Understanding Product Line Runtime Performance with Behaviour Models and Regression Model Trees

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

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

Abstract

Performance models for software product lines encode the relation between configurable product features and performance attributes. These cover both static attributes such as binary size, and runtime-specific attributes such as throughput or latency. However, although runtime-specific attributes depend on workload and application behaviour, existing approaches typically only predict a single performance value for each attribute and product line configuration. They utilize fixed reference workloads for model learning and performance prediction, and are therefore inadequate for predicting feature-dependent runtime performance attributes of variable workloads (e.g., variable query sequences in a database management system). Moreover, viewing a product line as a black box that is only described by its feature model hinders efficient benchmark data acquisition and reasoning about unexpected performance effects. We propose to make performance models aware of software product line implementations, thus improving flexibility, interpretability, and learning time. To do so, we decompose runtime behaviour into distinct events (state machine transitions), and annotate each event with an event-specific performance model (regression model tree). The combination of state machine-based behaviour models with regression model trees allows us to predict performance attributes of arbitrary event sequences (words accepted by the state machine). In addition to more flexible models that are no longer bound to a specific workload, this approach improves model interpretability and learning time by using smaller models for individual workload components. We show the advantages of this approach in a database management system product line case study, and use it to explain unexpected behaviour on a real-world server system.

CCS Concepts

• **Software and its engineering → Software performance**; *Operational analysis*.

Keywords

Runtime Performance, Performance Models, Behaviour Models, Non-Functional Properties, Dynamic Software Product Lines

ACM Reference Format:
Birte Friesel and Olaf Spinczyk. 2025. Understanding Product Line Runtime Performance with Behaviour Models and Regression Model Trees. In *29th ACM International Systems and Software Product Line Conference - Volume A (SPLC-A '25), September 1–5, 2025, A Coruña, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3744915.3748472

ACM ISBN 979-8-4007-2024-6/2025/09
https://doi.org/10.1145/3744915.3748472

1 Introduction

The past two decades have seen a growing interest in performance attributes (also known as non-functional properties) of software product lines (SPLs), and performance models that are capable of predicting them for arbitrary product line configurations [1, 3, 9, 37, 41, 44, 46]. Performance models allow SPL users to reason about performance attributes without having to run configuration-specific benchmarks, and to assess how individual product line features affect product performance [2, 22, 35].

So far, the focus has been on building accurate performance models from benchmark data, preferably without resorting to an exhaustive configuration space exploration [38]. However, there is little research on understanding *why* certain features have a specific performance influence. Performance modelling treats product line runtime behaviour as a black box whose performance attributes are only known thanks to benchmark results, and furthermore utilizes fixed reference workloads for model learning. So, SPL engineers and users can neither explain which feature-dependent runtime components of a product line are causing certain performance effects, nor how SPL performance would fare with different workloads (e.g., analytical rather than transactional database queries).

Consider a database management system (DBMS) product line. In addition to typical DBMS tunables, it provides optional support for offloading query kernels to processing-in-memory (PIM) modules: DRAM modules with built-in data processing units (DPUs) that can process data independent of the CPU. Such modules are commercially available from UPMEM [27], and have sparked considerable interest within the database community [5–7, 30].

DBMS features include PIM support and the default number of CPU cores and PIM memory modules (ranks) used for query processing, and a performance model predicts how DBMS configuration influences the latency of a reference benchmark. The model relies on classification and regression trees (CART) [11], which are commonly used in the product line engineering community [38].

Now, the engineers observe an unexpected relation between features and predicted system performance. When using PIM and varying the number of ranks allocated to running their reference queries, they find that the optimal number of ranks is neither the minimum nor the maximum that is available in the system. Moreover, the sweet spot depends on the column size used for the benchmark runs from which the performance model is generated, and updating the UPMEM SDK (i.e., the software development kit for using these PIM modules) also alters it. For instance, as Fig. 1 shows, the sweet spot is 28 ranks for 2^{30} rows, and 20 to 28 ranks for 2^{28}

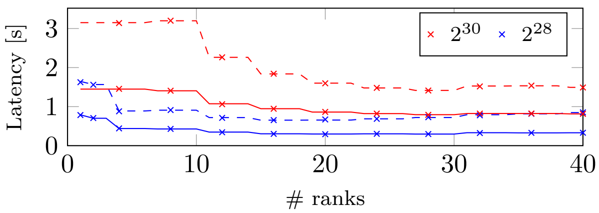

Figure 1: DBMS reference benchmark latency with variable number of PIM modules (X axis) and benchmark column size (coloured lines). Lines are CART performance model predictions for UPMEM SDK 2023.2.0 (dashed) and 2025.1.0 (solid), crosses refer to benchmark output (ground truth).

rows. Meanwhile, CPU execution behaves as expected: additional cores decrease latency until they hit the memory wall [33].

