Bayesian Multi-Level Performance Models for Multi-Factor Variability of Configurable Software Systems

Johannes Dorn,
Stefan Mühlbauer, Stefan Jahns
Leipzig University, Leipzig, Germany

Sven Apel
Saarland University
Saarbrücken, Germany

Norbert Siegmund
Leipzig University &
ScaDS.AI Dresden/Leipzig, Germany

Abstract

Tuning a software system's configuration is essential to meet performance requirements. However, not only do configuration options affect performance, but also the system's interaction with external factors such as the workload. Hence, tuning requires understanding how a specific *setting* of external factors (e.g., a specific workload) in combination with the system configuration influences performance. To address this issue, we propose HyPerf, a *Bayesian multi-level performance modeling approach* that systematically distinguishes between *setting-invariant* and *setting-variant* influences, that is, influences that remain consistent across settings versus those that exhibit substantial variation. With HyPerf, we aim at *balancing accuracy and efficiency*, achieving robust performance predictions with significantly fewer training samples. Unlike the state of the art, HyPerf is able to *identify a minimal set of settings* that captures essential performance variations, so that developers can approximate whether all setting-variant influences have been accounted for. Empirical evaluations on ten real-world software systems across up to 35 workloads and scalability experiments on the Linux kernel demonstrate that HyPerf matches or outperforms state-of-the-art approaches while requiring fewer measurements. Notably, HyPerf is indeed capable of *interpretable performance reasoning* and can identify minimal workload subsets that capture essential performance variations.

CCS Concepts

• **Software and its engineering** → *Software product lines*; **Software performance**; • **Computing methodologies** → **Machine learning approaches**; **Multi-task learning**.

Keywords

Performance-Influence Modeling, Bayesian Multi-Level Modeling

ACM Reference Format:
Johannes Dorn, Stefan Mühlbauer, Stefan Jahns, Sven Apel, and Norbert Siegmund. 2026. Bayesian Multi-Level Performance Models for Multi-Factor Variability of Configurable Software Systems. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773131

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773131

1 Introduction

Most modern software systems provide configuration options to tailor their behavior towards functional and non-functional requirements as well as to *external factors* such as workloads and hardware. Tuning a software system's configuration to meet performance requirements must, therefore, account not only for the influence of configuration options, but also for their interaction with external factors.

Prior research has explored learning predictive models that estimate the performance of unseen configurations to guide the configuration process [4, 8, 10, 12, 13, 39, 49]. Yet, most of these approaches assume a fixed *setting*, where external factors, such as workload and hardware, remain constant. In practice, however, software systems operate in vastly varying settings, rendering such performance estimates unreliable. More recent approaches incorporate external factors as features [22, 25, 30], treating them as dedicated configuration options. This design introduces scalability challenges: many factors, such as hardware, are inherently qualitative, and encoding them inflates model complexity. Even worse, as external factors interact with configuration options, the model size easily explodes when capturing all interactions. Alternatively, transfer learning might be able to reduce data needs when adapting to a new setting [19, 42]. Unfortunately, this implies that, for every new setting, we need to redo transfer learning, because it does not capture whether a configuration's performance remains stable or varies across settings.

More generally, existing approaches offer partial solutions but fail to capture how configuration options influence performance across settings in a way that remains both interpretable and efficient. This calls for a novel approach—one that extends interpretable models without assuming fixed influences or requiring excessive measurements. A natural way to extend models across factors is to train either a separate model for each setting or a single model across all settings. Since a single setting is defined by the characteristics of the external factors (e.g., different system inputs for the workload factor), the number of possible settings is potentially unbounded. So, learning a model per setting ensures high accuracy but requires extensive additional measurements, leading to poor sample efficiency. Moreover, single-factor models cannot capture whether a configuration's performance remains stable or varies across settings, such that they hardly generalize. Learning instead the influences across all settings within a single model improves efficiency, but assumes that performance influences remain constant. This is often unrealistic, as configuration options may have conditional effects depending on the setting, as has been demonstrated for varying workloads [19, 34, 42].

Neither of the two (i.e., learning a separate model for each setting or a single model across all settings) systematically distinguishes

setting-invariant and *setting-variant* influences, that is, influences that remain consistent across settings versus influences that exhibit substantial variation. Making this distinction explicit is the essential idea of this work—we introduce a third strategy: a *Bayesian multi-level performance modeling approach*, called HyPerf. Rather than treating external factors as fixed conditions or additional options, HyPerf models them in a hierarchy in which the upper level identifies stable performance influences that hold across settings, while the lower level refines these estimates with deviations that are specific to a particular setting. This allows HyPerf to balance accuracy and sample efficiency while also improving interpretability.

