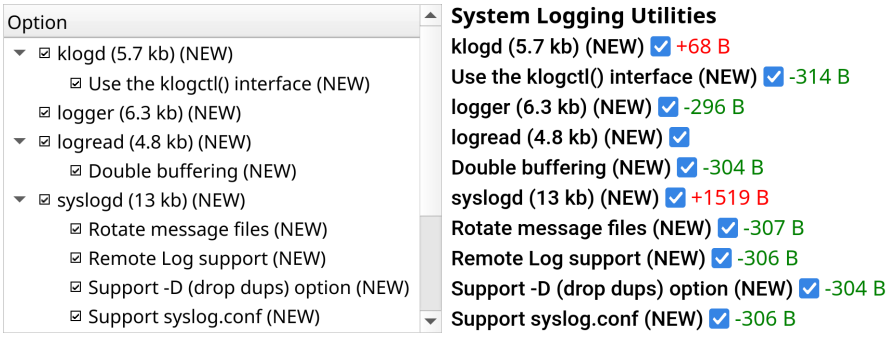
This has two advantages. First, users no longer need to worry about build systems and dependencies, and can even use tablets or smartphones for configuration and build service access. Second, users can configure products without having access to the product line's source code, which may be beneficial for closed-source applications.

8.2.4 *Case Studies*

We will now examine two case studies that utilize kconfig-webconf: busybox, where it serves as a drop-in replacement for an already-present Kconfig frontend, and resKIL.

Busybox

A busybox binary consists of *applets* that provide light-weight implementations of common UNIX applications such as `ls` or `tar`. Applets,



(a) kconfig-qconf (b) kconfig-webconf

Figure 8.8: System logging utilities and their in-line performance annotations in the busybox feature model. The kconfig-webconf interface includes model-based performance annotations (red/green).

applet configuration, and cross-cutting concerns such as logging or debugging features make up its feature model.

Many applets come with in-line annotations that indicate their expected effect on binary size (see Fig. 8.8(a)). Some features also provide annotations that refer to processing speed, e.g. MD5 and SHA3 hash calculation methods that offer different "trade bytes for speed" trade-offs. So, an important aspect of this case study is how kconfig-webconf with its native performance model support compares to in-line annotations such as those within the busybox feature model.

kconfig-webconf shows the predicted absolute performance attributes of the current configuration and annotates each boolean feature with its estimated relative effect on system performance (see Fig. 8.8(b)). In-line annotations only provide the latter, and users have to calculate absolute performance attributes by hand.

Moreover, in case of busybox, the in-line performance model is incomplete. There are no annotations for applet configuration options, and it is not clear whether features such as "syslogd (13 kb)" describe the effect on binary size with the applet's default configuration or with a configuration where all or none of syslogd's optional sub-features are enabled. The same applies to cross-cutting concerns such as circular buffers or debug features that affect almost all applets.

Features that provide in-line annotations referring to processing speed, e.g. MD5 and SHA3 hash calculation methods, suffer from the same limitations. However, their effect is less severe: the trade-offs involved in hash calculation method configuration likely only affect the corresponding applet, so the local view on system performance that they provide is sufficient.

While busybox provides most in-line size and performance annotations as part of the feature name, some are only documented in the feature description and thus easy to miss. kconfig-webconf's performance annotations, by contrast, follow a consistent design and can

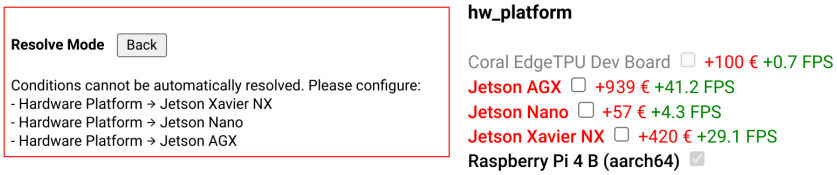


Figure 8.9: kconfig-webconf user interface with performance-aware dependency resolution.

be enabled and disabled for each individual performance attribute. When provided with a suitable performance model (e.g. CART, RMT), they also respect feature interaction: when the user toggles a feature, the performance prediction of all interacting features changes as well. Overall, this makes kconfig-webconf's performance model integration more flexible, more accurate, and easier to use than in-line annotations.

Adding a `make webconf` target to the busybox build system takes less than an hour. The same applies to Multipass and Kratos, with the time requirement shrinking to about ten minutes for engineers who are already familiar with the software system and kconfig-webconf. The performance-aware configuration workflow does not require any additional user input, so using kconfig-webconf as a drop-in replacement works as intended.

resKIL

In the resKIL product line, turning a configuration into a ready-to-use product is a manual process. There is no `make` command or similar that results in an application binary. Instead, feature model and performance models are meant to aid engineers in selecting and configuring hardware and software components for AI products.

Hence, resKIL's performance-aware configuration setup consists of an online kconfig-webconf build that is automatically provided with up-to-date feature and performance models whenever new features and/or new benchmark results become available. This way, there is a single URL that always holds the latest available models. It enables stakeholders such as resKIL project partners to make informed decisions about the configuration space of the resKIL product line. They can immediately see how different hardware or software choices would impact product cost and performance.

kconfig-webconf's guided performance-aware dependency resolution also allows them to select the best configuration that satisfies specific requirements. For instance, if they need to switch to an NN platform that does not support the currently selected hardware, kconfig-webconf will indicate suitable hardware platforms and their performance attributes. Fig. 8.9 shows an example where kconfig-webconf indicates that the currently selected Raspberry Pi 4 B platform and the Coral EdgeTPU Dev Board are not supported. Instead, the

stakeholder has to decide between three NVIDIA Jetson variants with different cost and throughput attributes.

When used with tree-based performance models, kconfig-webconf also has experimental support for automatic performance-aware dependency resolution. This feature is frequently requested by users of product line configuration frontends [HXC12]. The idea is to let users configure the part of the system they care about (i.e., the functional requirements they need), and have the configuration frontend automatically configure all remaining features while optimizing specific performance attributes.

For instance, an AI engineer may be looking for a way of doing image classification with a maximum latency of 20 ms while minimizing hardware cost and maximizing throughput. They do not care about the specific hardware platform and the specific NN architecture. So, they will specify their functional requirement ("Classification"), have kconfig-webconf determine Pareto-optimal configurations with respect to hardware cost and throughput, and then select one of those.

While kconfig-webconf's performance-aware dependency resolution algorithm has not been thoroughly validated yet, it already shows that once performance-aware configuration software is available, performance-aware auto-configuration is within reach.

Finally, the resKIL case study also uncovered aspects of modeling methods that do not become apparent from just SMAPE and related metrics. By design, CART models always interpolate between measurements that were present during model training, whereas LMT and RMT models are capable of extrapolation. In rare cases, this can lead to LMT and RMT models making clearly wrong predictions, especially when queried with numeric configurations that fall outside the training range. The most prominent examples are negative values for latency, throughput, or model size.

While interpolation-only models are safer from this perspective, extrapolation can be a helpful feature, particularly if the range of numeric features used during product line configuration is not known beforehand. It is up to product line engineers to decide which kind of performance model is most appropriate, and whether it might make sense to adjust extrapolating models to avoid predictions that are obviously wrong.

8.2.5 *Related Work*

By themselves, performance prediction models and product line configuration software are popular topics in product line research and literature [Rab+18]. The combination of both – i.e., performance-aware product line configuration software – is less common.

The most prominent example that I am aware of is *SPL Conqueror* [Sie+12b]. It covers the entire life cycle from product line and per-

formance attribute definition over data acquisition to performance-aware configuration and product line optimization. Thanks to this all-encompassing design, it is more powerful than the pair of dfatool and kconfig-webconf. However, it uses a custom variability modeling language and is therefore not suitable as a drop-in replacement for existing configuration interfaces.

ClaferMoo and *TVL* utilize feature models with built-in annotations for performance prediction [Ola+12; Bou+10]. While they are more expressive than in-line annotations such as those used in the busybox Kconfig file, separate performance models are better still in most cases (cf. Section 6.1). Researchers have presented performance-aware configuration frontends for ClaferMoo and TVL in the past, but I am not aware of usable, up-to-date examples. These are also less flexible than kconfig-webconf with its plugin-based performance model support.

Finally, while the *VM* language does not come with a user-facing configuration frontend, its intended application is closely related to performance models [Alf+19]. It has native support for built-in feature attributes, and compatible applications can use these to automatically generate product line instances for configuration space exploration and performance measurements. Performance-aware configuration is out of scope.

8.2.6 *Conclusion*

As we have seen, given appropriate concepts and tools, performance-aware configuration of product lines and software systems in general is not a complex undertaking. With dfatool and kconfig-webconf, the only requirements for retrofitting performance-aware configuration support onto existing Kconfig-based software systems are

- three benchmark commands (`make nfpkeys`, `make randconfig`, `make nfpvalues` – see Section 5.1),
- some time for unattended benchmark execution, and
- a `make webconf` command (or similar) that calls kconfig-webconf.

dfatool and kconfig-webconf also support hybrid product lines such as resKIL. Here, kconfig-webconf enables stakeholders to assess performance attributes of product line configurations and helps them compare the non-functional properties of different configurations with identical functional properties. They can do all of this without having to deal with benchmarks or raw performance models.

In contrast to approaches such as SPL Conqueror and ClaferMoo (and related utilities), kconfig-webconf separates feature model and performance model. It utilizes feature vectors as the only interface between the two. In principle, this allows it to be used with any kind

of feature model from which feature vectors can be extracted, and with any kind of performance model that takes feature vectors as input.

At the time of writing this thesis, kconfig-webconf supports ULS, CART, LMT, RMT, and XGB models. It has been used with busybox, Kratos, Multipass and the resKIL AI product line, and provided valuable insights for developers and stakeholders alike.

The next section returns to performance model generation and analysis, but on a lower level than applications such as busybox and resKIL: it examines data transfer between IoT devices.

8.3 DATA SERIALIZATION FORMATS FOR THE INTERNET OF THINGS

> *Related publication*: Birte Friesel and Olaf Spinczyk. "Data Serialization Formats for the Internet of Things". In: *Electronic Communications of the EASST* 80 (Sept. 2021). DOI: 10.14279/tuj.eceasst.80.1134 [FS21]

In order for a set of computers to exchange data, they have to agree on a common communication method and data layout, and ensure that all data transfers use this layout. The process of converting data objects to a specific layout so that they can be sent to another party is called *serialization* (or *marshalling*), and the transfer itself exchanges *serialized data*. The inverse part – converting serialized data so that the receiving computer can work with – is called *deserialization* (or *unmarshalling*). Within this section, the term *data processing* refers to data serialization and deserialization.

Data (de)serialization may seem like a not particularly noteworthy aspect of wireless IoT product lines. The open-source ecosystem offers a variety of data formats and corresponding libraries, so developers can just pick any that provides the required functional properties.

However, when using resource-constrained low-power devices, non-functional aspects and trade-offs between those become relevant. On the one hand, serialized data should be compact to minimize air-time and thus the energy spent for transmitting and receiving it. On the other hand, data processing itself takes up CPU cycles and thus energy. It is not clear whether engineers should focus on keeping radio transmissions short or whether fast data processing is more important. In addition, they have to take the ROM and RAM usage of data processing libraries into account.

This section explores the effect of microcontroller, radio and serialization library selection on the performance attributes of data processing and transmission: latency, energy usage, ROM usage, and RAM usage. To do so, it augments feature models with PFA-based *behaviour models* that express feature-dependent run-time behaviour. It also examines the insights that can be gained from these models and corresponding benchmarks, such as whether specific data serialization

formats and corresponding libraries are good (or poor) choices in general or whether there is no single best (or worst) option.

The following three sub-sections introduce the evaluated hardware platforms, data serialization formats, and corresponding libraries. After an overview over related work in Section 8.3.4, Section 8.3.5 presents the behaviour model extension, its application for modeling data processing and transfer performance, and the corresponding performance models. Section 8.3.6 follows up with performance insights, and Section 8.3.7 concludes this analysis.

8.3.1 *Hardware Platforms*

This evaluation uses four resource-constrained low-power hardware platforms that are incapable of running Linux or other fully-fledged operating systems. This is a deliberate decision: if a system has sufficient energy for running an entire Linux kernel, the energy spent on data processing is likely less relevant. The platforms range from 8- and 16-bit microcontrollers with external radio modules to a 32-bit *System-on-Chip* (SoC) with built-in WLAN transceiver.

Atmel's *ATMega328P* is an 8-bit microcontroller that is frequently used in Arduino products. With 32 KiB of Flash and 2 KiB of SRAM, it is the most resource-constrained device in the evaluation set. Code and data are organized in a Harvard architecture with distinct address spaces and distinct instructions for Flash and SRAM access. Here, it runs at 16 MHz, which is close to its maximum supported clock frequency of 20 MHz. It can execute one instruction per cycle in most cases, with SRAM access taking two cycles. There are no caches.

Texas Instruments markets its *MSP430FR5994* towards ultra-low-power applications. The microcontroller combines a 16-bit architecture with two 4 KiB regions of SRAM and up to 256 KiB of non-volatile FRAM, both of which can be used for code and data. The usable FRAM range depends on memory model and pointer sizes configured at compile-time. FRAM reads go through a 2-way associative cache with four 64-bit cache lines; writes are not cached. While it supports a CPU clock of up to 16 MHz, FRAM access is limited to 8 MHz and requires a no-operation wait cycle at higher CPU frequencies.

In this evaluation, the CPU runs at 8 MHz and uses a small memory model with 4 KiB SRAM and 48 KiB FRAM. This avoids wait states and allows for using an efficient 16-bit addressing mode as opposed to 20-bit addresses required for full FRAM access. So, all memory usage and CPU cycle measurements are lower bounds for MSP430FR5994 configurations with higher CPU clock or more memory.

espressif's *ESP8266* is an IoT-focused SoC with built-in WLAN transceiver powered by a 32-bit 80 MHz Xtensa LX106 microcontroller. According to official datasheets, it provides applications that run within espressif's software framework with approximately 50 kB of

data SRAM depending on background tasks and WLAN usage. Code is stored on an external flash chip which typically provides 1 to 4 MiB of storage, and is coupled with a 32 KiB SRAM section serving as instruction cache. Based on code annotations, individual program symbols are either loaded and locked into SRAM on startup, or loaded and optionally cached on-demand. The benchmarks in this section load and lock all code into SRAM to achieve stable execution timings.

Finally, the 32-bit *STM32F446RE* microcontroller is based on ARM's Cortex-M architecture. It combines 512 KiB Flash for code and read-only data with 128 KiB SRAM and supports a CPU clock of up to 180 MHz, making it by far the most powerful microcontroller in this ensemble. Here, it runs at 168 MHz.