The CART model used for performance prediction is not at fault here: underlying benchmark data (crosses in Fig. 1) confirms both observations. However, the model is unsuitable for understanding why the DBMS is behaving this way. On the one hand, with 4,391 tree nodes, it is too complex for manual interpretation. On the other hand, the model treats the DBMS as a black box (i.e., it only assigns a total latency to an entire benchmark run), and thus lacks fine-grained performance insights.

This paper proposes a white-box performance modelling approach that builds upon state machine-based behaviour models and regression model trees. It decomposes system behaviour into sequences of events, associates each event with individual performance models, and also takes runtime- and benchmark-specific attributes such as the column size used for performance model generation into account. The resulting performance prediction models for runtime attributes of software product lines are simple and accurate, thus helping SPL engineers and users understand and explain effects such as the one described in the previous paragraphs.

The next section introduces the concepts we propose for understanding feature- and workload-dependent runtime performance attributes of configurable software systems, using the DBMS product line as a running example to illustrate their use. Afterwards, Section 3 compares the training overhead, accuracy, complexity, and flexibility of performance-aware behaviour models with conventional performance modelling approaches, including models that are aware of workload-specific runtime attributes. Section 4 examines related work, and Section 5 concludes this paper.

2 Contribution

We combine three components: runtime variability, behaviour models, and regression model trees.

2.1 Runtime Variability

Prior research on performance models for software product lines has focused on the relation between configurable features and performance attributes. For instance, the x264 video codec and the influence of its flags (e.g., target bitrate or psycho-visual optimizations) on attributes such as encoding latency is a common evaluation target for performance model learning algorithms [38, 45]. However, encoding latency also depends on the input file: longer videos or higher resolutions will likely lead to increased encoding latencies. Yet, publications that utilize x264 encoding latency as an evaluation target for performance modelling methods do not take variable input files into account, instead using a fixed reference workload (i.e., a single file) for learning and evaluation [18, 26, 43, 45, 48].

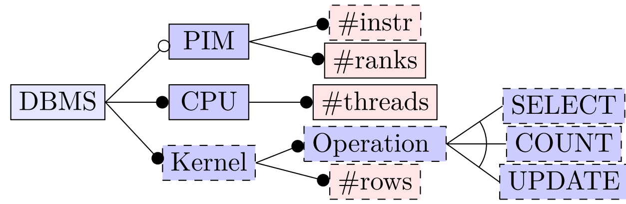

Figure 2: Variability model for a DBMS product line, including runtime-only variability (dashed boxes).

The same applies to our DBMS example: throughput and latency do not just depend on compile-time features, but also on runtime attributes such as the size of the columns that individual database operations access. Although column size is not a feature in the sense of product line engineering, it is relevant for performance prediction – otherwise, performance models can only predict performance attributes for whichever fixed column size has been used in the benchmark that they were trained on. Similarly, compile-time features may be changed at runtime. This is prevalent in, e.g., SQLite, where the majority of features encode default behaviour that can later be overridden for individual database connections. Hence, performance models must be able to consider runtime variability in addition to conventional product line features.

We propose to address this by adding runtime-only components to the variability model. These describe runtime variability that is relevant for performance prediction despite not encoding features in the product line engineering sense. Essentially, this turns our DBMS into a dynamic software product line [28]: we do not just allow compile-time features to be changed at runtime, but also support runtime-only variability.

Fig. 2 shows the extended variability model for our running example, with runtime-only entries marked as dashed boxes. It has optional support for PIM, and allows users to configure a default number of PIM ranks (if PIM is enabled) as well as a default number of cores used for CPU execution. Its runtime-only variability consists of the number of instructions used by kernels offloaded to PIM modules, the operation that each database kernel executes, and the number of rows that it operates on.

Note that we leave out compile-time database features such as shared caches or query planner configuration in order to keep the example small enough to fit within this paper. All of our methods also apply to more complex product lines that include a wider range of compile- and runtime options, such as SQLite or Postgres.

2.2 Behaviour Models

Similar to the lack of runtime variability mentioned in the previous section, all past performance modelling research for software product lines that we are aware of has used fixed reference workloads (benchmarks) for model learning and evaluation. For instance, existing SQLite or Postgres latency or throughput prediction models were learnt and evaluated on a standard benchmark, and are thus only valid for this benchmark's workload (i.e., query sequences) [25, 34, 38, 42]. While there are transfer learning approaches that are capable of adjusting a performance model to hardware or workload changes [29], those still require new benchmarks whenever the workload changes, as the underlying performance model is unaware of workload-dependent performance attributes.