Beyond improving prediction accuracy, HyPerf addresses a critical bottleneck in performance testing. Traditional methods evaluate software across an exhaustive set of workloads (as an important factor) to account for possible performance variations—an expensive process. Instead, HyPerf is able to identify a *minimal coverage workload subset* that captures essential variability in option influences, significantly reducing the time and cost of testing.

To evaluate HyPerf's effectiveness, we conduct an empirical study on *ten real-world software systems*, each tested across up to *35 workloads* as the factor of interest, thereby building up one of the most comprehensive sets of evaluation material in this field. Our results demonstrate that HyPerf achieves a predictive accuracy on par with or exceeding that of models tailored to the given setting while requiring fewer measurements. Compared to complex black-box state-of-the-art learners, HyPerf delivers competitive accuracy at a fraction of the training cost. A scalability analysis with the Linux kernel further confirms that HyPerf remains tractable even for systems with thousands of options. More importantly, HyPerf provides actionable insights into configuration–workload interactions, enabling targeted performance tuning and efficient workload selection. By improving both efficiency and interpretability, HyPerf offers a scalable solution for performance modeling in configurable systems, reducing the cost of performance testing without sacrificing accuracy.

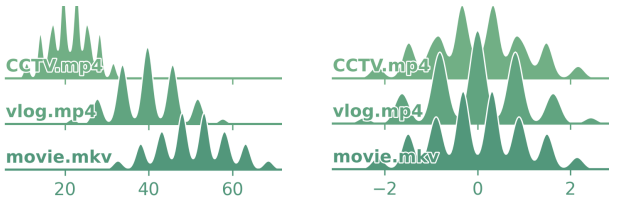
In summary, we make the following contributions:

- a Bayesian multi-level modeling approach, called HyPerf, for modeling software performance considering multiple factors;
- a comparison of HyPerf's prediction accuracy with state-of-the-art methods for ten software systems across up to 35 workloads, highlighting scenarios where the methods are most effective;
- a scalability analysis based on the Linux kernel;
- an investigation of HyPerf's multi-level reasoning capability, revealing configuration options whose influence is setting-invariant or varies substantially across settings;
- a demonstration of HyPerf's ability to calculate a coverage workload set, reducing time and cost in performance testing;
- a companion website[1] to replicate our results and measurements.

2 Running Example & Related Work

In this section, we provide basic definitions used throughout the paper, and we introduce a running example to explain existing approaches for constructing performance models that consider a single or multiple factors.

[1] https://github.com/AI-4-SE/hyperf-companion/



(a) Absolute performance (s) **(b) Standardized performance**

Figure 1: Performance distribution across EncodeX's settings

2.1 Preliminaries

A configurable software system provides a set of configuration *options* O, each possibly binary $(\text{dom}(o) = \mathbb{B})$ or numeric $(\text{dom}(o) \subseteq \mathbb{N})$. A *configuration* is a mapping from options to values in their respective domains. The set of valid configurations C results from the Cartesian product of these domains, restricted by inter-option constraints. Hence, a configuration $\boldsymbol{c} \in C$ is a vector whose elements represent the values assigned to each option.

Once configured, a software system is deployed into a specific *setting* s, which represents the values of external factors that affect the software behavior, e.g., the specific workload. Since these factors are often inherently qualitative (e.g., different operating systems and hardware), we treat settings as categorical rather than attempting numeric encodings of categorical values. Let $\mathcal{S}$ be the set of all settings, we define a software system's *execution context* $x \in \mathcal{X}$ as a pair of configuration and setting, giving rise to the execution context space $\mathcal{X} = C \times \mathcal{S}$. Given the execution context x of a software system, a performance-influence model $\hat{\Pi}(x) : \mathcal{X} \mapsto \mathbb{R}$ can be learned from a set of empirical observations to approximate the true performance behavior $\Pi(x) : \mathcal{X} \mapsto \mathbb{R}$. The measured configurations C^* and used settings $\mathcal{S}^*$ result in the set of observed execution contexts $\mathcal{X}^*$. The resulting measurement dataset is $\mathcal{D} = \{ (x, \Pi(x)) \mid x \in \mathcal{X}^* \}$.

