Performance is not Boolean: Supporting Scalar Configuration Variables in NFP Models

Birte Friesel
birte.friesel@uos.de
Universität Osnabrück
Osnabrück, Germany

Olaf Spinczyk
olaf@uos.de
Universität Osnabrück
Osnabrück, Germany

ABSTRACT

Non-functional properties (NFPs) such as memory requirements, timing, or energy consumption are important characteristics of embedded software systems and software product lines (SPLs) in general. Both during system design and at runtime, the goal is to optimize resource utilization (and, thus, NFPs) by appropriate system configuration or orchestration. NFP models, learned from benchmarks of various SPL configurations, allow for the prediction of these properties, thus enabling NFP-aware software configuration and runtime decisions. However, many existing approaches for automated learning of NFP models limit their scope to boolean variables. We argue that this is no longer sufficient: NFP models must accommodate scalar variables to achieve suitable accuracy when faced with today's highly configurable software systems and variable workloads. To this end, we evaluate four regression tree-based NFP modeling approaches on eight use cases, and examine model complexity and model accuracy. We find that models with support for scalar variables achieve up to three times lower mean model error when predicting configurations that were not part of the training set. At the same time, the complexity of scalar and boolean-only models is nearly the same; only benchmarking becomes more time-intensive due to the need to explore scalar variables. We conclude that scalar-enabled models provide increased accuracy almost free of charge, and recommend using them when generating NFP models for embedded systems and workloads with scalar configuration variables.

KEYWORDS

Performance Prediction, Regression Model Trees, Software Product Lines, Non-Functional Properties

1 INTRODUCTION

Models for non-functional system properties (NFP models for short) have proven to be useful for system design in a wide area of use cases. Given a system (or product-line) configuration, they provide estimates for attributes such as ROM/RAM utilization, data processing throughput, or energy requirements. Having NFP models at hand allows system designers to estimate system properties before building or using it, compare different solutions, and even automatically configure system components to optimize relevant NFPs [9, 15]. Similarly, operating systems for heterogeneous many-core environments can use NFP-aware scheduling algorithms to automatically execute tasks on appropriate hardware [5].

However, NFP models do not appear out of thin air: generating them requires a formal variability model, a translation of configuration variables to feature vectors $\vec{x}$ that are used as model input, and benchmarks of various system configurations that can be used to learn the model. In the world of software product lines (SPLs), feature vectors are still often considered to be boolean [10], resulting in an NFP model $f : \{0, 1\}^n \rightarrow \mathbb{R}$. While this allows for simple, efficient sampling methods [4, 6], it disregards the influence of scalar variables. Although approaches with support for scalar variables exist [13], to our knowledge, all of them require scalar variables to be well-defined ($x_i \in \mathbb{R}$ for all variables x_i in all measurements). In practice, however, scalar variables may be undefined in some configurations. This can happen whenever a scalar variable configures details of an optional feature: if the feature is disabled, there is nothing to configure, so the scalar variable has no value ($x_i = \perp$).

Considering this, and aspects such as the impact of buffer sizes on RAM usage, or the number of available cores on parallel workload performance, we argue that accurate NFP models need to handle (possibly undefined) scalar variables, resulting in an NFP model $f : (\mathbb{R} \cup \{\perp\})^n \rightarrow \mathbb{R}$. To assess this, we generate and evaluate eight NFP models with and without scalar variable support for four different applications: ROM/RAM usage of the embedded research operating systems *Kratos* and *Multipass*, ROM/RAM usage of the *busybox* multi-call binary, and encoding duration as well as output file size of the *x264* video codec. All of these are highly configurable.

We examine NFP models using classification and regression trees (CART) with and without scalar variable support, linear model trees (LMT) with scalar variable support, and a

FGBS '22, March 17–18, 2022, Hamburg, Germany

https://doi.org/10.18420/fgbs2022f-03

Table 1: Number of benchmark samples and boolean/integer variables in feature vectors of evaluated applications.

Application	# Samples	# Boolean	# Scalar
Busybox	2,000	1,001	17
Kratos	30,085	70	3
Multipass	16,761	125	4
x264	3,646	9	4

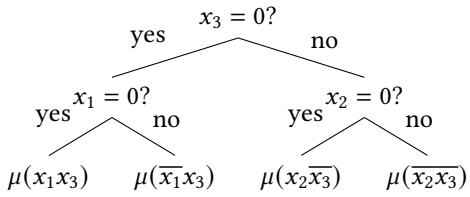

Figure 1: A regression tree. Each leaf holds a constant model μ for the corresponding partial system configuration.

custom CART extension, regression model trees (RMT). We contribute a definition of RMT, and a comparison of boolean-only and scalar-aware model accuracy and complexity for the aforementioned use cases. Using these observations, we show that NFP models with support for scalar variables achieve up to three times lower model error when predicting configurations that were not part of the training set than boolean-only models, with nearly identical model complexity.