We propose to make the workload an explicit component of the performance model instead, thus allowing the model to predict runtime-specific performance attributes of arbitrary workloads. In order to do so, we decompose the runtime behaviour of the product line into individual steps, and define a state machine that expresses how those steps interact. Each state machine transition corresponds to a runtime step and is annotated with a set of performance models that predict runtime performance attributes of just this step, while taking feature configuration and runtime variability into account.

Our concept builds upon featured transition systems [4, 14, 15]. These provide an established method for modelling feature-dependent runtime behaviour of product lines by extending state machine transitions (runtime steps) with *feature guards.* A transition can only be taken if the current product line configuration satisfies the corresponding feature guard, which expresses feature constraints by means of a logic formula that references the variability model. In contrast to related works that annotate transitions with constant energy or latency values (thus transforming them into weighted featured transition systems or featured weighted automata) [10, 19, 36], we annotate them with arbitrary performance models such as CART or regression model trees.

In our DBMS example, runtime steps relate to individual database kernels. If PIM is disabled and a benchmark exercises a sequence of n SELECT and m UPDATE queries, the total latency is n times the latency of a single SELECT kernel plus m times the latency of a single UPDATE kernel. For PIM execution, the sequence of steps is more complex, and also depends on follow-up queries. The DBMS needs to allocate PIM ranks, write query-specific column data to them, upload a kernel binary, write query arguments such as column size or a WHERE clause, run the kernel, and read back its output. In case follow-up queries access the same column, the cost for rank allocation and data transfer has to be paid just once for each set of consecutive queries.

Fig. 3 shows a state machine that encodes this range of runtime behaviours, thus providing a behaviour model. For CPU execution (prefixed with [CPU]), runKernel is a single step that may be repeated (as described by the dotted ε-transition) in case the workload consists of multiple queries. For PIM execution (prefixed with [PIM]), a single query consists of the steps outlined previously. Follow-up queries may re-use existing data and kernel (ε-transition to d), upload a new kernel to work on existing data (ε-transition to c), or start anew (ε-transition to the initial state a).

The B and T annotations refer to performance models for throughput (B) and latency (T), normalized to Bytes per second and seconds, respectively. Now, we can determine the latency of arbitrary workloads by querying the associated performance models and calculating the sum of all latency predictions. For a throughput model B, given workload data size D, we calculate latency $T = \frac{D}{B}$.

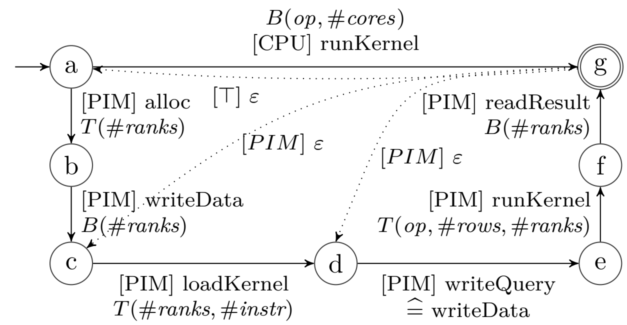

Figure 3: Annotated behaviour model for running an SQL query kernel on the CPU or on PIM modules.

We deliberately support throughput and latency models so that SPL engineers can utilize whichever is least complex and most accurate.

Given a behaviour model $\mathcal{A}$ and a variability model with a set of variables (features and runtime variability) V, we define a workload as a sequence $W = (\sigma_1, \vec{x}_1), \ldots, (\sigma_n, \vec{x}_n)$ with $\sigma_1 \cdots \sigma_n \in L(\mathcal{A})$. The mapping $\vec{x}_i : V \rightharpoonup \mathbb{R}$ encodes the feature and runtime configuration of each transition; numeric features with unsatisfied dependencies as well as unconfigured runtime variables are encoded as $\perp$. Thus, the total latency of a workload is $T(W) = \sum_{i=1}^{n} T_{\sigma_i}(\vec{x}_i)$.

For instance, the latency of running two consecutive SELECT operations on the same column, using four CPU cores and accessing 2^{30} rows (2^{33} Bytes) of data, is $2 \cdot \frac{2^{33}}{B_{\text{runKernel}}(\text{SELECT},4)}$. The latency of these operations on PIM, using 20 ranks, 538 kernel instructions, 64 Bytes of query arguments, and 2^{27} Bytes of results, is as follows.

$$T_{\text{alloc}}(20) + \frac{2^{33}}{B_{\text{writeData}}(20)} + T_{\text{loadKernel}}(20, 538)$$
$$+2 \cdot \left(\frac{64}{B_{\text{writeData}}(20)} + T_{\text{runKernel}}(\text{SELECT}, 2^{30}, 20) + \frac{2^{27}}{B_{\text{readResult}}(20)} \right)$$

2.3 Regression Model Trees

Behaviour models alone ore not sufficient for understanding product line performance: the individual CART models in this example are still complex, with 79 to 1,279 tree nodes.