Two radio modules facilitate the data transfer component of this performance analysis. The *CC1200* is a highly configurable radio operating in the sub-1 GHz ISM bands (see Section 3.6.3) and coupled with ATMega328P, MSP430FR5994, and STM32F446RE. ESP8266 modules come with a built-in 2.4 GHz WLAN transceiver.

8.3.2 *Data Formats*

The data processing part of this evaluation consists of ten data formats:

- JSON (JavaScript Object Notation)[5],
- UBJSON (Universal Binary JSON)[6],
- BSON (Binary JSON)[7],
- CBOR (Concise Binary Object Representation)[8],
- MessagePack[9],
- Protocol Buffers v3[10],
- Cap'n Proto[11],
- Avro[12],
- Thrift[13], and
- XDR (eXternal Data Representation)[14].

5 https://json.org/
6 https://ubjson.org/
7 https://bsonspec.org/
8 https://tools.ietf.org/html/rfc7049
9 https://github.com/msgpack/msgpack/blob/master/spec.md
10 https://protobuf.dev/programming-guides/proto3/
11 https://capnproto.org/
12 https://avro.apache.org/
13 https://github.com/apache/thrift
14 https://tools.ietf.org/html/rfc4506

JSON	{ " d a t a " : [4 2 2 3 7 1 , 2 5 7 7 7] , " s e n s o r " : " g p s " }
UBJSON	{ U 04 d a t a [l 00 06 71 e3 I 64 b1] U 06 s e n s o r S U 03 g p s }
BSON	2e 00 00 00 04 d a t a 00 13 00 00 00 10 30 00 e3 71 06 00 10 31 00 b1 64 00 00 00 02 s e n s o r 00 04 00 00 00 g p s 00 00
CBOR	a2 64 d a t a 82 1a 00 06 71 e3 19 64 b1 66 s e n s o r 63 g p s
MsgPack	82 a4 d a t a 92 ce 00 06 71 e3 cd 64 b1 a6 s e n s o r a3 g p s
ProtoBuf	0a 06 e3 e3 19 b1 c9 01 12 03 g p s
Cap'nProto	10 05 40 02 11 05 14 11 05 22 37 e3 71 06 b1 64 07 g p s
Avro	02 18 06 g p s 04 c6 c7 33 e2 92 03 00
Thrift	0b 00 01 00 00 00 03 g p s 0f 00 02 08 00 00 00 02 00 06 71 e3 00 00 64 b1 00
XDR	00 00 56 3b 00 00 00 02 00 06 71 e3 00 00 64 b1 00 00 00 03 g p s 00

Table 8.1: Serialization example in several data formats. Each one- or two-character group represents a single byte. Red denotes key names or indices, green length indicators, and blue data values.

While XML and EXI (light-weight XML) are also commonly considered in related works, they are not part of this evaluation. Libraries for these data formats tend to have memory requirements that are unsuitable for resource-constrained embedded devices, and they are known to perform no better than JSON and Protocol Buffers [GT11].

All evaluated formats work with objects, that is, collections of key-value pairs with ASCII key names. Values may be strings, numbers, lists, or nested objects. Depending on the format, keys are stored

- as ASCII strings (resulting in self-descriptive, *schema-less* data),
- as indices (requiring an external schema to map indices to keys and value types, also known as *schema-enabled* data), or
- not at all (key and type of a value have to be inferred from its offset in the object; this is also *schema-enabled*).

Schema-less data is easiest to debug and modify, whereas schema-enabled formats are often more compact at the cost of a less human-friendly and less flexible representation.

Support for lists of lists, lists of objects, and mixed-type lists (e.g. a list containing both integer and string values) varies. Evaluation data only contains objects that all data formats can handle.

Table 8.1 gives an overview of the evaluated data formats by showing how they encode the object "data = [422371, 25777], sensor = gps".

It starts with JSON and four JSON variants, all of which encode data in a schema-less manner. The next five data formats are schema-enabled.

Schema-Less Formats

JSON is one of the most ubiquitous data formats to date, with a plethora of libraries available for nearly any programming language and hardware platform. It is the only format in this evaluation that serializes to plain ASCII rather than binary data and does not use type or length indicators. Hence, binary payloads require Base64 encoding, and JSON control characters such as quotation marks must be escaped when used within key names or string values.

UBJSON is precisely what the name suggests: a binary JSON variant. Each UBJSON object can be converted to JSON and vice versa. UBJSON introduces type and (for strings) length annotations, thus removing the need for escaping of control characters.

BSON, *CBOR*, and *MessagePack* provide a superset of JSON that also supports binary data values. BSON was developed specifically for use with the MongoDB database software and supports MongoDB-specific data types such as dates or regular expressions; CBOR and MessagePack focus on concise object representation. All three store not just string lengths, but also the length of objects and lists, allowing for fast partial deserialization when only specific keys are of interest.

CBOR and MessagePack are capable of encoding numeric values below 24 and 16, respectively, in an especially compact manner by storing type information and the encoded number in a single byte. For example, MessagePack prefixes a six-element string with `0xa6`, where the first nibble (`0xa0`) indicates that it is a string and the second nibble (`0x06`) contains its length.

All of these formats are schema-less: key names and (with the exception of JSON) types are encoded within the serialized object.

Schema-Enabled Formats

With Google's *Protocol Buffers v3*, all communication parties must share a common protocol definition (i.e., a schema). Serialized data references individual keys by their index in this definition, thus achieving a more compact representation. Schema definitions are extensible in one direction only. Engineers can add new keys and still perform bidirectional communication with participants unaware of those; changing key types or removing keys is not supported.

Cap'n Proto is essentially Protocol Buffers without explicit serialization and deserialization. It relies on automatically generated accessor functions that always work with serialized data to provide instant (de)serialization at the cost of increased data access latency.

Cap'n Proto is optimized for almost arbitrarily large data sets on 64-bit architectures and uses 64-bit aligned fields internally. Just like

Protocol Buffers, its schema definitions can be extended in one direction only and must be available to all communication parties. It supports optional compression for more efficient data transfer; all measurements here refer to Cap'n Proto with compression enabled.

Avro and *Thrift* focus on *Remote Procedure Calls* (RPC), but can also be used for ordinary data serialization tasks. Avro expects that its schema is either stored with serialized data or exchanged at the beginning of an RPC session. In contrast to Protocol Buffers and Cap'n Proto, the schema is not extensible: serialized data objects contain neither key names nor IDs and exclusively rely on the order of serialized values to determine key and type information. Thrift, on the other hand, encodes field types in serialized data.

Finally, *XDR* is by far the oldest data format in the evaluation set, having been specified in 1987. Like Avro, it encodes neither schema information nor object identifiers. It simply concatenates data values with 32-bit alignment, which in turn means that XDR is optimized for 32-bit CPUs. It is used by the Network File System (NFS).

The next sub-section presents implementations for these data serialization formats, with a focus on libraries that target embedded devices.

8.3.3 *Implementations*

Libraries for embedded systems often focus on a low memory footprint in terms of both code size and run-time memory usage. However, the lengths to which developers go to achieve this and the trade-offs they are willing to make in the process vary. They may optimize libraries towards specific architectures or follow a generic approach.

The intention of this data format analysis is not to explore the maximum amount of efficiency that heavily optimized microcontroller-specific implementations may be able to provide. Instead, it is meant to explore the relation between processing cost and transfer cost provided by existing libraries. Apart from the trivial-to-implement XDR, it only uses the following publicly availale open-source implementations.

ArduinoJSON[15] v6.18 is a mature and well-documented C++ JSON implementation. It supports general-purpose and embedded systems, with a focus on 8-bit AVR microcontrollers such as ATMega328P. All benchmarks in this section enable the `ARDUINOJSON_EMBEDDED_MODE` preprocessor macro to compile ArduinoJSON for embedded-system usage. Note that there are no ESP8266 ArduinoJSON measurements due to incompatibilities with the ESP8266 toolchain.

MPack[16] v1.0 is a C++ MessagePack implementation with a similar level of maturity and platform support. Embedded applications must disable its `MPACK_STDIO` and `MPACK_STDLIB` flags. The benchmarks in

15 https://arduinojson.org/
16 https://github.com/ludocode/mpack

this section also disable its dynamic node API (`MPACK_NODE`), which allocates memory from the heap at run-time. Instead, they rely on MPack's static "expect" API.

Despite its low version number, *NanoPB*[17] v0.4.5 also appears to be mature and well-documented. It provides a C library for using Protocol Buffers on 32-bit microcontrollers and other embedded systems. In contrast to ArduinoJSON and MPack, developers do not have to configure preprocessor options for embedded usage.

The Protocol Buffers specification demands that each key is explicitly marked as *optional* or *required*. While this does not influence serialized object size, it can affect serialization and deserialization cost. All results presented here refer to protocol definitions using optional items; we will discuss the difference to required ones later.

XDR is specified in RFC 4506 and was first documented in RFC 1014. While I am not aware of XDR libraries that target embedded systems, the subset used here is trivial to implement[18].

I did not find usable embedded implementations for UBJSON, BSON, CBOR, Cap'n Proto, Avro, and Thrift. Hence, benchmarks for these are limited to serialized data size measurements provided by the *ubjson*, *bson*, *cbor*, *avro*, and *thrift* Python3 modules as well as the *Cap'n Proto* C++ reference implementation[19].

8.3.4 *Related Work*

It appears that most evaluations of data serialization formats on embedded systems have been published in 2012 or earlier. Considering the pace of IoT technology development, and the fact that the least powerful device considered in these studies is an Android smartphone, their findings do not necessarily apply to today's devices.

First of all, Nurseitov et al. compare JSON and XML for encoding and transmission of 20,000 to 1,000,000 Java objects on an x86 computer. They conclude that, while JSON and XML serialization have similar memory demands, JSON is faster than XML by more than an order of magnitude [Nur+09]. They do not examine deserialization.

Gil and Trezentos also compare JSON and XML, and include Protocol Buffers v2 in addition to those. They examine the time and energy cost of serialization, serialized data size, and data transmission cost when handling SMS, web history, bookmarks, and other user data on an Android 2.1 smartphone [GT11]. All measurements assess plain (uncompressed) and gzip-compressed serialized data.

Accounting for CPU and radio energy, they find uncompressed Protocol Buffers to be most and uncompressed XML to be least efficient to serialize and transmit; deserialization on a powerful server is fastest

17 https://jpa.kapsi.fi/nanopb/
18 https://ess.cs.uos.de/git/bf/xdr-eval
19 https://capnproto.org/cxx.html

for JSON and slowest for XML. This shows that it pays off to consider both serialization and transmission cost. Although serialization energy is minimal for plain JSON or XML (depending on benchmark) and transmission cost is minimal for compressed JSON, they find Protocol Buffers to be most efficient overall.

Sumaray and Makki evaluate JSON, XML and Protocol Buffers as well, and add Thrift to the evaluation set [SM12]. They also examine data processing time and serialized data size on Android smartphones; here the use case consists of objects describing books and video files. They report that Protocol Buffers have the lowest deserialization and second-lowest serialization time next to Thrift, both of which are two to ten times faster than JSON and XML. Results are similar for serialized data size, with Protocol Buffers being most compact. However, the authors note that Protocol Buffers and Thrift are schema-enabled and thus harder to use.

Another analysis from 2012 further extends the evaluation set with Avro, but utilizes an x86 computer rather than a smartphone [Mae12]. Here, the evaluation looks at the effect of different libraries for the same data format in addition to different data formats for the same data objects. The author reports that Protocol Buffers and Avro perform best when it comes to data formats in general, and that results for other libraries are partially inconclusive.

Popić et al. compare the serialized data size of JSON, BSON, Protocol Buffers, and a proprietary binary data format in an *Internet of Vehicles* application [Pop+16]. They collect 51,690 messages in the proprietary data format and convert them to JSON, BSON and Protocol Buffers for evaluation. Message size ranges from hundreds of bytes to several MB; most would not fit into the memory of the devices evaluated in this section. Results indicate that the proprietary format gives the most compact encoding and is closely followed by Protocol Buffers, with JSON and BSON messages being up to one order of magnitude larger.

Finally, in an analysis of protocols and data formats for 5G core networks, Zhang et al. compare JSON, XML, BSON, and Protocol Buffers v3 on a powerful x86 computer [Zha+18]. They find that the Protocol Buffers reference library provides fastest, most compact, and least resource-intensive data processing. Interestingly, although it is a binary format, BSON (de)serialization is reported to be twice as slow as JSON. XML is an order of magnitude slower than Protocol Buffers.

In summary, existing literature often finds that Protocol Buffers (v2 and v3) are the most energy-efficient data serialization format when working with smartphones and more powerful desktop and server computers, whereas XML is seldom a good choice. Before looking into the energy efficiency of data serialization formats on significantly less powerful embedded devices, we will augment feature models with PFA-based *behaviour models*.

8.3.5 *Behaviour Models*

Just like in Section 8.1, where we modeled the resKIL product line, this wireless sensor node product line combines variable hardware and software components, and has both static and run-time variability. However, in contrast to resKIL, its run-time behaviour is so complex that we cannot use a single RMT model per performance attribute. For instance, data transmission energy depends not just on radio configuration, but also on serialized data size, which in turn depends on serialization library configuration and the data objects that are being serialized. These data objects are workload-specific run-time attributes rather than configurable features, and performance models must take this fact into account.

We can tackle this challenge by augmenting the feature model with a *behaviour model* that describes the product line's feature-dependent run-time behaviour. The literature offers a variety of formalisms for this, such as UML state machines [HN96]. Here, we will use PFA models. These are compatible with both feature models and RMT performance models thanks to their common feature vector interface.

Feature vectors in a combined feature and behaviour model reference both product line features (e.g. library selection) and run-time parameters (e.g. data rate or length). Run-time parameters may be part of the feature model and thus allow the product line configuration to specify default values which can be changed at run-time.

However, there are also run-time attributes that are not part of the feature vector despite affecting system performance. For instance, in the x264 product line in Section 2.7.2 and related works [Zha+15; Guo+18; Sie+15; Sie+13], encoder performance depends on the input file being encoded. Yet, the file is not part of the feature model, and all evaluations perform measurements and performance predictions for a single, constant input file. In our case, exchanged data objects and their serialized counterparts have the same role.

Considering the variability in IoT workloads and corresponding data objects, we will not work with constant data. Instead, this section introduces *opaque parameters*: workload-specific run-time attributes that are not part of the feature vector and whose effect on system behaviour and run-time parameters must be observed by means of benchmarks. A performance analysis can then combine performance models (e.g. for transmit latency, which depends on serialized data size) with benchmarks (e.g. to determine serialized data size for various data objects) to gain performance insights with fewer benchmarks than an approach without performance models would require.