In the next section, we give a brief introduction to the four applications and their boolean and scalar configuration variables. We then introduce CART, LMT, and RMT in section 3, and provide evaluation results in section 4. After presenting related work in section 5, we conclude in section 6.

2 FEATURE EXTRACTION

All four applications use the Kconfig language for configuration and variability modeling, and can therefore be treated as software product lines [14]. Busybox, Kratos, and Multipass are instances of embedded software systems, whereas x264 video encoding is a parallel computing workload.

The Kconfig language offers five types of variables: bool, tristate, string, hex, and int. Each variable may have dependencies on other variables; if a variable's dependencies are not met, it is not visible and cannot be configured. Tristate variables are used in the Linux kernel to distinguish between disabled, enabled, and module features. Our product lines do not use them, and we therefore do not consider them. We also ignore string variables.

For boolean-only NFP models, we simply build a feature vector $\vec{x} \in \{0,1\}^n$ out of the n bool variables present in the configuration. If a variable x_i is selected, we set $x_i = 1$; if it is deselected or invisible, we set $x_i = 0$. We ignore scalar variables.

For scalar-aware NFP models, we build a feature vector $\vec{x} \in (\mathbb{R} \cup \{\bot\})^n$ out of the n bool, hex, and int variables present in the configuration. We handle boolean variables as above, and set hex and int variables to their respective scalar configuration value. If a scalar variable x_i is invisible, it does not have a well-defined value, so we set $x_i = \bot$.

All applications we use for evaluation purposes have more boolean than scalar variables, though the ratio differs. Table 1 shows the number of benchmark samples and feature vector components.

3 REGRESSION TREES

We first introduce the common regression tree data structure, and then present the (DE)CART, LMT, and RMT generation methods used in this paper. All of them use a set of benchmark observations $S = \{y_1, y_2, \dots\}$ and corresponding feature vectors $\vec{x}_1, \vec{x}_2, \dots$ to learn a model function $f : M \to \mathbb{R}$. When describing model generation, we use x_i to refer to the i-th feature vector element (i.e., the i-th boolean or scalar Kconfig variable).

3.1 Data Structure

Regression Trees (also known as Classification and Regression Trees, or CART) have been introduced in 1984 and continue to be relevant to this day [2]. Formally, they express a function $f : M \to \mathbb{R}$ by means of a binary decision tree, for an arbitrary set M.

Each non-leaf node holds a binary decision, and each leaf defines the function output for the partial configuration described by the path from the root to the leaf. The function result $f(\vec{x})$ is determined by following the decisions from the root until ending up in a leaf, and then returning the leaf value. Due to the binary decisions involved in each node, it is well-defined for any input $\vec{x} \in M$.

Fig. 1 shows an example CART for a binary variability model with $M = \{0,1\}^3$. In this case, leaf values $\mu(\cdots)$ are the arithmetic mean of the corresponding benchmark observations.

3.2 Boolean CART

CART are generated top-down by greedily selecting splits that minimize the *loss* (i.e., model error) until a stop criterion is reached or no further variables can be split on [2]. Each leaf holds a static value, resulting in a piecewise constant function $f : \{0,1\}^n \to \mathbb{R}$.

We now outline the generation algorithm for binary-only CART. As these allow for efficient sampling, they are also known as data-efficient CART, or *DECART* [4].

(1) If a stop criterion is satisfied: return a leaf node using the mean of observed data $\mu(S)$ as model value.
(2) Split S into partitions S_i (with $x_i = 1$) and $\overline{S_i}$ (with $x_i = 0$) for each variable index i.
(3) Associate each partition with a static model using the arithmetic mean of corresponding observations: $\mu(x_i) = \mu(S_i)$ and $\mu(\overline{x_i}) = \mu(\overline{S_i})$.
(4) Select the variable x_i with the lowest loss (i.e., model error) and transform it into a decision tree node.
(5) Repeat recursively with S_i and $\overline{S_i}$ to generate child trees.