This is due to a design limitation of CART and related methods: each decision node holds a boolean query (e.g., $\#ranks < 20$), and each leaf node holds a static weight. Thus, they are incapable of naturally expressing continuous (e.g., linear or inverse) relationships between runtime variables and performance attributes. While least-squares regression can express such natural relationships with ease, it relies on a suitable template for fitting, and suffers when dealing with interactions between boolean and numeric variables.

Regression model trees (RMT) extend regression trees by replacing static weights in leaf nodes with regression formulas, thus combining the benefits of decision trees (an easy-to-grasp tree structure for boolean variables) and least-squares regression (concise formulas for numeric variables) [24]. They do so by splitting variability into two parts: decision nodes exclusively reference boolean variables, and leaf nodes exclusively reference numeric variables.

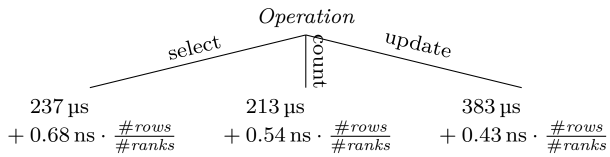

Figure 4: A regression model tree (RMT) for runKernel latency prediction when using PIM.

Fig. 4 shows an RMT for predicting runKernel latency on PIM. Since SELECT, COUNT and UPDATE are marked as alternatives in the variability model, they are mutually exclusive. The RMT learning algorithm is aware of this relationship and thus uses a single categorical variable $\mathit{Operation} \in \{\mathrm{SELECT}, \mathrm{COUNT}, \mathrm{UPDATE}\}$ rather than three variables $\mathit{SELECT}, \mathit{COUNT}, \mathit{UPDATE} \in \{0, 1\}$. This provides an additional benefit for interpretability by having the performance model structure closely resemble the variability model, despite having been learnt automatically.

The RMT learning algorithm builds the regression model tree in a recursive, top-down manner. First, it greedily adds decision nodes that partition the training data based on whichever non-numeric feature or runtime variable provides the greatest loss reduction (i.e., the highest model accuracy improvement). This handling of decision nodes is almost identical to the CART learning algorithm [11], except that RMT only consider boolean and categorical variables (see above) for decision nodes. Once additional splits no longer provide adequate loss reduction, the RMT learning algorithm adds a leaf node. Here, it utilizes unsupervised least-squares regression (ULS) to automatically find and fit a suitable least-squares regression formula [21], ignoring any non-numeric variables. This formula expresses how numeric variables affect product line performance in the partial configuration that is defined by the path from the root to the leaf. For instance, the rightmost leaf formula in Fig. 4 is only valid for workload steps with $\mathit{Operation} = \mathrm{UPDATE}$.

ULS uses a domain-specific set G of regression templates. Here, we are only interested in linear and roofline functions, and set $G = \{x \mapsto \beta_1 \cdot x, x \mapsto \beta_1 \cdot \min(x, \beta_2)\}$. For each variable i, the algorithm first determines whether it affects product line performance in the partial configuration corresponding to the leaf node that is currently being processed. If that is the case, it looks for the function template $g_i \in G$ that is best suited for predicting how x_i affects product line performance. Finally, it combines all functions $g_{i_1}, g_{i_2}, \ldots$ into a single function f, and annotates the leaf with it [21]. As the RMT learning algorithm has already taken care of all boolean variables, ULS exclusively deals with numeric variability.

3 Evaluation

We will now examine how runtime variability, behaviour models, and regression model trees allow SPL engineers and users to understand product line performance (qualitative evaluation), and compare their accuracy, complexity, and data acquisition overhead to behaviour models with CART as well as CART without behaviour models (quantitative evaluation). Artefacts are available at https://ess.cs.uos.de/git/artifacts/splc25-behaviour-models and archived at https://zenodo.org/records/15827230 [20].

We use a systematic configuration space exploration during model learning to ensure that inappropriate sampling does not interfere with our evaluation. For the behaviour model's RMT and CART models, we benchmark 1 to 40 ranks (40 measurements each) for alloc, writeData and readResult, and 1 to 40 ranks times 22 to 3,862 instructions for loadKernel ($40 \cdot 16$ measurements). Each database kernel is benchmarked on 2^{20} to 2^{32} database rows times 1 to 40 ranks (steps of 1/2/4), giving $2 \cdot 13 \cdot 13$ measurements. For CART without behaviour models, we benchmark the product of two operations, 2^{20} to 2^{32} database rows, 1 to 40 ranks, and 1 to 100 consecutive operations. Benchmarks with 1 to 20 operations (7 in total) exclusively use COUNT or SELECT operations; benchmarks with 100 operations mix COUNT, SELECT, and UPDATE.