Fig. 8.10 shows the combined feature and behaviour model for this wireless sensor node product line. It combines three abstract configuration-time features (Microcontroller, Library and Radio) with three run-time parameters (*dr*, *tp* and *len*) and four opaque parameters

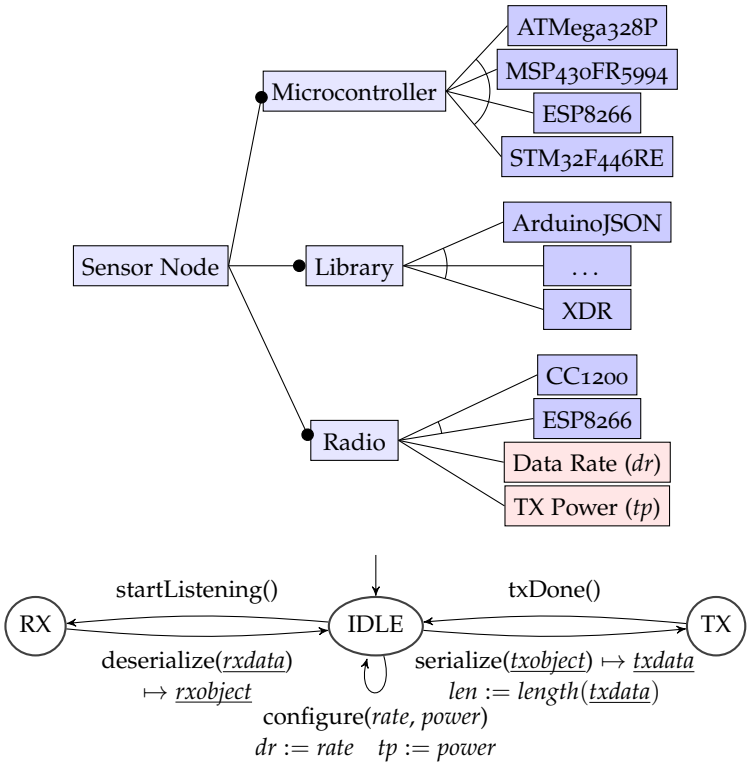


Figure 8.10: Feature model (top) and behaviour model (bottom) for a wireless sensor node product line. *Emphasized* features in the feature model are linked with run-time parameters in the behaviour model. Data rate is given in kbit/s; TX power equals dBm for ESP8266 and $\frac{tp+1}{2} - 18\,\text{dBm}$ for CC1200. *txobject*, *txdata*, *rxdata* and *rxobject* are opaque run-time parameters and not part of the feature model. The run-time parameter *len* depends on *txobject*.

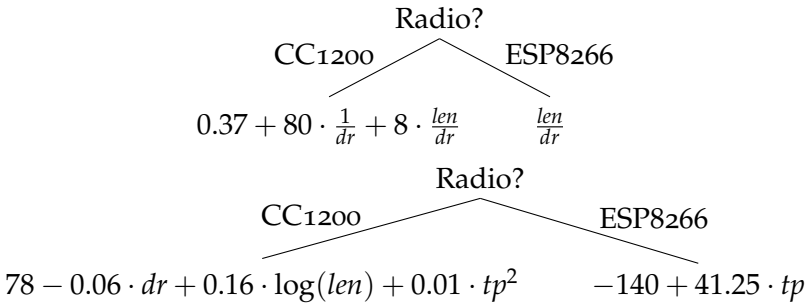


Figure 8.11: RMT models for TX latency [ms] (top) and power [mW] (bottom).

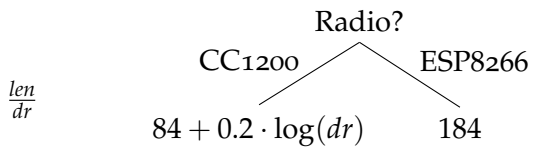


Figure 8.12: RMT models for RX latency [ms] (left) and power [mW] (right).

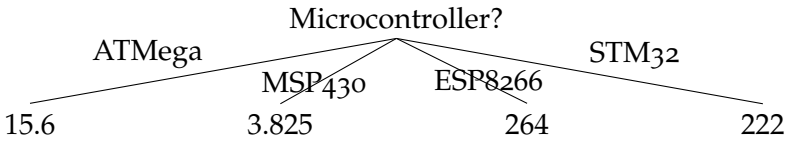


Figure 8.13: RMT model for (de)serialize power usage [mW].

(rxdata, rxobject, txdata and txobject). The feature model includes Data Rate (*dr*) and TX Power (*tp*) so that engineers can specify default values. Serialized data size (*len*) depends on the opaque txobject parameter and is therefore not part of the feature model.

As usual, performance models interface with the product line by means of feature vectors. Note that there is one set of performance models for each PFA state/transition in the behaviour model rather than just a single set (consisting of one model for each performance attribute) for the feature model. Also, in this case, we build performance models from already-available data and use benchmarks only to determine the effect of opaque parameters on product line performance.

The models for TX and RX power usage as well as TX duration (Figures 8.11 and 8.12) rely on the CC1200 energy model presented in Section 3.6.3 and ESP8266 datasheet values. The RX duration model assumes that the time spent in the RX state without receiving data is negligible e.g. due to wake-on-radio mechanisms, hence it only depends on data rate and data size. The data processing model in Fig. 8.13 is based on datasheet values. Note that there is no model for serialize and deserialize latency: these depend on opaque parameters and must be obtained from benchmarks of appropriate workloads. The models for IDLE, startListening and txDone are not relevant here.

The next sub-section examines the energy efficiency of data serialization formats by combining RMT models with benchmarks of data processing latency and serialized data size for domain-specific data objects, thus providing benchmark data for all observations and parameters that depend on opaque parameters. As Protocol Buffers v3 have been available for several years by now, it does not consider v2.

8.3.6 *Observations*

The evaluation dataset consists of real-world JSON payloads sampled from the public `mqtt.eclipse.org` IoT hub as well as objects from the two smartphone-centric studies presented in Section 8.3.4 [Mae12; SM12]. Each object has one to 13 key-value pairs, including lists and sub-objects, with the smartphone study datasets providing the largest and most text-heavy samples. Some objects underwent minor manual adjustments to ensure compatibility with all evaluated data formats.

Benchmarks rely on a script that generates and executes Multipass applications for each combination of hardware platform, serialization

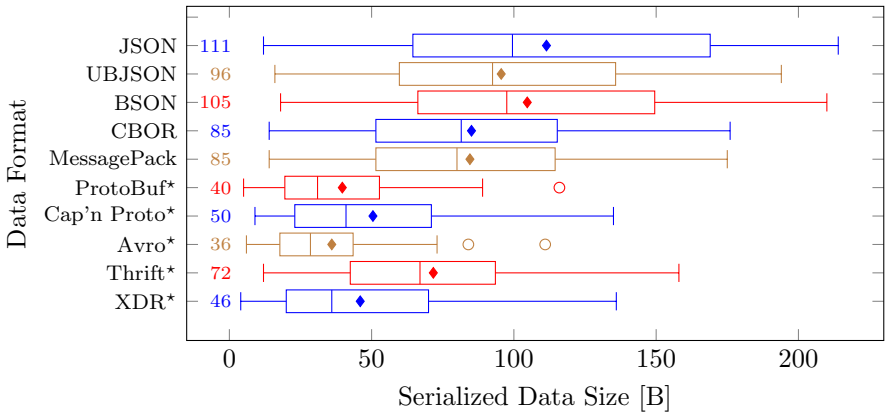


Figure 8.14: Serialized data size of benchmark objects. Schema-enabled data formats are marked with a star (⋆); schema size is not included. Bar elements represent 25th, 50th, and 75th percentile. Mean values are denoted by the diamond symbol (⧫) and also printed on the left.

library, and data object (*txobject*). It uses these to measure (de)serialize latency, serialized data size (*len*), text segment size, and memory usage including run-time stack allocations. Combined with the performance models from Figures 8.11 to 8.13, this allows for reasoning about the energy usage of data processing and wireless data exchange.

All benchmarks are compiled with `-Os -fno-rtti -fno-exceptions`; ATMega328P compilation additionally enables link-time optimizations (`-flto`). We will now examine the findings for serialized data size, data processing and transfer cost, and memory usage.

Data Size

Schema-enabled formats place key names and type information in a separate schema that must be available to each communication party. Once defined and distributed, the schema is constant – hence its size is irrelevant when only looking at serialized data. Schema-less data formats, on the other hand, encode key names and (if applicable) type information within serialized data. With this in mind, the distribution of serialized data size shown in Fig. 8.14 is not surprising: schema-enabled formats are consistently more compact than schema-less ones.

Avro and Protocol Buffers provide the smallest encoding, closely followed by XDR and Cap'n Proto. Thrift is the least compact schema-enabled format and close to the schema-less ones. This is likely due to its deliberate use of fixed-size identifiers as opposed to the variable identifier sizes used by ProtoBuf, Cap'n Proto, and Avro. When it comes to schema-less formats, CBOR and MessagePack are the most space-efficient data formats, while JSON is least compact.

So, from a transfer cost perspective, Avro and Protocol Buffers (for schema-less applications) as well as MessagePack and CBOR (schema-

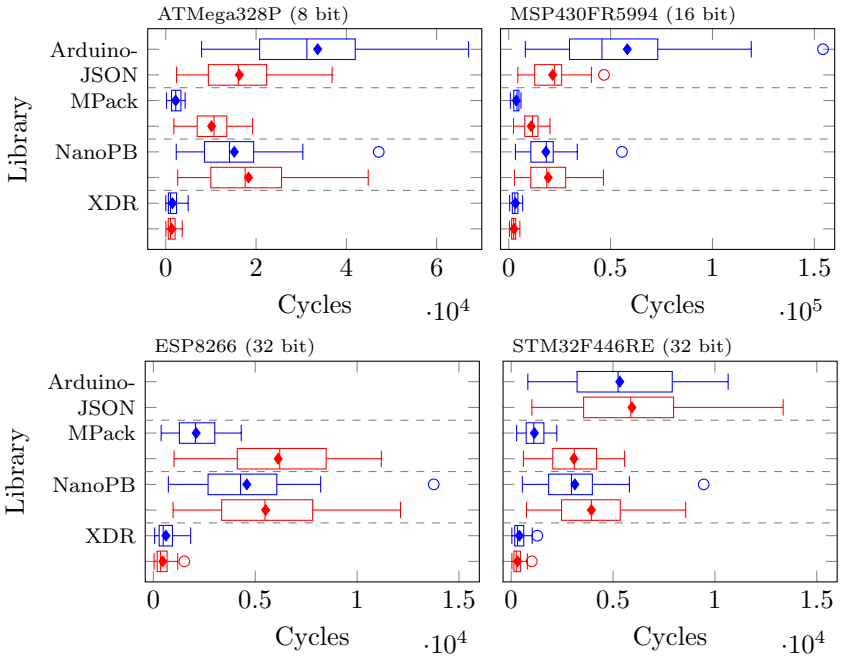


Figure 8.15: Clock cycles for serialization (blue, top) and deserialization (red, bottom). ESP8266 values for ArduinoJSON are not available.

enabled) are promising candidates. These observations are in line with findings reported in related studies [SM12; GT11].

Data Processing Latency

Fig. 8.15 shows the distribution of CPU cycles required for data serialization and deserialization on each evaluated architecture. Note that the individual plots use different X axis scales. Although absolute cycle counts differ, especially between 8-, 16- and 32-bit architectures, the relation between the four libraries is similar on all four platforms.

Again, the main takeaway is not surprising: thanks to its simplistic encoding scheme, XDR is by far the fastest library on all architectures. MPack also performs well, especially for data serialization – it is the fastest schema-less serialization library evaluated here.

NanoPB is slightly slower than MPack, likely due to the fact that MPack can encode and decode objects as they come in, whereas NanoPB has to ensure schema compliance. Indeed, changing the schema so that all keys are marked as required rather than optional – and thus compliance checks are more costly – further decreases NanoPB processing speed by up to 6 %.

Finally, serializing data with ArduinoJSON is consistently slower than using any of the other three libraries, often by a factor of two or more. This appears to be a combination of two factors. First, ArduinoJSON has been specifically optimized for a low code footprint, which often comes at the cost of reduced processing performance. Second,

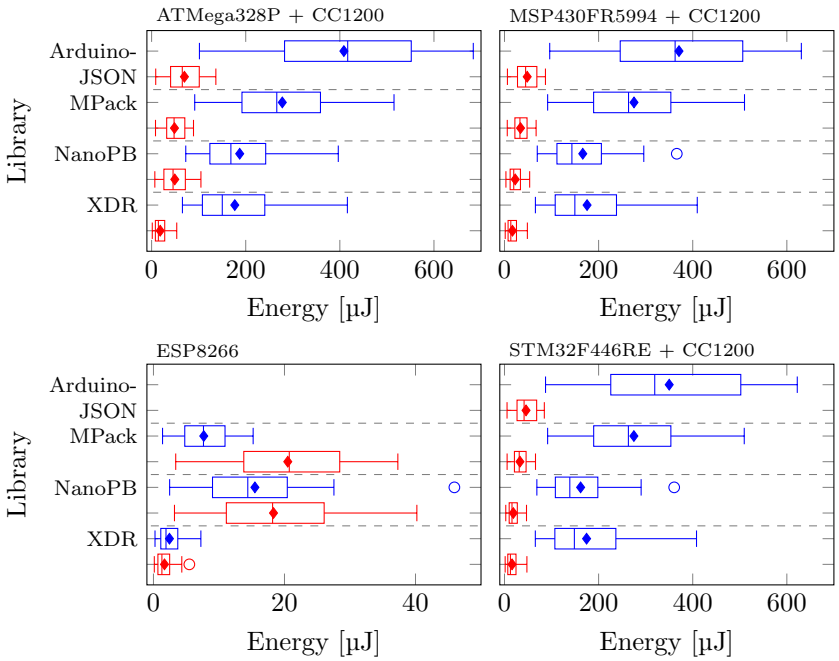


Figure 8.16: Energy spent on data serialization and transmission (blue, top) as well as data reception and deserialization (red, bottom). ESP8266 values for ArduinoJSON are not available.

although ArduinoJSON is schema-less just like MPack, it serializes to ASCII data. This is less compact and thus also takes more time to write out. ArduinoJSON deserialization is similar to NanoPB.

Data Processing and Transfer Cost

In real-world IoT applications, data is typically serialized and then transmitted, or received and then deserialized. Reducing the size (and thus energy usage) of data transfers often comes with additional time (and thus energy usage) for data processing, and vice versa.

Fig. 8.16 shows the total energy usage distribution of data exchange, determined by combining benchmark data (see above) with RMT models (see Figures 8.11 to 8.13). CC1200 radios are configured for 250 kbit/s at 3 dBm and ESP866 for 54 Mbit/s at 15 dBm.

We see that MSP430FR5994 and STM32F446RE behave similarly. Both combine an efficient CPU with a CC1200 radio that has a relatively high per-byte cost, so data processing essentially does not matter and data transfer makes up the bulk of overall energy usage. Hence, the most compact data format (Protocol Buffers) is the most energy-efficient one on these platforms, and NanoPB's time-intensive data processing (see Fig. 8.15) does not matter. XDR is a close contender.