Stop criteria may be user-specified thresholds for the number of samples $|S|$ or sample standard deviation $\sigma(S)$, or tree depth and size limits. These are meant to reduce the risk of overfitting. A common loss function is the sum of squared residuals:

$$\sum_{y \in S_i} (y - \mu(x_i))^2 + \sum_{y \in \overline{S_i}} (y - \mu(\overline{x_i}))^2$$

3.3 Scalar CART

DECART are a special case of (scalar) *CART*, and scalar CART generation works in a similar manner [2].

(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 variable index i, let $T_i = \{t_{i,1}, t_{i,2}, \dots\}$ be the ordered set of its unique values.
(3) Split S into partitions $S_{i,j,\text{left}}$ (with $x_i \leq t_{i,j}$) and $S_{i,j,\text{right}}$ (with $x_i > t_{i,j}$) for each pair $(x_i, t_{i,j})$.
(4) Associate each partition with a static model, as above.
(5) Select the pair $(x_i, t_{i,j})$ with the lowest loss and transform it into a decision tree node "$x_i \leq \frac{t_{i,j}+t_{i,j+1}}{2}$?"
(6) Repeat recursively with $S_{i,j,\text{left}}$ and $S_{i,j,\text{right}}$.

Loss function and stop criteria remain unchanged. The result is a piecewise constant function $f : \mathbb{R}^n \to \mathbb{R}$.

In our case, the input set is $(\mathbb{R} \cup \{\bot\})^n$. Therefore, in steps 2–5, we only consider variables x_i with $\bot \notin T_i$. As $\{0, 1\} \subset \mathbb{R}$, the algorithm can handle both boolean and scalar variables.

3.4 Linear Model Trees

Linear Model Trees (LMT) are an extension of CART. They also rely on decision trees, but use both static values and linear functions in leaves [11]. This allows them to express piecewise linear functions and capture the influence of scalar configuration variables.

For instance, the model tree in Fig. 2 describes the output file size after encoding a specific input file with x264. If

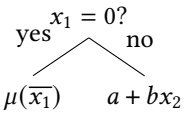

Figure 2: A linear model tree. Each leaf holds a constant model μ or a linear model $a + bx_i + cx_j + \cdots$.

constant bitrate encoding ($x_1 \in \{0, 1\}$) is enabled, it depends on the configured target bitrate ($x_2 \in \mathbb{R}$). Otherwise, in this example, it is constant. We note that $x_1 = 0$ implies $x_2 = \bot$, as variable bitrate encoding does not use a target bitrate.

Model tree generation works by building a CART and using a bottom-up *pruning algorithm* combined with linear regression analysis to transform sub-trees into linear functions. Details of the pruning algorithm vary between publications; we refer to the M5' algorithm for details and an example [16].

3.5 Regression Model Trees

Our custom *Regression Model Tree* (RMT) approach is an extension of CART as well, but not limited to linear functions. Instead, we first generate a decision tree according to the DECART algorithm (only considering boolean configuration variables in tree nodes), and then use a *fitted function set* algorithm to find and fit functions for each leaf node. The result is a decision tree expressing a piecewise continuous function.

The fitted function set algorithm automatically generates functions to express complex linear and non-linear effects of scalar configuration variables. It uses statistical analysis to determine relevant configuration variables, and multi-step regression analysis to find and fit an appropriate function. We originally developed it for energy model generation, and found that it also works well when faced with software-centric NFP models. We refer to our previous work for details [3].

4 EVALUATION

We examine model accuracy and complexity by generating DECART, CART, LMT, and RMT models for all eight use cases. To this end, we first generate a set of observations S for each application by means of random sampling with neighbourhood exploration, and then pass it to the respective model generation algorithms.

Our RMT implementation builds upon the DECART algorithm from the literature [4]. It handles undefined variables as described in section 3.3. For LMT generation, we use an external open-source implementation[1] whose maximum tree

[1] https://github.com/cerlymarco/linear-tree

Table 2: Symmetric Mean Absolute Percentage Error (SMAPE) and tree depth of static, regression tree, linear model tree, and regression model tree-based NFP models with 10-fold cross validation.