2.2 Running Example

We introduce EncodeX, a hypothetical video encoder with four configuration options evaluated using three video files as workloads: CCTV.mp4 (a 10-second 720p clip without sound), vlog.mp4 (a 20-second 1080p mobile clip), and movie.mkv (a 30-second 4k movie with multiple sound channels). EncodeX's configuration options reflect typical categories of encoding options, specifically nullO (no effect), constO (always adds 5 seconds), varyO (increases time relative to video length [17]), and conO (influence varies conditional on sound presence [34]) with two options per category. EncodeX's execution time $\Pi^{\text{EncodeX}} : \mathcal{X} \mapsto \mathbb{R}$ depends both on the configuration and the workload. The performance of a given configuration is calculated by multiplying the vector of workload-specific influences $\boldsymbol{\beta_s}$ by the configuration vector $\boldsymbol{c}$ and adding a workload-specific base run-time of α_s. Adding a 5% noise factor ε, the execution time of a given configuration is given by $\Pi^{\text{EncodeX}}(s, \boldsymbol{c}) = \alpha_s + \boldsymbol{\beta_s c} + \varepsilon$. Figure 1a shows how performance varies across workloads, highlighting setting-specific shifts and conditional effects (e.g., handling audio only matters for certain videos). Such multi-factor variability complicates the modeling process.

2.3 Modeling Single-Factor Software Variability

Performance-influence modeling aims at understanding how different options and their interactions affect software performance [38]. Most performance-influence models consider configurability as the sole factor and assume a fixed setting [20]. Among those, linear-regression-based methods [7, 8, 39] capture option influences and interactions in an interpretable additive form. Others employ Fourier expansions [14, 49], Bayesian approaches [8], rule-based trees [11, 12], or white-box data [44, 47] from dynamic and static analysis [40, 45, 46]. All these approaches rely on performance data originating from a single setting, limiting their ability to generalize across different workloads or hardware. For instance, such an approach would falsely dismiss option conO as non-influential when only analyzing EncodeX's first workload, overlooking its influence on others.

2.4 Modeling Multi-Factor Software Variability

Multi-factor software variability captures the interplay of factors that affect software performance, including not only configurability but also variations in hardware, operating systems, and input data [28]. In what follows, we outline the challenges that this complexity presents for model adaptability across settings and review existing approaches that address these challenges.

2.4.1 Performance Variation Across Settings. The performance of a configuration may correlate across similar settings [17], prompting the idea of a single *"bellwether"* setting [24] to generalize performance-influence models across settings. However, not all settings are sufficiently similar, and certain options can exhibit drastically different or conditional performance influences [29, 34]. Thus, reusing a single model can be unreliable, motivating techniques that explicitly account for setting-specific influences.

2.4.2 Transfer Learning. Transfer learning leverages data from a well-understood setting to reduce the effort for sampling and learning in a new one, for instance, by identifying influential options [18] or transferring a kernel function [19, 42]. Although effective for pairwise setting transfers, these methods typically rely on a *single* source model. As a result, when learning a model for a third setting, such transfer-learning methods lack the ability to pin down options that are consistently influential across settings, that can be disregarded, and that may be conditionally influential.

2.4.3 Multitask Learning. While transfer learning adapts knowledge from a single source setting to a new one, multitask learning aims to build a *single* model for multiple settings. Several lines of work touch on multitask principles. Krishna et al. propose using a single bellwether setting for aggregating cross-setting measurements [25]. However, this strategy is undermined by the substantial variability of option influences across settings, as reported by related work [29, 34]. Ding et al. first characterizes workloads [6] to derive new workload features and then evaluates different machine learning algorithms in both a traditional and an active-learning setting [22, 30]. However, it is limited to setting changes with a quantifiable characteristic. Even for such setting changes, it attributes the setting variability of options' influences to the new characteristic instead, obscuring interpretability. In contrast to transfer learning, multitask learning simultaneously considers multiple settings at once, which is key to identify both invariant and conditionally influential options. The challenges of sparse data and reported substantial variance in option influences across settings [29, 34] highlight not only transfer learning's inadequacies but also underscore the necessity for multitask learning's holistic approach.

3 Multi-Level Performance-Influence Modeling

Data sparsity is a major challenge resulting from the high effort of conducting performance measurements. Considering multiple factors elevates this issue even further. Relying on an already constrained measurement budget, the inflated problem space $\mathcal{X}$ (see Section 2.1) limits the ability of traditional performance-influence models to offer reliable predictions across settings. To address this issue, we explain how to process sparse multi-factorial data to allow models to detect commonalities. From this, we derive how to extend single-factor models to accommodate multiple factors.