Data processing on ATMega328P is less efficient; here XDR's simplicity pays off and it slightly outperforms NanoPB when it comes to overall energy usage. However, in all three cases, the differences between XDR and NanoPB are minimal. While the majority of data

objects is handled most efficiently by NanoPB on MSP430FR5994 and STM32F446RE, and by XDR on ATMega328P, there are exceptions where the inverse is true.

On all three platforms, MPack is less efficient than NanoPB and XDR, and ArduinoJSON is less efficient than MPack. ArduinoJSON has both the highest data processing cost and the least compact data of all evaluated libraries, so this is not surprising. While MPack processing is more efficient than NanoPB, the difference in serialized data size is so large that this does not help.

The findings for ESP8266 energy usage confirm that spending additional CPU cycles to reduce serialized data size is less helpful on platforms where data processing and transfer cost are close. Here, XDR is clearly the most efficient data format thanks to its fast data processing. NanoPB and MPack come in second.

Considering that the absolute cost of data processing and transfer on ESP8266 is an order of magnitude lower than on the other three platforms, the energy impact of data format choice is less relevant here and engineers may decide to select a format based on other factors.

Memory Usage

Finally, ROM and RAM usage decide whether a data serialization library is a viable candidate for a specific application in the first place.

Here, benchmark data shows that XDR once more benefits from its simplistic design. It has by far the lowest binary size footprint, taking up just 1 to 4 KiB of space in the text segment. Its RAM usage is also low, though not much lower (and, on ATMega328P, even slightly higher) than that of MPack and ArduinoJSON. Considering that XDR builds upon 32-bit values whereas ArduinoJSON and MPack do not, the discrepancy on the 8-bit ATMega328P platform makes sense.

There is no architecture-agnostic second-best option after XDR, and also no library that is clearly the worst choice. Instead, benchmark results show that the CPU architecture which an embedded library is used with can notably affect its memory utilization. XDR and ArduinoJSON ROM usage is largely independent of the architecture, whereas MPack and NanoPB take up more space as architectures become less powerful on the way from STM32F446RE (32 bit) to ATMega328P (8 bit). MPack ROM usage on ATMega328P is an exception caused by ATMega328P being the only platform where code is compiled with link-time optimizations enabled. NanoPB, on the other hand, does not appear to benefit from these optimizations.

8.3.7 *Conclusion*

This section has used *behaviour models* with *opaque parameters* to analyze performance attributes of data exchange on wireless IoT nodes.

Behaviour models augment feature models that express configuration-time variability with PFA models that cover feature-dependent runtime variability; opaque parameters refer to workload-dependent runtime parameters that cannot be used as part of a feature vector.

The performance analysis has combined RMT models for microcontroller processing power, data transfer power and data transfer latency with benchmarks of serialized data size and processing latency. It has shown that behaviour models and opaque parameters pay off: reasoning about workload-dependent energy attributes of hybrid product lines is possible without additional energy benchmarks; latency and data size measurements suffice.

On the product line side, we have seen that platforms with relatively costly data transfer benefit most from compact data. When data transfer cost is closer to data processing cost, the decades-old XDR format is in fact the best option thanks to its simplistic data layout.

More specifically, Protocol Buffers (as implemented by NanoPB) and XDR are the best choice for ATMega328P, MSP430FR5994 and STM32F446RE. On ESP8266, XDR is the clear winner. XDR also has the lowest binary size footprint, whereas NanoPB can take up a sizeable amount of available ROM and RAM resources. However, engineers pay for XDR's low footprint with an absence of extensibility and validation features. Protocol Buffers, on the other hand, provide a mature ecosystem of code generation and schema handling libraries and a process for extending schemas over time.

The overall take-away for data format selection is a clear "it depends". On severely energy- and memory-constrained devices, the simplicity and low memory footprint of XDR will likely offset its usability deficiencies. NanoPB is a more energy-efficient and more feature-rich option when data transfer is costly compared to data processing. On devices that have sufficient energy to spare, the differences in energy usage between data formats are so low that developers should focus on memory and usability aspects instead.

8.4 CHAPTER SUMMARY

This chapter has presented three applications of performance models for configurable software systems and hardware components.

First off, the resKIL agricultural AI product line has shown that it is viable to combine variable software and hardware components in a single product line. This is a diversion from related works in the area of AI engineering, which typically focus on white-box performance models for neural network optimization and do not utilize an SPLE approach. We have also seen that RMT models provide valuable insights into product line behaviour. Thanks to decision nodes that work with categorical features and ULS-fitted formulas in tree leaves,

resKIL models generated with the RMT learning algorithm are shallow and interpretable even when they have a high complexity score.

Similarly, the data format analysis in the previous section has demonstrated that combining performance models with workload-specific benchmarks to analyze trade-offs between data processing and data transfer cost in a wireless sensor node application can provide developers with helpful insights. It has also shown that it pays off to focus on more than one non-functional product line property: on platforms such as ESP8266, the differences in energy utilization are so low that engineers should instead consider memory footprint and usability aspects.

In order to do so, the data format analysis has augmented feature models with behaviour models and introduced opaque parameters. This combination of feature models and PFA models allows for describing the variability of product lines with feature-dependent run-time behaviour, while also taking run-time attributes that cannot be expressed as part of a feature vector into account. Moreover, it enables engineers to reason about energy usage and other performance attributes of a product line by combining RMT models with benchmarks that do not perform additional energy measurements.

Finally, with kconfig-webconf, we have seen that performance models such as RMT can be retrofitted onto existing software systems with minimal effort. By separating feature model and performance model and using feature vectors as the only interface between the two (see Section 6.1), this approach works with any configurable software system, even if it is not developed according to SPLE principles. kconfig-webconf has also shone light on an aspect that error metrics alone do not capture: models that utilize methods such as least-squares regression or ULS can extrapolate beyond performance values that were present in training data. This can improve prediction accuracy outside of the training range, but comes with the risk of unsuitable performance predictions such as negative latency or size values.

Altogether, this chapter provides a positive answer to **RQ4**: product line engineering and performance modeling techniques are indeed applicable to product lines that cover soft- and hardware variability. Morevoer, the techniques developed and presented in this thesis (i.e., dfatool, RMT, and kconfig-webconf) are not limited to those and, in fact, work with nearly any configurable system regardless of whether it follows product line engineering methods. Finally, the resKIL benchmarks have confirmed that RMT provide accurate and interpretable models for manual performance analysis.

This concludes the application side of product lines, performance models, and RMT. The next chapter recapitulates the findings and contributions of this thesis, and examines further research opportunities in the intersection of SPLE, energy models, and RMT development.

9 CONCLUSION

The past chapters have examined performance models for embedded software product lines, ranging all the way from data acquisition over the interface between feature model and performance model to model learning and model applications. After briefly summarizing each chapter in Section 9.1, Section 9.2 recaps the scientific contributions within this thesis. Section 9.3 examines their limitations and opportunities for future research, and Section 9.4 provides closing remarks.

9.1 SUMMARY

Chapter 2 has introduced the notion of product line engineering, software product lines, feature models, and performance models. We have seen that textual variability modeling languages are diverse and wide-spread, even in software projects that are not developed according to SPLE principles. The most prominent example for the latter is the Kconfig language, used by projects such as the Linux kernel and busybox. When it comes to performance models, we have seen that engineers can embed them into the feature model by means of feature- and variant-wise annotations or use a separate model that interfaces with the feature model by means of feature vectors. Both approaches are common in the literature, with separate models often relying on least-squares regression, CART, DECART, or XGB.

Chapter 3 has provided an introduction to energy models for embedded peripherals and device drivers. In contrast to software product lines, these system components have distinct states (operating modes) and transitions between them (driver functions). Energy models take states and transitions into account in order to provide a single, workload-independent model that can be used to estimate energy attributes of arbitrary workloads. We have seen that there is little research on modeling the influence of device- and driver-specific configuration variables, with most approaches using static energy values or manually specified regression templates. My ULS algorithm, which is prior work in the context of this thesis, enables unattended learning of regression templates for numeric configuration variables, but does not support boolean feature toggles.

Following up on this, Chapter 4 has given a detailed motivation of the research questions addressed in this thesis. These refer to energy measurement automation without out-of-band synchronization signals (RQ1), the interface between variability models and performance models (RQ2), a common machine learning algorithm for SPLE and

CPS/IoT performance models (RQ3), and hybrid product lines with software and hardware variability (RQ4). The next section covers those and the corresponding contributions in detail.

Chapter 5 has focused on the data acquisition methods used within this thesis. On the SPLE side, it has described automatic sampling and data acquisition methods for Kconfig-based software product lines. On the CPS/IoT side, and in dealing with the first research question, it has presented a novel synchronization and drift compensation algorithm that exclusively uses in-band signals, an on-board timer, and buffered UART communication for synchronizing benchmark events to energy measurements. This allows for unattended energy measurements even in situations where conventional, out-of-band synchronization is not feasible. An evaluation on the EnergyTrace technology embedded in MSP430FR5994 LaunchPads has shown promising results.

Chapter 6 has examined interfaces between variability models and performance models (RQ2) as well as the differences (and similarities) between SPLE and CPS/IoT performance modeling approaches. It has found that performance models should not be part of the variability model, and that performance modeling for SPL and CPS/IoT applications has more similarities than a glance at the literature might suggest. It has also found that, while most related works consider exclusively boolean or exclusively numeric variability, performance attributes depend on boolean and numeric features in both domains.

Building on top of this, Chapter 7 has introduced my Regression Model Tree data structure and machine learning method. RMT combine and extend ideas from the SPLE and CPS/IoT domains in order to generate interpretable performance models for arbitrary software and hardware systems such as software product lines or configurable embedded peripherals. They do not rely on any kind of manually provided model structure. A quantitative evaluation has shown that RMT achieve lower model error and complexity scores than CART, LMT and XGB when it comes to hybrid and hardware-centric applications with influential numeric features. This gives a positive answer to RQ3.

Chapter 8 has followed up with a qualitative evaluation of model accuracy and interpretability when working with real-world product lines. Here, we have seen that RMT models are useful for reasoning about the effect of software and hardware changes on NN inference performance, and for deciding on data serialization formats in wireless IoT applications. In addition, it has presented and analyzed the hybrid resKIL product line that covers hardware and software variability (RQ4). It has also presented the kconfig-webconf utility for performance-aware configuration of Kconfig-based product lines.

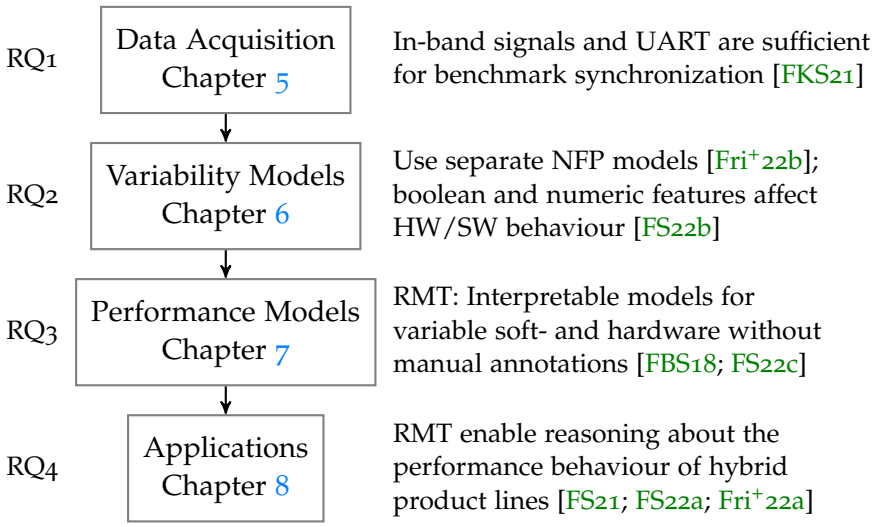


Figure 9.1: Findings when answering the research questions stated in this thesis and corresponding publications in relation to a typical performance model generation and usage workflow.

9.2 CONTRIBUTIONS

The key contributions of this thesis revolve around its research questions; Fig. 9.1 summarizes the findings obtained from answering them. We will now revisit the research questions outlined in Chapter 4 with a focus on how the corresponding results contribute to the state of the art, followed by an overview over contributed software artifacts.

RQ1: Energy Measurement Synchronization

While automated performance data acquisition for software product lines is mostly a question of sampling strategies and appropriate tooling (see Section 5.1), data acquisition for energy models (i.e., energy measurement) is more complex. Here, benchmark events such as driver function calls must be synchronized with energy readings so that each benchmark phase (e.g. a specific hardware state) can be associated with the corresponding interval in the energy measurement. Conventional synchronization approaches use out-of-band signals, which may unavailable for a variety of reasons. Hence, **RQ1** asks: are automated and accurate CPS/IoT energy measurements feasible on hardware that lacks suitable out-of-band synchronization methods?

To this end, Section 5.2 contributes a generic benchmark synchronization and drift compensation algorithm that exclusively relies on in-band signals and on-board timer measurements [FKS21]. All that it needs are a means to generate energy usage spikes with a well-defined duration and intensity for in-band synchronization (e.g. status LEDs), a CPU timer for timing measurements, and a communication channel such as UART. While UART communication is part of the synchroniza-

tion algorithm, it is not used as a time-critical synchronization method, and the algorithm deliberately delays communication to ensure that it does not interfere with the benchmark.

Implementing the algorithm within dfatool and evaluating it on the EnergyTrace technology embedded in TI MSP430FR5994 LaunchPads gave a positive answer to **RQ1**. The algorithm works as intended and its accuracy is mostly limited by EnergyTrace itself, with a maximum measurement error of 53 µA and 0.95 ms.

The state of the art benefits from this in two ways. First, the synchronization component shows that out-of-band synchronization is not a strict necessity for automated energy measurements. Second, the drift compensation component shows that it is viable to augment synchronization mechanisms with automated energy behaviour analysis to reduce timing uncertainty when more than one clock source is involved. Related work that focuses on energy behaviour analysis either does so manually [YF09], or uses it as the only synchronization source and thus lacks a well-defined mapping to benchmark events [Che+17].

As a positive side effect, the algorithm enables automatic generation of energy models with low-cost off-the-shelf hardware components.

RQ2: Linking Variability Models and Performance Models

Performance models can be part of the variability model or kept separate from it. The former relies on a variability modeling language with support for performance annotations; the latter can link any configurable system whose configurations can be expressed as a feature vector $\vec{x}$ with any performance model that provides a function $f(\vec{x})$. Deciding on one of these two approaches is a fundamental question that impacts all aspects of a model's life cycle, including the annotation process, model expressiveness and flexibility, and model maintenance. Given the lack of a consensus on a recommended variability modeling language or performance modeling approach in the SPLE community [Sun+21b], **RQ2** asks: should performance models be integrated into variability models, or should they be separate entities?