	SMAPE [%]					Tree depth			
Application	Static	DECART	CART	LMT	RMT	DECART	CART	LMT	RMT
Busybox ROM	92.6	0.3	0.3	–	0.3	158	162	–	163
Busybox RAM	142.4	0.2	0.2	–	0.3	159	163	–	164
kratos ROM	41.9	0.5	0.5	0.5	0.5	36	36	5	36
kratos RAM	83.4	0.8	0.8	0.8	0.8	34	34	5	34
multipass ROM	76.8	6.5	6.6	12.9	2.0	44	44	20	38
multipass RAM	57.1	3.7	3.7	19.8	2.6	37	37	20	28
x264 time	48.0	45.5	22.6	21.1	16.9	7	18	20	10
x264 size	105.7	97.9	87.9	92.1	57.9	7	17	20	10

depth is hardcoded to 20. We use the implementation provided in the Python3 scikit-learn package for CART and DECART. Both do not support undefined variables, so we leave them out during model generation. Apart from the hard-coded LMT limit, we do not configure stop criteria.

4.1 Sampling

We use `kconfig-conf --randconfig` to generate random configurations, and measure their NFP values. For each random configuration, we additionally toggle each individual (visible) boolean configuration variable, and take five equidistant samples from the allowed range of each linear configuration variable. For each of these samples, whose configuration differs from the original (randomly generated one) in precisely one variable, we also run a benchmark. With n_b boolean and n_i scalar configuration variables, this results in up to $n_b + 5n_i$ measurements per random configuration, depending on constraints imposed by Kconfig dependencies.

For busybox, we further decrease the sample size by taking 2,000 random samples from the set of observations generated using random sampling with neighbourhood exploration. In all other cases, we pass the entire set of observations to the model generation algorithms. We deliberately do not attempt to perform data-efficient sampling, as we want to focus on model accuracy alone.

4.2 Results

We assess model accuracy by performing configuration-aware 10-fold cross validation: we partition observations into ten pairs of (mutually exclusive) training and validation sets based on their feature vector $\vec{x}$. So, $\vec{x}_t \neq \vec{x}_v$ for any pair of training sample (with configuration $\vec{x}_t$) and validation sample (with configuration $\vec{x}_v$). This ensures that the validation set only contains system configurations that were not part of the training set.

We calculate the symmetric mean absolute percentage error (SMAPE) for each regression tree model as well as for a *static* model that simply uses $\mu(S)$ for prediction. Given predictions $P = \{p_1, \ldots, p_n\}$ and ground truth $Y = \{y_1, \ldots, y_n\}$, SMAPE is defined as follows.

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

Table 2 shows the mean model error with cross validation. We see that the prediction error of regression model trees is up to three times lower than the boolean-only DECART variant for two of four applications, and nearly identical for busybox and kratos. CART are better than DECART in one application. In case of busybox, the LMT implementation we used was unable to generate a regression tree; for multipass, it performs worse than both scalar and boolean-only models. So, RMT generally outperform their boolean-only counterpart if scalar variables play a large role (multipass, x264), and do not impair model performance when scalar variables are not important (busybox, kratos). They achieve the lowest model error, followed by CART. On the model complexity side, we see that linear model trees are by far the smallest, but also often the least accurate NFP model. CART and RMT, on the other hand, are only slightly more complex than the boolean-only DECART model. In case of multipass, RMT are both more accurate and more compact. As such, we conclude that the increase in accuracy gained by respecting scalar variables does not lead to significantly more complex models.

5 RELATED WORK

Researchers in the field of energy modeling have been considering scalar configuration variables for a long time, and frequently show their importance [1, 17]. When it comes to software-centric NFP models, this appears to be a rare case [10].

The most prominent example we are aware of is the work by Siegmund et al., presenting an approach for efficient sampling and accurate model generation with boolean and scalar variables [13]. However, their model relies on linear regression, which – to the best of our knowledge – is incapable of handling variables that may be undefined.

FLASH, on the other hand, combines data acquisition with multi-objective optimization [8]. It uses CART with boolean and scalar variables internally, but focuses on optimization, not NFP model generation: optimization goals must be known beforehand.

On the regression tree side, linear model trees are known to be a suitable modeling method for boolean and scalar inputs [7]. A recent software fault prediction study uses them to achieve 5 to 50 % model error [12]. This is similar to our results.

Fourier Learning is an entirely different approach [18]. In contrast to previously mentioned works, it provides guaranteed accuracy bounds, but requires a large amount of samples. It is limited to boolean feature variables.

6 CONCLUSION

In an evaluation of four different applications, we have shown that scalar configuration variables can be an important aspect when modeling non-functional properties of embedded software systems and compute workloads, even if most variables are boolean. RMT and CART models that respect scalar variables achieve an up to three times lower model error than regression trees without support for scalar variables and show negligible additional complexity. Our own regression tree variant, regression model trees (RMT), consistently achieves the lowest model error.