3.1 Qualitative Evaluation

We are still on a quest to learn about the unexpected latency behaviour of database kernels on UPMEM PIM shown in Fig. 1. We will start with performance on SDK 2023.2.0, and then examine the changes brought by SDK 2025.1.0. Fig. 4 shows the model for runKernel; the others are as follows.

$$
\begin{aligned}
T_{\text{alloc}} &= 13.6\,\text{ms} + 12.4\,\text{ms} \cdot \#\mathit{ranks} \\
T_{\text{loadKernel}} &= 460\,\mu\text{s} + 2.9\,\mu\text{s} \cdot \#\mathit{instr} + 16.7\,\mu\text{s} \cdot \#\text{ranks} \\
B_{\text{writeData}} &= 1423\,\tfrac{\text{MB}}{\text{s}} + 254\,\tfrac{\text{MB}}{s} \cdot \min(\#\mathit{ranks}, 28.3) \\
B_{\text{readResults}} &= 2278\,\tfrac{\text{MB}}{\text{s}} + 174\,\tfrac{\text{MB}}{s} \cdot \min(\#\mathit{ranks}, 28.7)
\end{aligned}
$$

So, $\#\mathit{ranks}$ is both beneficial and detrimental: while runKernel latency is inversely proportional to the number of ranks, alloc latency is proportional to it. Moreover, additional ranks have little influence on data transfer throughput, and stop being helpful once more than 28 ranks are in use. Combined with the near-linear heads and tails visible in Fig. 1, this indicates that limited runKernel parallelism is the main driver of total latency when few ranks are in use, whereas DPU allocation (up to 500 ms) and limited data throughput are the bottlenecks when using (almost) all available ranks. So, it is clear that optimizing PIM kernel implementations will likely only lead to marginal improvements.

With UPMEM SDK 2025.1.0, runKernel latency does not change beyond the measurement uncertainty. DPU allocation latency and data transfers to PIM improve notably, whereas kernel upload becomes slightly (but not significantly) slower.

$$
\begin{aligned}
T_{\text{alloc}} &= 23.3\,\text{ms} + 2.5\,\text{ms} \cdot \#\mathit{ranks} \\
T_{\text{loadKernel}} &= 524\,\mu\text{s} + 2.8\,\mu\text{s} \cdot \#\mathit{instr} + 18.2\,\mu\text{s} \cdot \#\text{ranks} \\
B_{\text{writeData}} &= 4798\,\tfrac{\text{MB}}{\text{s}} + 348\,\tfrac{\text{MB}}{s} \cdot \min(\#\mathit{ranks}, 22.7) \\
B_{\text{readResults}} &= 2656\,\tfrac{\text{MB}}{\text{s}} + 174\,\tfrac{\text{MB}}{s} \cdot \min(\#\mathit{ranks}, 27.9)
\end{aligned}
$$

Data transfer to PIM modules is much faster, especially when only few ranks are allocated, thus explaining why the high-latency head in Fig. 1 is nearly gone. While there is no throughput improvement beyond 23 ranks, the penalty for allocating more ranks is lower, hence the tail is also less pronounced.

Moreover, the behaviour model allows users to determine for which configurations PIM execution is faster than CPU execution.

Table 1: Cross-validated prediction error and complexity score of evaluated CART and behaviour model variants.

Metric	CART	CART-B	BM-CART	**BM-RMT**
Complexity	2,937	4,391	2,899	**22**
Error	14.6 %	3.8 %	3.8 %	**7.7 %**

For instance, on our own server, PIM is only beneficial once column size exceeds 256 MiB (COUNT) to 2 GiB (SELECT), and when at least two consecutive queries are to be expected [23]. All of these insights are only feasible thanks to the combination of runtime variability, behaviour models, and regression model trees.

3.2 Quantitative Evaluation

Regression model trees deliberately trade accuracy for interpretability. Here, we examine how much accuracy they lose, and show that they are still sufficiently accurate for reasoning about product line performance. Our baseline (CART) consists of a CART model based on the variability model shown in Fig. 2. We compare it with a CART model that also knows about the amount of SELECT, COUNT, and UPDATE operations in the benchmark (CART-B), a behaviour model augmented with CART performance models (BM-CART), and a behaviour model augmented with regression model trees (BM-RMT). Column size is variable, but constant during each individual benchmark run, hence the two CART-only models and the two behaviour model variants can be compared in a fair manner.