After a qualitative and quantitative evaluation, Section 6.1 found that performance models and variability models should be separated [Fri+22b]. This is a valuable insight for two reasons. First, while there is no common variability modeling language in the SPLE community, it shows that researchers need not consider performance modeling aspects when designing or selecting variability modeling languages. Second, when dealing with energy models for embedded peripherals and other product line-like systems that do not come with a formal variability model, Section 6.2 showed that benchmark data consisting of feature vectors and corresponding measurements is sufficient for performance model generation. So, researchers need not define a variability model in order to reap the benefits of performance models.

Complementing this, Sections 6.3 and 6.4 contribute the insight that system performance depends on numeric and boolean features independent of whether the configurable system is part of the SPLE or the CPS/IoT domain [FS22b]. Hence, data acquisition and performance modeling methods should always take both kinds of feature into account, and not blindly assume that only one of them is relevant.

RQ3: Interpretable Machine Learning

At first glance, separate performance models (as recommended in Section 6.1) pose a dilemma for model interpretability. Integrated performance models benefit from the structure provided by the variability model: it makes them easy to interpret and thus helps users with performance-aware system configuration. Separate performance models, on the other hand, are black boxes of arbitrary complexity. In general, they do not offer interpretable links between performance model components and configurable features, and their structure does not (or, in cases where no formal variability model is available, cannot) rely on feature relationships expressed in the variability model.

While tree-based modeling methods such as CART provide an interpretable visual representation of the effect of configurable features on system performance, they are often complex and ill-suited for product lines with influential numeric features. At the same time, interpretable modeling methods from the CPS/IoT domain are unsuitable when dealing with boolean features. Hence, **RQ3** asks: can a common machine learning algorithm for SPLE and CPS/IoT performance models provide lower prediction error and model complexity than conventional approaches, without requiring manually provided domain information or model structure?

Here, my main contribution is the *Regression Model Tree* data structure (Section 7.3) and machine learning algorithm (Section 7.4) [FS22c]. RMT combine regression trees from the SPLE domain with unattended least-squares regression from the CPS/IoT domain [FBS18], and extend the regression tree with non-binary splits in decision nodes for improved interpretability. Decision nodes handle boolean and categorical features, whereas leaf nodes express the effect of numeric features by means of ULS functions.

This allows RMT to turn benchmark data into compact, yet accurate performance models without relying on a user-provided variability model. As Section 7.5 has shown, they excel at dealing with product lines with influential numeric features (SPLE domain) and configurable hardware (CPS/IoT domain), and behave similar to existing modeling methods when numeric features have little influence.

The take-away is that the difference between these two domains is smaller than it seems, and that it pays off to consider approaches from both domains rather than limiting literature reviews to just one

of them. By doing so, RMT provide a positive answer to RQ3 and enable practitioners to obtain interpretable performance models for configurable systems without having to provide a variability model.

RQ4: Hybrid Product Lines

When it comes to applications used in the literature, there is a divide between software-centric product lines with associated performance models and hardware-centric embedded systems with associated energy models. This goes as far as SPLE methods for updating a performance model to reflect hardware changes rather than making those part of the variability model to begin with [Jam+18]. At the same time, use cases such as an agricultural AI product line or a wireless sensor network may contain variability in both software and hardware components. Hence, **RQ4** asks: are product line engineering and performance modeling techniques also applicable to product lines that cover soft- and hardware variability?

Chapter 8 gives a positive answer to this question. The hybrid resKIL product line (Section 8.1) follows SPLE methods, and RMT allow for manual and tool-assisted analysis of its performance behaviour [FS22a]. It deviates from typical approaches in the literature by using an SPL-inspired black-box approach rather than the white-box variant that is common in the NN performance analysis domain.

Similarly, a study on data serialization format selection in wireless IoT sensor nodes (Section 8.3) showed that it pays off to consider variability in hardware and software components when analyzing energy efficiency. Here, the performance analysis augments the feature model with a PFA-based behaviour model and introduces opaque parameters in order to deal with feature-dependent run-time behaviour.

Again, this underlines that there is considerable overlap between the SPLE and CPS/IoT domains. SPLE researchers need not refrain from building feature models that consider hardware variability in addition to software components, and CPS/IoT researchers may well build energy models that incorporate variable software components in addition to hardware configuration and workload.

Artifacts

The findings in this thesis have been supported by four software artifacts, all of which were developed specifically for this purpose and all of which are publicly available under open-source licenses.

First of all, *dfatool*[1] provides unattended benchmark data acquisition and performance model generation for configurable software and hardware systems. It supports Kconfig-based software product lines, embedded peripherals with text-based PFA models, and hybrid

1 https://ess.cs.uos.de/git/software/dfatool

product lines. It implements synchronization and drift compensation (RQ1), utilizes separate performance models to work with arbitrary configurable systems without relying on variability models (RQ2), and supports the RMT machine learning algorithm and data structure (RQ3) in addition to CART, DECART, LMT, and XGB. In addition to this thesis, dfatool has been used in the Bachelor's theses of Janis Falkenhagen, Lennart Kaiser, and Johannes Horas.

The *Multipass*[2] library operating system works in conjunction with dfatool to enable a variety of automated energy measurements. It deliberately lacks multi-tasking support and directly exposes the main function and low-level interrupt control to applications. This way, timing and energy measurements are not affected by timer interrupts or other unwanted sources of operating system noise. It provides helper functions for in-band synchronization and drift compensation (RQ1) and has been used for BME680 energy model generation, the data serialization format analysis presented in Section 8.3, and the Bachelor's theses of Leon Nienhüser, Kevin Lass, and Johannes Horas.

msp430-etv[3] bridges the gap between Multipass and dfatool by providing EnergyTrace measurement automation and, if desired, visualization. It supports a variety of statistical analysis methods including changepoint detection, and has found its use in exercises and lab courses of the Embedded Software Systems group.

Finally, the *kconfig-webconf*[4] utility allows engineers to retrofit performance models and performance-aware configuration onto existing Kconfig-based product lines, and also supports hybrid product lines such as resKIL (RQ4) [Fri+22a]. It has contributed to the success of the resKIL project by visualizing the performance influence of feature toggles to a variety of project partners. The student assistants Kathrin Elmenhorst and, later, Lennart Kaiser supported its development.

9.3 LIMITATIONS AND FUTURE WORK

Naturally, the contributions presented in this thesis are not without limitations or room for future improvements. These mainly concern data acquisition, the RMT data structure and learning algorithm, and performance model applications.

9.3.1 *Data Acquisition*

First of all, by design, automated data acquisition relies on a formal variability model. In case of embedded peripherals, PFA models fulfil this task. Just like SPL feature models, they are not part of the source code, but kept in a separate file – hence, engineers must ensure con-

2 https://ess.cs.uos.de/git/bf/multipass
3 https://ess.cs.uos.de/git/software/msp430-etv
4 https://ess.cs.uos.de/git/software/kconfig-webconf

sistency whenever they make changes to either component. This is known to be a challenging task in SPLE [Tar+09; Tar+11].

However, unlike a feature model, a PFA belongs to a specific device driver and thus to a well-defined (and typically small) set of files. Moving the PFA model into the device driver by means of machine-readable source code annotations would offer an opportunity to ease the task of maintaining consistency. I have already used AspectC++ to show that this is possible in principle with custom C++ annotations [FBS17], but not followed up on it.

Once an energy benchmark has been generated, my drift compensation algorithm for automating energy measurements in the absence of out-of-band synchronization signals relies on a configurable fraction of benchmark events coinciding with observable changes in energy usage (see Section 5.2.5). This limitation is a design issue: the algorithm uses changepoint detection to detect state and transition boundaries within a series of energy readings.

Benchmark events that do not affect energy usage are not an issue: precise synchronization does not matter if energy usage remains unchanged, and users need only specify an appropriate set of interpolation edges (Equation 5.3). However, energy usage spikes that are not part of the benchmark, e.g. due to background tasks or scheduler interrupts, can reduce the drift compensation algorithm's accuracy. While those are undesirable in general, their negative effect on benchmarks with out-of-band synchronization signals is limited to the affected hardware state or transition and thus slightly lower.

This thesis avoids such effects by performing energy measurements with the Multipass library operating system, which does not have a scheduler or other unwanted interrupt sources. Another possible workaround is using the energy spikes caused by UART dumps of timer data for changepoint detection-based drift compensation. While those are delayed so as not to affect the actual energy benchmark, their timing with respect to benchmark events is still well-known.

9.3.2 *Regression Model Trees*

On the modeling side, RMT have a strong focus on interpretability, which is both their strongest asset and their biggest limitation. RMT handle boolean and categorical features only as part of the tree structure and numeric features only within leaf nodes; the learning algorithm deliberately leaves out numeric features when greedily deciding which feature to split on. However, as the loss function still includes uncertainty caused by variable numeric features, the learning algorithm cannot distinguish between variability caused by boolean or categorical features and variability caused by numeric features. The former can be reduced by adding another decision node, whereas the latter mandates a ULS leaf node instead. In the latter case, adding more

decision nodes would only increase model complexity and over-fitting risk without providing an accuracy benefit.

Currently, the learning algorithm does not detect whether the remaining variable boolean and categorical features have (nearly) no influence on the modeled performance attribute and whether it should return a ULS leaf node instead. Instead, it follows the conventional CART approach of continuing to add decision nodes. This is the main reason for the sub-optimal Multipass results discussed in Section 7.5.2.

Extending the learning algorithm to discard irrelevant features before considering split candidates may help here. A similar approach, with a single pre-processing step before tree generation rather than a distinct check for each tree node, has already been shown to decrease model complexity and improve model accuracy when using regression forests to predict Linux kernel size [Ach+22]. Provably optimal and interpretable sparse regression trees are also a promising source of inspiration for improving RMT [Zha+23] – however, these do not support categorical or numeric features yet.

Hyper-parameter tuning (i.e., automatically limiting the tree depth) might also reduce tree complexity in this situation. However, the whole point of RMT is that users should not be burdened with such details, and the algorithm deliberately does not expose hyper-parameters.

Another improvement opportunity is handling of categorical features. Currently, a categorical decision node has one sub-tree for each value of the corresponding feature. While this reflects the variability model's structure, it discards opportunities for reducing model complexity and prediction error by using sub-trees for groups of values.

For instance, in resKIL, Int8- and Default-quantized neural networks often have similar performance attributes, whereas Float16 is slightly different (see Fig. 8.3). An RMT model with a common sub-tree for $x_{\text{Opt}} \in \{\text{Default}, \text{Int8}\}$ and another sub-tree for $x_{\text{Opt}} = \text{Float16}$ would improve interpretability by encoding this behaviour. The challenge here lies in handling the combinatorial explosion of split candidates in the learning process, and deciding between common sub-trees (lower complexity) and separate sub-trees (lower prediction error). LMT-style bottom-up pruning to merge sub-trees is also worth considering.

Lastly, as the x264 performance models in Section 7.5.2 have shown, there are cases where decision nodes that split on numeric features significantly reduce prediction error compared to purely boolean and categorical splits. Allowing the RMT learning algorithm to split on numeric features in special cases would improve its performance on product lines that contain numeric features whose performance influence cannot adequately be described by ULS. Again, the main challenges are split candidate handling during model generation and maintaining an appropriate compromise between accuracy and interpretability – i.e., deciding which conditions justify a tree node that splits on a numeric feature rather than a boolean or categorical one.

9.3.3 *Applications*

When it comes to application domains, I am already following up on the examples presented in Section 8 by utilizing RMT to analyze the performance behaviour of disruptive memory technologies [FLS23; LFS24]. These are neither software product lines nor embedded peripherals, but similar to hybrid product lines: memory performance depends on aspects such as operating system behaviour and software development kits in addition to pure hardware. Moreover, here, performance models also have to take resource contention into account.

The resKIL NN performance analysis in Section 8.1 has examined neural networks as black boxes, unaware of network layers and layer attributes. While this is useful for reasoning about the effect of hardware and software configuration on performance attributes of neural networks, their influence on individual NN layers is also interesting. On the one hand, layer-level performance models can predict performance attributes for NN architectures that were not part of the training set [Ban+21]. On the other hand, they allow for making optimization and offloading decisions at the granularity of NN layers rather than entire networks. Applying RMT to this task would provide interpretable models for NN layer performance and support existing work on low-level optimizations for NN inference [FFS23].

9.4 FINAL REMARKS

As we have seen, performance models for embedded software product lines are an active research area. While there is a plethora of variability modeling languages and performance modeling approaches to choose from, only a subset of those is capable of dealing with complex interactions between boolean and numeric features. An even smaller subset also takes model interpretability into account.

With RMT, this thesis offers an approach for improving model interpretability without sacrificing accuracy when dealing with hybrid and hardware-centric product lines. By combining SPLE and CPS/IoT approaches, RMT have shown that seemingly unrelated fields address common challenges, and that it pays off to take inspiration from them. Thanks to this, RMT models are not limited to product lines, but support arbitrary configurable systems – independent of whether they have variability in hardware, software, both, or something else entirely.

I am certain that there are more opportunities for joint modeling efforts across different research communities, and I am curious to see what will come of it next.

BIBLIOGRAPHY

[Abe+10] Andreas Abele et al. "The CVM Framework – A Prototype Tool for Compositional Variability Management". In: *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems*. Ed. by David Benavides, Don S. Batory, and Paul Grünbacher. VaMoS '10. Linz, Austria: Universität Duisburg-Essen, Jan. 2010, pp. 101–105. DOI: 10.17185/duepublico/47086 (cited on p. 16).

[Ach+22] Mathieu Acher et al. "Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size". In: *Proceedings of the 26th International Systems and Software Product Line Conference - Volume A*. SPLC '22. Graz, Austria: Association for Computing Machinery, Sept. 2022, pp. 85–96. ISBN: 978-1-4503-9443-7. DOI: 10.1145/3546932.3546997 (cited on pp. 2, 102, 143, 144, 146, 161, 197).

[AH09] Jacob Andersen and Morten Tranberg Hansen. "Energy Bucket: A Tool for Power Profiling and Debugging of Sensor Nodes". In: *Proceedings of the 3rd International Conference on Sensor Technologies and Applications*. SENSORCOMM '09. Athens, Greece: IEEE, June 2009, pp. 132–138. ISBN: 978-0-7695-3669-9. DOI: 10.1109/SENSORCOMM.2009.29 (cited on pp. 80, 96).

[Alf+19] Mauricio Alférez et al. "Modeling variability in the video domain: language and experience report". In: *Software Quality Journal* 27.1 (Mar. 2019), pp. 307–347. ISSN: 1573-1367. DOI: 10.1007/s11219-017-9400-8 (cited on p. 171).