Regression model trees are able to handle partially undefined scalar variables with ease. Even if all scalar variable are undefined in at least one benchmark, each split during tree generation decreases the sample size. In our experience, this, combined with the dependencies between boolean variables and scalar variables in Kconfig files, means that lower tree levels are not impacted by undefined variables. Typically, once a few boolean variables have been handled, each scalar variable in the remaining sample set is either defined in all measurements (and therefore usable) or undefined in all measurements (and therefore irrelevant). As such, all scalar variables can contribute to model accuracy.

Although sampling with support for scalar variables is more time-consuming than benchmark generation for boolean-only configuration spaces, we consider the gain in model accuracy to be worth the effort. Considering the existing approaches to improve sampling in both cases, we are hopeful that data-efficient sampling and accurate, scalar-aware NFP models are not mutually exclusive.

REFERENCES

[1] Gautier Berthou, Kevin Marquet, Tanguy Risset, and Guillaume Salagnac. 2020. Accurate Power Consumption Evaluation for Peripherals in Ultra Low-Power embedded systems. In *2020 Global Internet of Things Summit (GIoTS)*. 1–6. https://doi.org/10.1109/GIOTS49054.2020.9119593

[2] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. 1984. *Classification and regression trees*. Routledge. https://doi.org/10.1201/9781315139470

[3] Birte Friesel, Markus Buschhoff, and Olaf Spinczyk. 2018. Parameter-Aware Energy Models for Embedded-System Peripherals. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*. 1–4. https://doi.org/10.1109/SIES.2018.8442096

[4] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-Efficient Performance Learning for Configurable Systems. *Empirical Softw. Engg.* 23, 3 (June 2018), 1826–1867. https://doi.org/10.1007/s10664-017-9573-6

[5] Simon Holmbacka and Jörg Keller. 2017. Workload Type-Aware Scheduling on big.LITTLE Platforms. In *Algorithms and Architectures for Parallel Processing*, Shadi Ibrahim, Kim-Kwang Raymond Choo, Zheng Yan, and Witold Pedrycz (Eds.). Springer International Publishing, Cham, 3–17.

[6] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to Sample: Exploiting Similarities across Environments to Learn Performance Models for Configurable Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3236024.3236074

[7] D. Malerba, F. Esposito, M. Ceci, and A. Appice. 2004. Top-down induction of model trees with regression and splitting nodes. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 5 (2004), 612–625. https://doi.org/10.1109/TPAMI.2004.1273937

[8] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2020. Finding Faster Configurations Using FLASH. *IEEE Transactions on Software Engineering* 46, 7 (2020), 794–811. https://doi.org/10.1109/TSE.2018.2870895

[9] Rafael Olaechea, Steven Stewart, Krzysztof Czarnecki, and Derek Rayside. 2012. Modelling and Multi-Objective Optimization of Quality Attributes in Variability-Rich Software. In *Proceedings of the Fourth International Workshop on Nonfunctional System Properties in Domain Specific Modeling Languages* (Innsbruck, Austria) *(NFPinDSML '12)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/2420942.2420944

[10] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2021. Learning software configuration spaces: A systematic literature review. *Journal of Systems and Software* 182 (2021), 111044. https://doi.org/10.1016/j.jss.2021.111044

[11] John R Quinlan et al. 1992. Learning with continuous classes. In *5th Australian joint conference on artificial intelligence*, Vol. 92. World Scientific, 343–348.

[12] Santosh Singh Rathore and Sandeep Kumar. 2016. A decision tree regression based approach for the number of software faults prediction. *ACM SIGSOFT Software Engineering Notes* 41, 1 (2016), 1–6.

[13] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association

for Computing Machinery, New York, NY, USA, 284–294. https://doi.org/10.1145/2786805.2786845

[14] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. 2007. Is The Linux Kernel a Software Product Line?. In *Open Source Software and Product Lines (SPLC-OSSPL 2007)*, Frank van der Linden and Björn Lundell (Eds.). Kyoto, Japan.

[15] Julio Sincero, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. 2010. Approaching Non-functional Properties of Software Product Lines: Learning from Products. In *2010 Asia Pacific Software Engineering Conference*. 147–155. https://doi.org/10.1109/APSEC.2010.26

[16] Yong Wang and Ian H Witten. 1996. Induction of model trees for predicting continuous classes. (1996).

[17] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang. 2010. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 105–114.

[18] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance Prediction of Configurable Software Systems by Fourier Learning (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 365–373. https://doi.org/10.1109/ASE.2015.15