3.1 Processing Multi-Factorial Data

As our goal is to model performance-influence changes across settings, and given that factors are inherently qualitative in nature, we treat the setting as a categorical variable rather than attempting to extract numerical characteristics. This decision shapes how we process multi-factorial data such that it supports model evaluation, enables finding commonalities across settings, and facilitates interpretability and learning.

3.1.1 Stratification. To ensure that our model is able to capture which performance influences change across settings, we incorporate performance observations stemming from different settings. Therefore, we stratify with respect to the setting and split our data $\mathcal{D}$ into $|\mathcal{S}^*|$ distinct datasets $\mathcal{D}_s$, with $s \in \mathcal{S}^*$. Before training a model, we apply a split operation to each dataset $\mathcal{D}_s$ to obtain a training set $\mathcal{D}_s^{\text{train}}$ and a test set $\mathcal{D}_s^{\text{test}}$, giving rise to a corresponding global training sample $\mathcal{D}^{\text{train}} = \bigcup_{s \in \mathcal{S}^*} \mathcal{D}_s^{\text{train}}$ and a global test sample $\mathcal{D}^{\text{test}} = \bigcup_{s \in \mathcal{S}^*} \mathcal{D}_s^{\text{test}}$.

3.1.2 Common Scale. As our data stem from different settings, performance observations can vary substantially in scale. To mitigate this, we standardize the *performance values* $\Pi(\mathcal{D}_s^{\text{train}})$ of each distinct dataset $\mathcal{D}_s$ by subtracting the mean performance and dividing by the standard deviation. This standardization implements a linear transfer of data between settings, addressing the common shifting and scaling effects in multi-factorial data and thereby aligning performance distributions [19, 29, 34, 34, 42], as shown in Figure 1.

To ensure that the *option values* are within the same order of magnitude, as required by various machine learning techniques such as regularization, we normalize the values of each option $o \in O$ to the range $[0, 1]$ via min-max scaling. This step leaves binary options unchanged, while the values of numeric options are scaled to the same order of magnitude.

3.1.3 Reducing Multicollinearity. The values of configuration options may be correlated, such as an option being selected in all configurations (mandatory) or groups of option values being mutually exclusive (alternative groups). Although such *multicollinearity* does not affect predictive accuracy, it blurs the attribution of performance influences to individual options [5]. To correct for multicollinearity, we follow a method from the literature [8]. Specifically,

we transform our dataset $\mathcal{D}$ by selecting and removing a default option from mutually exclusive options and removing options with unchanged values in $\mathcal{D}^{\text{train}}$, as they do not account for any performance variation. This transformation implicitly sets the influence of removed options to zero and removes interactions between them and other options from the dataset, since such interactions were indistinguishable from individual options' influences with the given data. This leads to simpler models where the influences of the remaining options are interpretable.

3.2 Quantifying Uncertainty

Given the sparsity of multi-factorial data, Bayesian models provide a promising alternative to frequentist models, as they explicitly model uncertainty arising from data sparsity, measurement noise, and the variance across settings. In what follows, we explain the concept and implementation of Bayesian models and how they can cope with data sparsity due to setting-specific budget splits.

Bayesian modeling provides a principled way to handle uncertainty and variance in data analysis. Using Bayes' theorem, initial beliefs about model parameters θ are updated based on new evidence, i.e., training data $\mathcal{D}^{\text{train}}$. Model parameters are represented as random variables, each with a *probability density function* (PDF) that indicates the likelihood of different values for a random variable. Bayesian models combine these random variables to compute PDFs for predictions, denoted as $\tilde{\Pi}$.

Three key components define Bayesian models. First, the *prior* $P(\theta)$ represents initial assumptions about the distribution of model parameter values θ. In our case, priors capture the observation that most configuration options have negligible impact on performance [23] and that option influences remain stable across similar settings but may diverge in less comparable settings [17]. To avoid introducing arbitrary biases, we select priors that maximize entropy while ensuring high probability mass in $[-1, 1]$, matching the standardized performance scale and accounting for possible outliers. This strategy balances informativeness with flexibility: too narrow priors could miss important effects, whereas too wide priors would provide insufficient regularization. Second, the *posterior* $P(\theta \mid \mathcal{D})$ represents the updated beliefs about parameter values after considering new data, blending prior knowledge with observed evidence. Third, the *likelihood* $P(\mathcal{D} \mid \theta)$ quantifies how probable observed data is under a given set of model parameters. Unlike non-Bayesian methods, this allows us to formalize how options influence performance, opening unique ways of learning from data.