[AMS06] Timo Asikainen, Tomi Mannisto, and Timo Soininen. "A Unified Conceptual Foundation for Feature Modelling". In: *Proceedings of the 10th International Software Product Line Conference*. SPLC '06. Baltimore, MD, USA: IEEE, Aug. 2006, pp. 31–40. DOI: 10.1109/SPLINE.2006.1691575 (cited on p. 16).

[Ant+13] Michał Antkiewicz et al. "Clafer Tools for Product Line Engineering". In: *Proceedings of the 17th International Software Product Line Conference Co-Located Workshops*. SPLC '13 Workshops. Tokyo, Japan: Association for Computing Machinery, Aug. 2013, pp. 130–135. ISBN: 978-1-4503-2325-3. DOI: 10.1145/2499777.2499779 (cited on p. 16).

[Ape+13] Sven Apel et al. *Feature-Oriented Software Product Lines*. 1st ed. Springer Berlin, Heidelberg, 2013. ISBN: 978-3-642-37521-7. DOI: 10.1007/978-3-642-37521-7 (cited on p. 11).

[Bąk+16] Kacper Bąk et al. "Clafer: unifying class and feature modeling". In: *Software & Systems Modeling* 15.3 (July 2016), pp. 811–845. ISSN: 1619-1366. DOI: 10.1007/s10270-014-0441-1 (cited on pp. 16, 101).

[Ban+21] Colby Banbury et al. "MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers". In: *Proceedings of the 4th Conference on Machine Learning and Systems*. Ed. by Alex Smola, Alex Dimakis, and Ion Stoica. MLSys '21. Virtual Event, Apr. 2021 (cited on pp. 150, 159, 198).

[BCW10] Kacper Bąak, Krzysztof Czarnecki, and Andrzej Wasowski. "Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled". In: *Proceedings of the 3rd International Conference on Software Language Engineering*. Ed. by Brian Malloy, Steffen Staab, and Mark van den Brand. SLE '10. Eindhoven, The Netherlands: Springer Berlin, Heidelberg, Oct. 2010, pp. 102–122. ISBN: 978-3-642-19439-9. DOI: 10.1007/978-3-642-19440-5_7 (cited on p. 16).

[BFS18] Markus Buschhoff, Birte Friesel, and Olaf Spinczyk. "Energy Models in the Loop". In: *Procedia Computer Science* 130 (2018). Proceedings of the 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018) / the 8th International Conference on Sustainable Energy Information Technology (SEIT 2018) / Affiliated Workshops, pp. 1063–1068. ISSN: 1877-0509. DOI: 10.1016/j.procs.2018.04.154 (cited on pp. viii, ix, 50, 53, 56, 59, 73, 114).

[BGS13] Markus Buschhoff, Christian Günter, and Olaf Spinczyk. "MIMOSA, a Highly Sensitive and Accurate Power Measurement Technique for Low-Power Systems". In: *Proceedings of the 5th International Workshop on Real-World Wireless Sensor Networks*. REALWSN '13. Como, Italy: Springer International Publishing, Sept. 2013, pp. 139–151. ISBN: 978-3-319-03071-5. DOI: 10.1007/978-3-319-03071-5_16 (cited on pp. 58, 64, 97).

[BM01] Peter L. Bartlett and Shahar Mendelson. "Rademacher and Gaussian Complexities: Risk Bounds and Structural Results". In: *Proceedings of the 14th International Conference on Computational Learning Theory*. Ed. by David Helmbold and Bob Williamson. COLT '01. Amsterdam, The Netherlands: Springer Berlin, Heidelberg, Jan. 2001, pp. 224–240.

ISBN: 978-3-540-44581-4. DOI: 10.1007/3-540-44581-1_15 (cited on p. 42).

[Bou+10] Quentin Boucher et al. "Introducing TVL, a Text-based Feature Modelling Language". In: *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems*. Ed. by David Benavides, Don S. Batory, and Paul Grünbacher. VaMoS '10. Linz, Austria: Universität Duisburg-Essen, Jan. 2010, pp. 159–162. DOI: 10.17185/duepublico/47086 (cited on pp. 16, 171).

[BP21] Vaishak Belle and Ioannis Papantonis. "Principles and Practice of Explainable Machine Learning". In: *Frontiers in Big Data* 4 (July 2021). ISSN: 2624-909X. DOI: 10.3389/fdata.2021.688969 (cited on p. 103).

[Bre+84] Leo Breiman et al. *Classification and Regression Trees*. 1st ed. Routledge, 1984. ISBN: 978-1-3151-3947-0. DOI: 10.1201/9781315139470 (cited on pp. 26, 27, 144).

[BSE19] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. "Textual Variability Modeling Languages: An Overview and Considerations". In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*. SPLC '19. Paris, France: Association for Computing Machinery, Sept. 2019, pp. 151–157. ISBN: 978-1-4503-6668-7. DOI: 10.1145/3307630.3342398 (cited on pp. 15, 73).

[Bus19] Markus Buschhoff. "Energy-Aware Design of Hardware and Software for Ultra-Low-Power Systems". Dissertation. Technische Universität Dortmund, Sept. 2019. DOI: 10.17877/DE290R-20241 (cited on pp. 44, 68, 69).

[BV04] Martin Bravenboer and Eelco Visser. "Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions". In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '04. Vancouver, BC, Canada: Association for Computing Machinery, Oct. 2004, pp. 365–383. ISBN: 1581138318. DOI: 10.1145/1028976.1029007 (cited on p. 15).

[CG16] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, CA, USA: Association for Computing Machinery, Aug. 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785 (cited on pp. 33, 144).

[Che+17] Nadir Cherifi et al. "Automatic Inference of Energy Models for Peripheral Components in Embedded Systems". In: *Proceedings of the 5th International Conference on Future Internet of Things and Cloud*. FiCloud '17. Prague, Czech Republic: IEEE, Aug. 2017, pp. 120–127. DOI: 10.1109/FiCloud.2017.53 (cited on pp. 53, 72, 73, 96, 97, 108, 114, 146, 192).

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. "Staged Configuration through Specialization and Multilevel Configuration of Feature Models". In: *Software Process: Improvement and Practice* 10.2 (2005), pp. 143–169. DOI: 10.1002/spip.225 (cited on p. 11).

[DAS21] Johannes Dorn, Sven Apel, and Norbert Siegmund. "Mastering Uncertainty in Performance Estimations of Configurable Software Systems". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Melbourne, Australia: Association for Computing Machinery, Sept. 2021, pp. 684–696. ISBN: 978-1-4503-6768-4. DOI: 10.1145/3324884.3416620 (cited on pp. 139, 144).

[Don+84] Véronique Donzeau-Gouge et al. "Document Structure and Modularity in Mentor". In: *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. SDE '84. New York, NY, USA: Association for Computing Machinery, Apr. 1984, pp. 141–148. ISBN: 978-0-89791-131-8. DOI: 10.1145/800020.808259 (cited on p. 15).

[DS98] Norman R Draper and Harry Smith. *Applied Regression Analysis*. John Wiley & Sons, 1998. ISBN: 978-1-1186-2559-0. DOI: 10.1002/9781118625590 (cited on pp. 22, 40, 104).

[Dut+08] P. Dutta et al. "Energy Metering for Free: Augmenting Switching Regulators for Real-Time Monitoring". In: *Proceedings of the International Conference on Information Processing in Sensor Networks*. IPSN '08. St. Louis, MO, USA: IEEE, Apr. 2008, pp. 283–294. DOI: 10.1109/IPSN.2008.58 (cited on p. 96).

[DZ11] Mian Dong and Lin Zhong. "Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems". In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. MobiSys '11. Bethesda, MD, USA: Association for Computing Machinery, June 2011, pp. 335–348. ISBN: 978-1-4503-0643-0. DOI: 10.1145/1999995.2000027 (cited on p. 52).

[DZT12] Francisco Durán, Steffen Zschaler, and Javier Troya. "On the Reusable Specification of Non-functional Properties in DSLs". In: *Proceedings of the 5th International Conference on Software Language Engineering*. Ed. by Krzysztof Czarnecki and Görel Hedin. SLE '12. Dresden, Germany: Springer Berlin, Heidelberg, Sept. 2012, pp. 332–351. ISBN: 978-3-642-36089-3. DOI: 10.1007/978-3-642-36089-3_19 (cited on p. 2).

[EKS15] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. "Analysing the Kconfig Semantics and Its Analysis Tools". In: *Proceedings of the 14th International Conference on Generative Programming: Concepts and Experiences*. GPCE '15. Pittsburgh, PA, USA: Association for Computing Machinery, Oct. 2015, pp. 45–54. ISBN: 978-1-4503-3687-1. DOI: 10.1145/2814204.2814222 (cited on pp. 18, 161, 164).

[ES15] Holger Eichelberger and Klaus Schmid. "Mapping the design-space of textual variability modeling languages: a refined analysis". In: *International Journal on Software Tools for Technology Transfer* 17.5 (Oct. 2015), pp. 559–584. ISSN: 1433-2787. DOI: 10.1007/s10009-014-0362-x (cited on pp. 16, 73).

[Fal+17] Robert Falkenberg et al. "PhyNetLab: An IoT-Based Warehouse Testbed". In: *Proceedings of the Federated Conference on Computer Science and Information Systems*. FedCSIS '17. Prague, Czech Republic: IEEE, Sept. 2017, pp. 1051–1055. DOI: 10.15439/2017F267 (cited on p. viii).

[Fal14] Robert Falkenberg. "Entwurf eines energiegewahren Treibermodells für eingebettete Betriebssysteme". Masterarbeit. TU Dortmund, Dec. 2014 (cited on p. 54).

[FBS17] Birte Friesel, Markus Buschhoff, and Olaf Spinczyk. "Annotations in Operating Systems with Custom AspectC++ Attributes". In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. PLOS '17. Shanghai, China: Association for Computing Machinery, Oct. 2017, pp. 36–42. ISBN: 978-1-4503-5153-9. DOI: 10.1145/3144555.3144561 (cited on pp. vii, 196).

[FBS18] Birte Friesel, Markus Buschhoff, and Olaf Spinczyk. "Parameter-Aware Energy Models for Embedded-System Peripherals". In: *Proceedings of the 13th International Symposium on Industrial Embedded Systems*. SIES '18. Graz, Austria: IEEE, June 2018. DOI: 10.1109/SIES.2018.8442096 (cited on pp. vii, 6, 24, 191, 193).

[Fei+21] Kevin Feichtinger et al. "TRAVART: An Approach for Transforming Variability Models". In: *Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems*. VaMoS'21. Krems, Austria: Association for Computing Machinery, Feb. 2021. ISBN: 978-1-4503-8824-5. DOI: 10.1145/3442391.3442400 (cited on p. 104).

[FFS23] Matheus Ferraz, Birte Friesel, and Olaf Spinczyk. "Pros and Cons of Executable Neural Networks for Deeply Embedded Systems". In: *Proceedings of the Workshop on Compilers, Deployment, and Tooling for Edge AI*. CODAI '23. Hamburg, Germany: Association for Computing Machinery, Sept. 2023, pp. 16–20. ISBN: 979-8-4007-0337-9. DOI: 10.1145/3615338.3618118 (cited on p. 198).

[FG14] Brittany Finch and William Goh. *MSP430 Advanced Power Optimizations: ULP Advisor Software and EnergyTrace Technology*. Tech. rep. SLAA603. Texas Instruments, June 2014. URL: https://ti.com/lit/an/slaa603/slaa603.pdf (cited on p. 95).

[FKS21] Birte Friesel, Lennart Kaiser, and Olaf Spinczyk. "Automatic Energy Model Generation with MSP430 EnergyTrace". In: *Proceedings of the Workshop on Benchmarking Cyber-Physical Systems and Internet of Things*. CPS-IoTBench '21. Nashville, TN, USA: Association for Computing Machinery, May 2021, pp. 26–31. ISBN: 978-1-4503-8439-1. DOI: 10.1145/3458473.3458822 (cited on pp. vii, 6, 79, 191).

[FLS23] Birte Friesel, Marcel Lütke Dreimann, and Olaf Spinczyk. "A Full-System Perspective on UPMEM Performance". In: *Proceedings of the 1st Workshop on Disruptive Memory Systems*. DIMES '23. Koblenz, Germany: Association for Computing Machinery, Oct. 2023, pp. 1–7. ISBN: 979-8-4007-0300-3. DOI: 10.1145/3609308.3625266 (cited on pp. viii, 198).

[Fri+22a] Birte Friesel et al. "kconfig-webconf: Retrofitting Performance Models onto Kconfig-Based Software Product Lines". In: *Proceedings of the 26th International Systems and Software Product Line Conference - Volume B*. SPLC '22. Graz, Austria: Association for Computing Machinery, Sept. 2022, pp. 58–61. ISBN: 978-1-4503-9206-8. DOI: 10.1145/3503229.3547026 (cited on pp. viii, 6, 160, 191, 195).

[Fri+22b] Birte Friesel et al. "On the Relation of Variability Modeling Languages and Non-Functional Properties". In: *Proceedings of the 26th International Systems and Software Product Line Conference - Volume B*. SPLC '22. Graz, Austria: Association for Computing Machinery, Sept. 2022, pp. 140–144.

ISBN: 978-1-4503-9206-8. DOI: 10.1145/3503229.3547055 (cited on pp. viii, 6, 101, 191, 192).

[Fri17] Birte Friesel. "Automatisierte Verfeinerung von Energiemodellen für eingebettete Systeme". Masterarbeit. Technische Universität Dortmund, Mar. 2017. DOI: 10.17877/DE290R-18206 (cited on pp. 50, 54, 59).

[FS19] Birte Friesel and Olaf Spinczyk. "Poster Abstract: I²C Considered Wasteful: Saving Energy with Host-Controlled Pull-Up Resistors". In: *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*. IPSN '19. Montreal, QC, Canada: Association for Computing Machinery, Apr. 2019, pp. 315–316. ISBN: 978-1-4503-6284-9. DOI: 10.1145/3302506.3312606 (cited on p. vii).

[FS21] Birte Friesel and Olaf Spinczyk. "Data Serialization Formats for the Internet of Things". In: *Electronic Communications of the EASST* 80 (Sept. 2021). DOI: 10.14279/tuj.eceasst.80.1134 (cited on pp. vii, 6, 172, 191).

[FS22a] Birte Friesel and Olaf Spinczyk. "Black-Box Models for Non-Functional Properties of AI Software Systems". In: *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. CAIN '22. Pittsburgh, PA, USA: Association for Computing Machinery, May 2022, pp. 170–180. ISBN: 978-1-4503-9275-4. DOI: 10.1145/3522664.3528602 (cited on pp. viii, 6, 119, 148, 191, 194).

[FS22b] Birte Friesel and Olaf Spinczyk. "Performance is not Boolean: Supporting Scalar Configuration Variables in NFP Models". In: *Tagungsband des FG-BS Frühjahrstreffens 2022*. Hamburg, Germany: Gesellschaft für Informatik e.V., Mar. 2022. DOI: 10.18420/fgbs2022f-03 (cited on pp. vii, 6, 110, 191, 193).