Table 1 gives prediction error after cross-validation and model complexity (as a proxy metric for interpretability) for all four evaluation targets. Model complexity is the number of decision nodes in all referenced CART or RMT models plus the number of leaf nodes (CART) or least-squares regression weights referenced in leaf nodes (RMT). We see that awareness of workload-dependent runtime variability is mandatory for accurate performance predictions, and that CART are too complex for manual performance analysis. Behaviour models with regression model trees, on the other hand, reduce model complexity by two orders of magnitude. While this comes at the cost of higher prediction error, the models are still sufficiently accurate for gaining insights into product line performance. In our opinion, this trade-off is well worth it.

Data acquisiiton for CART and CART-B produces $2 \cdot 13 \cdot 13 \cdot 8 = 2704$ samples. Meanwhile, data acquisition for BM-CART and BM-RMT uses just 40 samples for alloc, writeData and readResult, $40 \cdot 13$ samples for loadKernel, and $2 \cdot 13 \cdot 13$ for runKernel, resulting in $3 \cdot 40 + 40 \cdot 16 + 2 \cdot 13 \cdot 13 = 1098$ samples. Moreover, as BM-CART and BM-RMT runKernel measurements do not have to take consecutive queries into account, total daca acquisition time for behaviour models is an order of magnitude lower than for CART-B.

4 Related Work

As discussed in Section 2.2, state machine-based behaviour models are well-known in the product line engineering domain [4], as is their extension with constant energy or latency attributes [10, 19, 36]. Applications include model checking [14, 15], model-based analysis of product families [8, 17, 31], and similar [16].

The only approaches that combine behaviour models with arbitrary performance models that we are aware of originate from energy modelling for cyber-physical systems and not SPL research. There, engineers often utilize state machine models (“energy models”) to describe the relation between runtime system behaviour and latency or energy usage [13, 49]. While these take runtime configuration into account, they lack a formal concept of configurable features, and do not follow product line engineering guidelines.

Regression model trees also originate from the energy modelling domain [24]. Their closest relative from the SPL domain are linear model trees [32, 39, 40]. These extend the CART learning algorithm with a bottom-up pruning step that merges sub-trees into linear regression functions. RMT, by contrast, support arbitrary functions in leaf nodes and do not use numeric features in decision nodes, making them easier to understand. Provably optimal and interpretable sparse regression trees are a promising candidate [47], but do not support categorical or numeric variability yet.

5 Conclusion and Future Work

We have shown how product line engineers and users can understand the runtime performance of software product lines by combining runtime variability (dynamic software product lines), behaviour models (featured transition systems), and regression model trees. Our novel combination of these three concepts reduces the runtime of training data acquisition by an order of magnitude and the complexity of the resulting performance models by two orders of magnitude, while retaining sufficient model accuracy for reasoning about product line performance.

We expect that these methods are applicable to a wide range of product lines and performance attributes beyond the custom DBMS application that we have used as a running example and evaluation target. For instance, as mentioned in the previous section, behaviour models and regression model trees also support power and energy usage prediction. We expect that other attributes, such as runtime memory usage or carbon emissions, can also be handled with ease.

Currently, the structure of the behaviour model must be provided manually, and acquisition of benchmark data in our DBMS example required changes to its source code for timing the steps involved in PIM operations. Neither of this is a design limitation. On the one hand, learning state machines from software systems is a well-researched topic [12], and we expect that we can apply those algorithms to behaviour models (featured transition systems) as well. On the other hand, depending on desired model granularity, source code changes may not be required. For instance, an SQLite behaviour model that encodes the latency of individual database operations (COUNT, SELECT, UPDATE, . . .) rather than an entire benchmark may already be good enough for many applications.

In our opinion, it is time for runtime performance models to step away from predicting performance attributes for fixed reference benchmarks, and instead become aware of runtime configuration and workload. Runtime variability, behaviour models, and regression model trees offer a first step into this direction.

Acknowledgments

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 502565817.

References

[1] Andreas Abele, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, David Servat, Martin Törngren, and Matthias Weber. 2010. The CVM Framework – A Prototype Tool for Compositional Variability Management. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems* (Linz, Austria) *(VaMoS '10)*, David Benavides, Don S. Batory, and Paul Grünbacher (Eds.). Universität Duisburg-Essen, 101–105. doi:10.17185/duepublico/47086

[2] Mathieu Acher, Hugo Martin, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Olivier Barais, and Juliana Alves Pereira. 2022. 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* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 85–96. doi:10.1145/3546932.3546997

[3] Timo Asikainen, Tomi Mannisto, and Timo Soininen. 2006. A Unified Conceptual Foundation for Feature Modelling. In *Proceedings of the 10th International Software Product Line Conference* (Baltimore, MD, USA) *(SPLC '06)*. IEEE, 31–40. doi:10.1109/SPLINE.2006.1691575