[FS22c] Birte Friesel and Olaf Spinczyk. "Regression Model Trees: Compact Energy Models for Complex IoT Devices". In: *Proceedings of the Workshop on Benchmarking Cyber-Physical Systems and Internet of Things*. CPS-IoTBench '22. Milan, Italy: IEEE, May 2022, pp. 1–6. DOI: 10.1109/CPS-IoTBench56135.2022.00007 (cited on pp. vii, viii, 6, 191, 193).

[Glio7] Martin Glinz. "On Non-Functional Requirements". In: *Proceedings of the 15th International Requirements Engineering Conference*. RE '07. Delhi, India: IEEE, Oct. 2007, pp. 21–26. DOI: 10.1109/RE.2007.45 (cited on p. 1).

[GT11] Bruno Gil and Paulo Trezentos. "Impacts of Data Interchange Formats on Energy Consumption and Performance in Smartphones". In: *Proceedings of the Workshop*

on Open Source and Design of Communication. OSDOC '11. Lisboa, Portugal: Association for Computing Machinery, July 2011, pp. 1–6. ISBN: 978-1-4503-0873-1. DOI: 10.1145/2016716.2016718 (cited on pp. 175, 178, 184).

[Guo+18] Jianmei Guo et al. "Data-Efficient Performance Learning for Configurable Systems". In: *Empirical Software Engineering* 23.3 (June 2018), pp. 1826–1867. ISSN: 1382-3256. DOI: 10.1007/s10664-017-9573-6 (cited on pp. 4, 29, 30, 32, 43, 46, 73, 106, 138, 143, 144, 161, 180).

[Har+16] D.C. Harrison et al. "Busting Myths of Energy Models for Wireless Sensor Networks". In: *Electronics Letters* 52.16 (Aug. 2016), pp. 1412–1414. ISSN: 0013-5194. DOI: 10.1049/el.2016.1591 (cited on pp. 3, 73).

[Her+21] Benedict Herzog et al. "Automated Selection of Energy-Efficient Operating System Configurations". In: *Proceedings of the 12th International Conference on Future Energy Systems*. e-Energy '21. Torino, Italy: Association for Computing Machinery, June 2021, pp. 309–315. ISBN: 978-1-4503-8333-2. DOI: 10.1145/3447555.3465327 (cited on p. 3).

[Her+22] Benedict Herzog et al. "Bears: Building Energy-Aware Reconfigurable Systems". In: *Proceedings of the 12th Brazilian Symposium on Computing Systems Engineering*. SBESC '22. Fortaleza/CE, Brazil: IEEE, Nov. 2022, pp. 1–8. DOI: 10.1109/SBESC56799.2022.9964629 (cited on p. 144).

[HHS19] Timo Hönig, Benedict Herzog, and Wolfgang Schröder-Preikschat. "Energy-Demand Estimation of Embedded Devices Using Deep Artificial Neural Networks". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19. Limassol, Cyprus: Association for Computing Machinery, Apr. 2019, pp. 617–624. ISBN: 978-1-4503-5933-7. DOI: 10.1145/3297280.3297338 (cited on pp. 49, 52, 73).

[HK17] Simon Holmbacka and Jörg Keller. "Workload Type-Aware Scheduling on big.LITTLE Platforms". In: *Proceedings of the 17th International Conference on Algorithms and Architectures for Parallel Processing*. Ed. by Shadi Ibrahim et al. ICA3PP '17. Helsinki, Finland: Springer International Publishing, Aug. 2017, pp. 3–17. ISBN: 978-3-319-65482-9. DOI: 10.1007/978-3-319-65482-9_1 (cited on p. 3).

[HN96] David Harel and Amnon Naamad. "The STATEMATE Semantics of Statecharts". In: *ACM Transactions on Software Engineering and Methodology* 5.4 (Oct. 1996), pp. 293–333. ISSN: 1049-331X. DOI: 10.1145/235321.235322 (cited on p. 180).

[Hur+11] Philipp Hurni et al. "On the Accuracy of Software-Based Energy Estimation Techniques". In: *Proceedings of the 8th European Conference on Wireless Sensor Networks*. EWSN '11. Bonn, Germany: Springer Berlin, Heidelberg, Feb. 2011, pp. 49–64. ISBN: 978-3-642-19186-2. DOI: 10.1007/978-3-642-19186-2_4 (cited on pp. 3, 53).

[HXC12] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. "A User Survey of Configuration Challenges in Linux and eCos". In: *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS '12. Leipzig, Germany: Association for Computing Machinery, Jan. 2012, pp. 149–155. ISBN: 978-1-4503-1058-1. DOI: 10.1145/2110147.2110164 (cited on p. 170).

[Ins13] Texas Instruments. *CC120X Low-Power High Performance Sub-1 GHz RF Transceivers User's Guide*. CC120X. Rev. SWRU346B. 2013 (cited on p. 68).

[Jam+18] Pooyan Jamshidi et al. "Learning to Sample: Exploiting Similarities across Environments to Learn Performance Models for Configurable Systems". In: *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE '18. Lake Buena Vista, FL, USA: Association for Computing Machinery, Nov. 2018, pp. 71–82. ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3236074 (cited on pp. 105, 145, 194).

[Jia+07] Xiaofan Jiang et al. "Micro Power Meter for Energy Monitoring of Wireless Sensor Networks at Scale". In: *Proceedings of the 6th International Symposium on Information Processing in Sensor Networks*. IPSN '07. Cambridge, MA, USA: IEEE, Apr. 2007, pp. 186–195. DOI: 10.1109/IPSN.2007.4379678 (cited on p. 97).

[Jun+12] Wonwoo Jung et al. "DevScope: A Nonintrusive and Online Power Analysis Tool for Smartphone Hardware Components". In: *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '12. Tampere, Finland: Association for Computing Machinery, Oct. 2012, pp. 353–362. ISBN: 978-1-4503-1426-8. DOI: 10.1145/2380445.2380502 (cited on p. 96).

[Kan+90] Kyo Kang et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Nov. 1990. URL: https://insights.sei.cmu.edu/library/feature-oriented-domain-analysis-foda-feasibility-study/ (cited on pp. 11, 12).

[KB11] Mikkel Baun Kjærgaard and Henrik Blunck. "Unsupervised Power Profiling for Mobile Devices". In: *Proceedings of the 8th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Ed. by Alessandro Puiatti and Tao Gu. MobiQuitous '11. Copenhagen, Denmark: Springer Berlin, Heidelberg, Dec. 2011, pp. 138–149. ISBN: 978-3-642-30973-1. DOI: 10.1007/978-3-642-30973-1_12 (cited on p. 53).

[Keh+21] Timo Kehrer et al. "Bridging the Gap Between Clone-and-Own and Software Product Lines". In: *Proceedings of the 43rd International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '21. Madrid, Spain: IEEE, May 2021, pp. 21–25. DOI: 10.1109/ICSE-NIER52604.2021.00013 (cited on p. 9).

[Kep96] Stanislav Keprta. "Non-binary classification trees". In: *Statistics and Computing* 6.3 (Sept. 1996), pp. 231–243. ISSN: 1573-1375. DOI: 10.1007/BF00140867 (cited on p. 121).

[KFE12] Rebecca Killick, Paul Fearnhead, and Idris A Eckley. "Optimal Detection of Changepoints With a Linear Computational Cost". In: *Journal of the American Statistical Association* 107.500 (Dec. 2012), pp. 1590–1598. DOI: 10.1080/01621459.2012.737745 (cited on pp. 86, 92).

[Koh95] Ron Kohavi. "A study of cross-validation and bootstrap for accuracy estimation and model selection". In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. Vol. 2. IJCAI '95. Montreal, QC, Canada: Morgan Kaufmann Publishers Inc., Aug. 1995, pp. 1137–1143. ISBN: 978-1-55860-363-9 (cited on p. 40).

[KZ08] Aman Kansal and Feng Zhao. "Fine-Grained Energy Profiling for Power-Aware Application Design". In: *ACM SIGMETRICS Performance Evaluation Review* 36.2 (Aug. 2008), pp. 26–31. ISSN: 0163-5999. DOI: 10.1145/1453175.1453180 (cited on p. 52).

[LCG12] Yin Lou, Rich Caruana, and Johannes Gehrke. "Intelligible Models for Classification and Regression". In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '12. Beijing, China: Association for Computing Machinery, Aug. 2012, pp. 150–158. ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339556 (cited on p. 42).

[LDL21] Edgar Liberis, Łukasz Dudziak, and Nicholas D. Lane. "μNAS: Constrained Neural Architecture Search for Microcontrollers". In: *Proceedings of the 1st Workshop on Machine Learning and Systems*. EuroMLSys '21. Virtual Event: Association for Computing Machinery, Apr. 2021, pp. 70–79.

ISBN: 978-1-4503-8298-4. DOI: 10.1145/3437984.3458836 (cited on pp. 149, 150, 159).

[LFS24] Marcel Lütke Dreimann, Birte Friesel, and Olaf Spinczyk. "HetSim: A Simulator for Task-based Scheduling on Heterogeneous Hardware". In: *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*. ICPE '24 Companion. London, UK: Association for Computing Machinery, May 2024, pp. 261–268. ISBN: 979-8-4007-0445-1. DOI: 10.1145/3629527.3652275 (cited on pp. ix, 198).

[Li+21] Jinyang Li et al. "Towards an Accurate Latency Model for Convolutional Neural Network Layers on GPUs". In: *Proceedings of the Military Communications Conference*. MILCOM '21. San Diego, CA, USA: IEEE, Nov. 2021, pp. 904–909. DOI: 10.1109/MILCOM52596.2021.9652907 (cited on p. 158).

[Lim+13] Roman Lim et al. "FlockLab: A Testbed for Distributed, Synchronized Tracing and profiling of wireless embedded systems". In: *Proceedings of the 12th International Conference on Information Processing in Sensor Networks*. IPSN '13. Philadelphia, PA, USA: Association for Computing Machinery, Apr. 2013, pp. 153–166. ISBN: 978-1-4503-1959-1. DOI: 10.1145/2461381.2461402 (cited on pp. 80, 97).

[Loh14] Wei-Yin Loh. "Fifty years of classification and regression trees". In: *International Statistical Review* 82.3 (Dec. 2014), pp. 329–348. ISSN: 0306-7734. DOI: 10.1111/insr.12016 (cited on p. 143).

[Mae12] K. Maeda. "Performance Evaluation of Object Serialization Libraries in XML, JSON and Binary Formats". In: *Proceedings of the 2nd International Conference on Digital Information and Communication Technology and its Applications*. DICTAP '12. Bangkok, Thailand: IEEE, May 2012, pp. 177–182. DOI: 10.1109/DICTAP.2012.6215346 (cited on pp. 179, 182).

[Mal+04] Donato Malerba et al. "Top-down induction of model trees with regression and splitting nodes". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.5 (May 2004), pp. 612–625. DOI: 10.1109/TPAMI.2004.1273937 (cited on p. 34).

[McC+11] John C. McCullough et al. "Evaluating the Effectiveness of Model-Based Power Characterization". In: *Proceedings of the USENIX Annual Technical Conference*. USENIX ATC '11. Portland, OR, USA: USENIX Association, June 2011, p. 12 (cited on pp. 4, 53).

[McC11] Trent McConaghy. "High-Dimensional Statistical Modeling and Analysis of Custom Integrated Circuits". In: *Proceedings of the IEEE Custom Integrated Circuits Conference*. CICC '11. San Jose, CA, USA: IEEE, Sept. 2011. DOI: 10.1109/CICC.2011.6055329 (cited on p. 42).

[Mur+12] Rahul Murmuria et al. "Mobile Application and Device Power Usage Measurements". In: *Proceedings of the 6th International Conference on Software Security and Reliability*. SERE '12. Washington, DC, USA: IEEE, June 2012, pp. 147–156. DOI: 10.1109/SERE.2012.19 (cited on p. 53).

[Nai+20] Vivek Nair et al. "Finding Faster Configurations Using FLASH". In: *IEEE Transactions on Software Engineering* 46.7 (July 2020), pp. 794–811. DOI: 10.1109/TSE.2018.2870895 (cited on pp. 11, 138, 145, 153).

[Nur+09] Nurzhan Nurseitov et al. "Comparison of JSON and XML Data Interchange Formats: A Case Study". In: *Proceedings of the 22nd International Conference on Computer Applications in Industry and Engineering*. CAINE '09. San Francisco, CA, USA, Nov. 2009, pp. 157–162. ISBN: 978-1-61567-666-8 (cited on p. 178).

[Ola+12] Rafael Olaechea et al. "Modelling and Multi-Objective Optimization of Quality Attributes in Variability-Rich Software". In: *Proceedings of the 4th International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages*. NFPinDSML '12. Innsbruck, Austria: Association for Computing Machinery, Oct. 2012. ISBN: 978-1-4503-1807-5. DOI: 10.1145/2420942.2420944 (cited on pp. 10, 171).

[Pal+17] James Pallister et al. "Data Dependent Energy Modeling for Worst Case Energy Consumption Analysis". In: *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '17. Sankt Goar, Germany: Association for Computing Machinery, June 2017, pp. 51–59. ISBN: 978-1-4503-5039-6. DOI: 10.1145/3078659.3078666 (cited on p. 52).

[Pat+11] Abhinav Pathak et al. "Fine-Grained Power Modeling for Smartphones Using System Call Tracing". In: *Proceedings of the 6th Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: Association for Computing Machinery, Apr. 2011, pp. 153–168. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966460. URL: https://doi.org/10.1145/1966445.1966460 (cited on p. 53).

[PB46] R. L. Plackett and J. P. Burman. "The Design of Optimum Multifactorial Experiments". In: *Biometrika* 33.4 (June 1946), pp. 305–325. ISSN: 0006-3444. DOI: 10.2307/2332195 (cited on p. 144).

[Per+21] Juliana Alves Pereira et al. "Learning software configuration spaces: A systematic literature review". In: *Journal of Systems and Software* 182 (2021), p. 111044. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111044 (cited on pp. 32, 38, 39, 43, 73, 78, 99, 102, 103, 111, 138, 143, 161, 164).

[Pop+16] Srđan Popić et al. "Performance Evaluation of Using Protocol Buffers in the Internet of Things Communication". In: *Proceedings of the International Conference on Smart Systems and Technologies*. SST '16. Osijek, Croatia: IEEE, Oct. 2016, pp. 261–265. ISBN: 978-1-5090-3720-9. DOI: 10.1109/SST.2016.7765670 (cited on p. 179).

[QBC16] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. "Towards Predicting Feature Defects in Software Product Lines". In: *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. FOSD '16. Amsterdam, The Netherlands: Association for Computing Machinery, Oct. 2016, pp. 58–62. ISBN: 978-1-4503-4647-4. DOI: 10.1145/3001867.3001874 (cited on p. 44).