[4] Joanne M. Atlee, Uli Fahrenberg, and Axel Legay. 2015. Measuring Behaviour Interactions between Product-Line Features. In *Proceedings of the 3rd FME Workshop on Formal Methods in Software Engineering* (Florence, Italy) *(FormaliSE '15)*. IEEE, 20–25. doi:10.1109/FormaliSE.2015.11

[5] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Accelerating Large Table Scan using Processing-In-Memory Technology. In *Proceedings of the 20th Conference on Database Systems for Business, Technology and Web (BTW'23)*. Gesellschaft für Informatik e.V., Bonn, 797–814. doi:10.18420/BTW2023-51

[6] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Adaptive Query Compilation with Processing-in-Memory. In *Proceedings of the 39th International Conference on Data Engineering Workshops (ICDEW '23)*. IEEE, 191–197. doi:10.1109/ICDEW58674.2023.00035

[7] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Processing-in-Memory for Databases: Query Processing and Data Transfer. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (Seattle, WA, USA) *(DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 107–111. doi:10.1145/3592980.3595323

[8] Harsh Beohar and Mohammad Reza Mousavi. 2016. Input–output conformance testing for software product lines. *Journal of Logical and Algebraic Methods in Programming* 85, 6 (2016), 1131–1153. doi:10.1016/j.jlamp.2016.09.007 NWPT 2013.

[9] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. 2010. Introducing TVL, a Text-based Feature Modelling Language. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems* (Linz, Austria) *(VaMoS '10)*. Universität Duisburg-Essen, 159–162. doi:10.17185/duepublico/47086

[10] Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, Nicolas Markey, and Jiří Srba. 2008. Infinite Runs in Weighted Timed Automata with Energy Constraints. In *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'08)*. Springer Berlin, Heidelberg, 33–47. doi:10.1007/978-3-540-85778-5_4

[11] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. 1984. *Classification and Regression Trees* (1 ed.). Routledge. doi:10.1201/9781315139470

[12] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. 2016. Active learning for extended finite state machines. *Formal Aspects of Computing* 28, 2 (2016), 233–263. doi:10.1007/s00165-016-0355-5

[13] Nadir Cherifi, Thomas Vantroys, Alexandre Boe, Colombe Herault, and Gilles Grimaud. 2017. 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* (Prague, Czech Republic) *(FiCloud '17)*. IEEE, 120–127. doi:10.1109/FiCloud.2017.53

[14] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. 2014. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming* 80, PB (2 2014), 416–439. doi:10.5555/2748144.2748397

[15] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1069–1089. doi:10.1109/TSE.2012.86

[16] Maxime Cordy, Xavier Devroey, Axel Legay, Gilles Perrouin, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Jean-François Raskin. 2019. *A Decade of Featured Transition Systems*. Springer International Publishing, Cham, 285–312. doi:10.1007/978-3-030-30985-5_18

[17] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2012. Behavioural Modelling and Verification of Real-Time Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1* (Salvador, Brazil) *(SPLC '12)*. Association for Computing Machinery, New York, NY, USA, 66–75. doi:10.1145/2362536.2362549

[18] Johannes Dorn, Sven Apel, and Norbert Siegmund. 2021. Mastering Uncertainty in Performance Estimations of Configurable Software Systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 684–696. doi:10.1145/3324884.3416620

[19] Uli Fahrenberg and Axel Legay. 2017. Featured Weighted Automata. In *Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE '17)*. IEEE, 51–57. doi:10.1109/FormaliSE.2017.2

[20] Birte Friesel. 2025. *Understanding Product Line Runtime Performance with Behaviour Models and Regression Model Trees (Artefact)*. doi:10.5281/zenodo.15827230

[21] Birte Friesel, Markus Buschhoff, and Olaf Spinczyk. 2018. Parameter-Aware Energy Models for Embedded-System Peripherals. In *Proceedings of the 13th International Symposium on Industrial Embedded Systems* (Graz, Austria) *(SIES '18)*. IEEE, 4 pages. doi:10.1109/SIES.2018.8442096

[22] Birte Friesel, Kathrin Elmenhorst, Lennart Kaiser, Michael Müller, and Olaf Spinczyk. 2022. 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* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 58–61. doi:10.1145/3503229.3547026

[23] Birte Friesel, Marcel Lütke Dreimann, and Olaf Spinczyk. 2025. Lightning Talk: Feasibility Analysis of Semi-Permanent Database Offloading to UPMEM Near-Memory Computing Modules. In *Datenbanksysteme für Business, Technologie und Web – Workshopband*. Gesellschaft für Informatik, Bonn, Germany, 355–366. doi:10.18420/BTW2025-140