[Qui+92] John R Quinlan et al. "Learning with Continuous Classes". In: *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*. AI '92. Hobart, Tasmania: World Scientific, Nov. 1992, pp. 343–348. ISBN: 978-981-4536-27-1. DOI: 10.1142/1897 (cited on pp. 34, 144).

[Rab+18] Rick Rabiser et al. "A Study and Comparison of Industrial vs. Academic Software Product Line Research Published at SPLC". In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. SPLC '18. Gothenburg, Sweden: Association for Computing Machinery, Sept. 2018, pp. 14–24. ISBN: 978-1-4503-6464-5. DOI: 10.1145/3233027.3233028 (cited on p. 170).

[Ray+19] Christian Raymond et al. "Genetic Programming with Rademacher Complexity for Symbolic Regression". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. CEC '19. Wellington, New Zealand: IEEE, June 2019, pp. 2657–2664. DOI: 10.1109/CEC.2019.8790341 (cited on pp. 24, 104).

[RK16] Santosh Singh Rathore and Sandeep Kumar. "A Decision Tree Regression based Approach for the Number of Software Faults Prediction". In: *SIGSOFT Software Engineering Notes* 41.1 (Jan. 2016). ISSN: 0163-5948. DOI: 10.1145/2853073.2853083 (cited on pp. 34, 143).

[Rom+22] Dario Romano et al. “Bridging the Gap between Academia and Industry: Transforming the Universal Variability Language to Pure::Variants and Back”. In: *Proceedings of the 26th International Systems and Software Product Line Conference - Volume B*. SPLC ’22. Graz, Austria: Association for Computing Machinery, Sept. 2022, pp. 123–131. ISBN: 978-1-4503-9206-8. DOI: 10.1145/3503229.3547056 (cited on p. 17).

[Ros+11] Marko Rosenmüller et al. “Multi-Dimensional Variability Modeling”. In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS ’11. Namur, Belgium: Association for Computing Machinery, Jan. 2011, pp. 11–20. ISBN: 978-1-4503-0570-9. DOI: 10.1145/1944892.1944894 (cited on pp. 11, 16, 105, 161).

[Sie+11] Norbert Siegmund et al. “Scalable Prediction of Non-functional Properties in Software Product Lines”. In: *Proceedings of the 15th International Software Product Line Conference*. SPLC ’11. Munich, Germany: IEEE, Aug. 2011, pp. 160–169. DOI: 10.1109/SPLC.2011.20 (cited on pp. 16, 105).

[Sie+12a] Norbert Siegmund et al. “Predicting Performance via Automated Feature-Interaction Detection”. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE ’12. Zurich, Switzerland: IEEE, June 2012, pp. 167–177. DOI: 10.1109/ICSE.2012.6227196 (cited on p. 16).

[Sie+12b] Norbert Siegmund et al. “SPL Conqueror: Toward optimization of non-functional properties in software product lines”. In: *Software Quality Journal* 20.3 (Sept. 2012), pp. 487–517. ISSN: 1573-1367. DOI: 10.1007/s11219-011-9152-9 (cited on pp. 4, 14, 16, 101, 161, 170).

[Sie+13] Norbert Siegmund et al. “Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption”. In: *Information and Software Technology* 55.3 (Mar. 2013), pp. 491–507. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2012.07.020 (cited on pp. 46, 180).

[Sie+15] Norbert Siegmund et al. “Performance-Influence Models for Highly Configurable Systems”. In: *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE ’15. Bergamo, Italy: Association for Computing Machinery, Aug. 2015, pp. 284–294. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786845 (cited on pp. 4, 23, 46, 139, 140, 144, 180).

[Sig+17] Lukas Sigrist et al. "Measurement and Validation of Energy Harvesting IoT Devices". In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. DATE '17. Lausanne, Switzerland: IEEE, Mar. 2017. DOI: 10.23919/DATE.2017.7927164 (cited on p. 97).

[Sin+07] Julio Sincero et al. "Is The Linux Kernel a Software Product Line?" In: *Proceedings of the International Workshop on Open Source Software and Product Lines*. Ed. by Frank van der Linden and Björn Lundell. SPLC-OSSPL '07. Kyoto, Japan, Sept. 2007 (cited on pp. 9, 19).

[SL09] Michael Schmidt and Hod Lipson. "Distilling Free-Form Natural Laws from Experimental Data". In: *Science* 324.5923 (Apr. 2009), pp. 81–85. ISSN: 0036-8075. DOI: 10.1126/science.1165893 (cited on p. 23).

[SM12] Audie Sumaray and S. Kami Makki. "A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform". In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. ICUIMC '12. Kuala Lumpur, Malaysia: Association for Computing Machinery, Feb. 2012. ISBN: 978-1-4503-1172-4. DOI: 10.1145/2184751.2184810 (cited on pp. 179, 182, 184).

[Sna+16] Boris Snajder et al. "Wireless sensor node modelling for energy efficiency analysis in data-intensive periodic monitoring". In: *Ad Hoc Networks* 49 (Oct. 2016), pp. 29–41. ISSN: 1570-8705. DOI: 10.1016/j.adhoc.2016.06.004 (cited on p. 49).

[SSS10] Julio Sincero, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. "Approaching Non-functional Properties of Software Product Lines: Learning from Products". In: *Proceedings of the Asia Pacific Software Engineering Conference*. APSEC '10. Sydney, Australia: IEEE, Nov. 2010, pp. 147–155. ISBN: 978-0-7695-4266-9. DOI: 10.1109/APSEC.2010.26 (cited on pp. 3, 10).

[Ste09] Dan Steinberg. "CART: Classification and Regression Trees". In: *The Top Ten Algorithms in Data Mining*. Ed. by Xindong Wu and Vipin Kumar. 1st ed. Chapman & Hall, Apr. 2009, pp. 193–216. ISBN: 978-0-4291-3842-3 (cited on p. 42).

[Sun+21a] Chico Sundermann et al. "Integration of UVL in FeatureIDE". In: *Proceedings of the 25th International Systems and Software Product Line Conference - Volume B*. SPLC '21. Leicester, UK: Association for Computing Machinery, Sept. 2021, pp. 73–79. ISBN: 978-1-4503-8470-4. DOI: 10.1145/3461002.3473940 (cited on p. 17).

[Sun+21b] Chico Sundermann et al. "Yet Another Textual Variability Language? A Community Effort towards a Unified Language". In: *Proceedings of the 25th International Systems and Software Product Line Conference - Volume A*. SPLC '21. Leicester, UK: Association for Computing Machinery, Sept. 2021, pp. 136–147. ISBN: 978-1-4503-8469-8. DOI: 10.1145/3461001.3471145 (cited on pp. 17, 99, 100, 102, 192).

[SUP21] Marco Schaarschmidt, Michael Uelschen, and Elke Pulvermüller. "Towards Power Consumption Optimization for Embedded Systems from a Model-driven Software Development Perspective". In: *Proceedings of the 16th International Conference on Software Technologies*. Ed. by Hans-Georg Fill, Marten van Sinderen, and Leszek A. Maciaszek. ICSOFT '21. Virtual Event: Springer International Publishing, July 2021, pp. 117–142. ISBN: 978-3-031-11513-4. DOI: 10.1007/978-3-031-11513-4_6 (cited on p. 73).

[Tan+19] Mingxing Tan et al. "MnasNet: Platform-Aware Neural Architecture Search for Mobile". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. CVPR '19. Long Beach, CA, USA: IEEE, June 2019, pp. 2815–2823. DOI: 10.1109/CVPR.2019.00293 (cited on pp. 149, 150, 159).

[Tar+09] Reinhard Tartler et al. "Dead or Alive: Finding Zombie Features in the Linux Kernel". In: *Proceedings of the 1st International Workshop on Feature-Oriented Software Development*. FOSD '09. Denver, CO, USA: Association for Computing Machinery, Oct. 2009, pp. 81–86. ISBN: 978-1-6055-8567-3. DOI: 10.1145/1629716.1629732 (cited on pp. 164, 196).

[Tar+11] Reinhard Tartler et al. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem". In: *Proceedings of the 6th Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: Association for Computing Machinery, Apr. 2011, pp. 47–60. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966451 (cited on pp. 43, 73, 79, 196).

[Tër+22] Xhevahire Tërnava et al. "On the Interaction of Feature Toggles". In: *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*. VaMoS '22. Florence, Italy: Association for Computing Machinery, Feb. 2022. ISBN: 978-1-4503-9604-2. DOI: 10.1145/3510466.3510485 (cited on pp. 14, 103).

[TI16] *MSP430FR5994 Launchpad Development Kit (MSP-EXP430-FR5994)*. SLAU678A. Texas Instruments. Apr. 2016. URL: https://ti.com/lit/ug/slau678b/slau678b.pdf (cited on p. 81).

[TOV20] Charles Truong, Laurent Oudre, and Nicolas Vayatis. "Selective review of offline change point detection methods". In: *Signal Processing* 167 (Feb. 2020). ISSN: 0165-1684. DOI: 10.1016/j.sigpro.2019.107299 (cited on p. 86).

[VK02] Arie Van Deursen and Paul Klint. "Domain-specific language design requires feature descriptions". In: *Journal of Computing and Information Technology* 10.1 (2002), pp. 1–17. ISSN: 1846-3908. DOI: 10.2498/cit.2002.01.01 (cited on p. 16).

[VS08] Aneta Vulgarakis and Cristina Seceleanu. "Embedded Systems Resources: Views on Modeling and Analysis". In: *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference*. COMPSAC '08. Turku, Finland: IEEE, July 2008, pp. 1321–1328. DOI: 10.1109/COMPSAC.2008.215 (cited on pp. 4, 56).

[WW97] Yong Wang and Ian H. Witten. "Inducing model trees for continuous classes". In: *Proceedings of the Poster Papers of the 9th European Conference on Machine Learning*. ECML '97. Prague, Czech Republic, Apr. 1997, pp. 128–137 (cited on p. 34).

[Yan+19] Chengrun Yang et al. "OBOE: Collaborative Filtering for AutoML Model Selection". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '19. Anchorage, AK, USA: Association for Computing Machinery, July 2019, pp. 1173–1183. ISBN: 978-1-4503-6201-6. DOI: 10.1145/3292500.3330909 (cited on p. 159).

[YF09] Kenji R Yamamoto and Paul G Flikkema. "Prospector: Multiscale Energy Measurement of Networked Embedded Systems with Wideband Power Signals". In: *Proceedings of the International Conference on Computational Science and Engineering*. CSE '09. Vancouver, BC, Canada: IEEE, Aug. 2009, pp. 543–549. DOI: 10.1109/CSE.2009.413 (cited on pp. 79, 192).

[Yu+21] Geoffrey X. Yu et al. "Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training". In: *Proceedings of the USENIX Annual Technical Conference*. USENIX ATC '21. Virtual Event: USENIX Association, July 2021, pp. 503–521. ISBN: 978-1-939133-23-6 (cited on pp. 149, 158).

[Zha+10] Lide Zhang et al. "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones". In: *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES/ISSS '10. Scottsdale, AZ, USA: Association for Computing Machinery, Oct. 2010, pp. 105–114. ISBN: 978-1-6055-8905-3. DOI: 10.1145/1878961.1878982 (cited on pp. 52, 72, 73, 114).

[Zha+15] Yi Zhang et al. "Performance Prediction of Configurable Software Systems by Fourier Learning". In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. ASE '15. Lincoln, NE, USA: IEEE, Nov. 2015, pp. 365–373. DOI: 10.1109/ASE.2015.15 (cited on pp. 46, 139, 153, 180).

[Zha+18] Cheng Zhang et al. "Performance Evaluation of Candidate Protocol Stack for Service-Based Interfaces in 5G Core Network". In: *Proceedings of the IEEE International Conference on Communications Workshops*. ICC '18 Workshops. Kansas City, MO, USA: IEEE, May 2018. DOI: 10.1109/ICCW.2018.8403675 (cited on p. 179).

[Zha+21] Li Lyna Zhang et al. "nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices". In: *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '21. Virtual Event: Association for Computing Machinery, June 2021, pp. 81–93. ISBN: 978-1-4503-8443-8. DOI: 10.1145/3458864.3467882 (cited on pp. 145, 159).

[Zha+23] Rui Zhang et al. "Optimal Sparse Regression Trees". In: *Proceedings of the 37th AAAI Conference on Artificial Intelligence*. Washington, DC, USA: AAAI Press, Feb. 2023, pp. 11270–11279. DOI: 10.1609/aaai.v37i9.26334 (cited on pp. 143, 144, 197).

[Zho+11] Hai-Ying Zhou et al. "Modeling of Node Energy Consumption for Wireless Sensor Networks". In: *Wireless Sensor Networks* 3.1 (Jan. 2011), pp. 18–23. DOI: 10.4236/wsn.2011.31003 (cited on p. 53).

[ZO13] Nanhao Zhu and Ian O'Connor. "Energy Measurements and Evaluations on High Data Rate and Ultra Low Power WSN Node". In: *Proceedings of the 10th International Conference on Networking, Sensing and Control*. ICNSC '13. Evry, France: IEEE, Apr. 2013, pp. 232–236. DOI: 10.1109/ICNSC.2013.6548742 (cited on pp. 4, 54).

[ZV16] Nanhao Zhu and Athanasios V. Vasilakos. "A generic framework for energy evaluation on wireless sensor networks". In: *Wireless Networks* 22.4 (May 2016), pp. 1199–1220. ISSN: 1572–8196. DOI: 10.1007/s11276-015-1033-x (cited on pp. 49, 53, 73).

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AI	Artificial Intelligence
CART	Classification and Regression Trees
CPS	Cyber-Physical Systems
DCO	Digitally Controlled Oscillator
DECART	Data-Efficient Classification and Regression Trees
DFA	Deterministic Finite Automaton
DUT	Device Under Test
FW	Feature-Wise Annotation
GPIO	General Purpose Input/Output
HFXT	High-Frequency External Crystal Oscillator
IoT	Internet of Things
LED	Light-Emitting Diode
LMT	Linear Model Trees
LUT	Lookup Table
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
MCU	Microcontroller Unit
MSE	Mean Square Error
NBCART	Non-Binary Classification and Regression Trees
NFP	Non-Functional Property
NN	Neural Network
OS	Operating System

PFA	Parameterized Finite Automaton
PPTA	Parameterized Priced Timed Automaton
RMT	Regression Model Trees
RPC	Remote Procedure Calls
SDR	Standard Deviation Reduction
SMAPE	Symmetric Mean Absolute Percentage Error
SMU	Source/Measure Unit
SoC	System on Chip
SPL	Software Product Line
SPLE	Software Product Line Engineering
SSR	Sum of Squared Residuals
TPU	Tensor Processing Unit
UART	Universal Asynchronous Receiver/Transmitter
ULS	Unsupervised Least-Squares Regression
XGB	Extreme Gradient Boosting