[24] Birte Friesel and Olaf Spinczyk. 2022. Regression Model Trees: Compact Energy Models for Complex IoT Devices. In *Proceedings of the Workshop on Benchmarking Cyber-Physical Systems and Internet of Things* (Milan, Italy) *(CPS-IoTBench '22)*. IEEE, 1–6. doi:10.1109/CPS-IoTBench56135.2022.00007

[25] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. 2013. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*. IEEE, 301–311. doi:10.1109/ASE.2013.6693089

[26] 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 Software Engineering* 23, 3 (6 2018), 1826–1867. doi:10.1007/s10664-017-9573-6

[27] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* 10 (2022), 52565–52608. doi:10.1109/ACCESS.2022.3174101

[28] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic Software Product Lines. *Computer* 41, 4 (2008), 93–95. doi:10.1109/MC.2008.123

[29] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: an exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 497–508.

[30] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. 2023. Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs. *Proceedings of the ACM on Management of Data* 1, 2, Article 113 (jun 2023), 27 pages. doi:10.1145/3589258

[31] Lars Luthmann, Andreas Stephan, Johannes Bürdek, and Malte Lochau. 2017. Modeling and Testing Product Lines with Unbounded Parametric Real-Time Constraints. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A* (Sevilla, Spain) *(SPLC '17)*. Association for Computing Machinery, New York, NY, USA, 104–113. doi:10.1145/3106195.3106204

[32] Donato Malerba, Floriana Esposito, Michelangelo Ceci, and Annalisa Appice. 2004. Top-down induction of model trees with regression and splitting nodes. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 5 (5 2004), 612–625. doi:10.1109/TPAMI.2004.1273937

[33] Sally A. McKee. 2004. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers (CF '04)*. Association for Computing Machinery, 162. doi:10.1145/977091.977115

[34] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 257–267. doi:10.1145/3106237.3106238

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

[36] Rafael Olaechea, Uli Fahrenberg, Joanne M. Atlee, and Axel Legay. 2016. Long-term average cost in featured transition systems. In *Proceedings of the 20th International Systems and Software Product Line Conference* (Beijing, China)

(SPLC '16). Association for Computing Machinery, New York, NY, USA, 109–118. doi:10.1145/2934466.2934473

[37] 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 4th 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. doi:10.1145/2420942.2420944

[38] 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. doi:10.1016/j.jss.2021.111044

[39] John R Quinlan et al. 1992. Learning with Continuous Classes. In *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence* (Hobart, Tasmania) *(AI '92)*. World Scientific, 343–348. doi:10.1142/1897

[40] Santosh Singh Rathore and Sandeep Kumar. 2016. A Decision Tree Regression based Approach for the Number of Software Faults Prediction. *SIGSOFT Software Engineering Notes* 41, 1 (1 2016), 6 pages. doi:10.1145/2853073.2853083

[41] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. 2011. Multi-Dimensional Variability Modeling. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems* (Namur, Belgium) *(VaMoS '11)*. Association for Computing Machinery, New York, NY, USA, 11–20. doi:10.1145/1944892.1944894

[42] Atrisha Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE, 342–352. doi:10.1109/ASE.2015.45

[43] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE '15)*. Association for Computing Machinery, New York, NY, USA, 284–294. doi:10.1145/2786805.2786845

[44] Norbert Siegmund, Marko Rosenmuller, Christian Kastner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. 2011. Scalable Prediction of Non-functional Properties in Software Product Lines. In *Proceedings of the 15th International Software Product Line Conference* (Munich, Germany) *(SPLC '11)*. IEEE, 160–169. doi:10.1109/SPLC.2011.20

[45] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G Giarrusso, Sven Apel, and Sergiy S Kolesnikov. 2013. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology* 55, 3 (3 2013), 491–507. doi:10.1016/j.infsof.2012.07.020

[46] Julio Sincero, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. 2010. Approaching Non-functional Properties of Software Product Lines: Learning from Products. In *Proceedings of the Asia Pacific Software Engineering Conference* (Sydney, Australia) *(APSEC '10)*. IEEE, 147–155. doi:10.1109/APSEC.2010.26

[47] Rui Zhang, Rui Xin, Margo Seltzer, and Cynthia Rudin. 2023. Optimal Sparse Regression Trees. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence* (Washington, DC, USA) *(AAAI '23)*. AAAI Press, 11270–11279. doi:10.1609/aaai.v37i9.26334

[48] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance Prediction of Configurable Software Systems by Fourier Learning. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, NE, USA) *(ASE '15)*. IEEE, 365–373. doi:10.1109/ASE.2015.15

[49] Nanhao Zhu and Athanasios V. Vasilakos. 2016. A generic framework for energy evaluation on wireless sensor networks. *Wireless Networks* 22, 4 (5 2016), 1199–1220. doi:10.1007/s11276-015-1033-x