3.3 To Pool, or Not to Pool, or To Pool Partially

Capturing the variation of options' performance influences across settings requires a balance between flexibility and interpretability. While complex models capture fine-grained interactions, they often obscure insights. With preprocessing that aligns performance values to a common scale, linear models allow us to extract meaningful patterns across settings without unnecessary complexity. In what follows, we discuss different ways to handle multi-factorial data with additive Bayesian models.

3.3.1 No Pooling (np). Most existing performance-influence models assume a fixed setting and vary only the configuration as the only considered *factor*. A natural extension of such models to multiple settings would fit a dedicated model for each setting. Here, no data are shared (*pooled*) across models, which, consequently, captures only influences within a setting. In the case of EncodeX, the data $\mathcal{D}$ is split into three setting-specific datasets $\mathcal{D}_{\text{CCTV.mp4}}$, $\mathcal{D}_{\text{vlog.mp4}}$, $\mathcal{D}_{\text{movie.mkv}}$, and a separate model on each dataset. For instance, Mühlbauer et al. apply Lasso models with no-pooling, requiring a comprehensive training set for each workload [34].

In our implementation of the *no-pooling* strategy, the likelihood function starts with a base performance α_s (analogous to the intercept in a linear model) and a vector of option influences $\boldsymbol{\beta}_s$ for each setting s. For the prior distribution of the base influence, we assume only that there is finite uncertainty about a most likely value. With this assumption, the normal distribution $\mathcal{N}(\mu, \sigma)$ is the maximum entropy probability distribution, making it a natural choice:

$$\alpha_s \sim \mathcal{N}(0, 1) \quad \forall s \in \mathcal{S}^* \tag{1a}$$

Acknowledging that not all options have a measurable influence on performance, we apply Lasso regularization to eliminate non-influential options. For this purpose, we choose a Laplace prior $\mathcal{L}$ for the option influences $\boldsymbol{\beta}_s$, because Lasso can be derived as the posterior mode of a Laplace prior [41]. The Laplace distribution's equivalent to Lasso's regularization strength-defining penalty hyperparameter is its spread parameter b. Recognizing that the share of influential options and, hence, the appropriate regularization strength differs across systems, we let the model infer the best regularization strength from the data:

$$b \sim \text{Exp}(1) \tag{1b}$$

$$\boldsymbol{\beta}_s \sim \mathcal{L}(0, b) \quad \forall s \in \mathcal{S}^* \tag{1c}$$

As a *hyper prior*, b introduces the first new model level because it priors the spread prior of the Laplace distribution. The exponential distribution is a suitable prior for b as it enforces positivity, and it is the maximum entropy distribution when we assume only that an expected value exists.

For each setting, we model the performance $\tilde{\Pi}^{\text{np}}(s, \boldsymbol{c})$ as a linear combination of option influences and a given configuration, offset by the base. While this model $\tilde{\Pi}^{\text{np}}$ can later be used for predictions, learning the likelihood $\mathcal{D} \mid \theta$ requires explicitly modeling the frequentist residual ε as the observation variance σ_s^2:

$$\tilde{\Pi}^{\text{np}}(s, \boldsymbol{c}) = \alpha_s + \boldsymbol{\beta}_s \boldsymbol{c} \quad \forall s \in \mathcal{S}^* \tag{1d}$$

$$\sigma_s^2 \sim \text{Exp}(1) \quad \forall s \in \mathcal{S}^* \tag{1e}$$

$$\mathcal{D} \mid \theta \sim \mathcal{N}\Big(\tilde{\Pi}^{\text{np}}(s, \boldsymbol{c}),\, \sigma_s^2\Big) \quad \forall s \in \mathcal{S}^* \tag{1f}$$

Since the individual models do not share information across settings, the no-pooling approach cannot generalize to unseen settings without a whole new set of training data. Combining single-factor models, no-pooling serves as a base-line multi-factor approach.

3.3.2 Complete Pooling (cp). In contrast to the no-pooling approach, *complete pooling* aggregates all training data without distinguishing between settings, and a single model is trained on this *complete pool* of data. While this strategy cannot detect setting-specific variations, it efficiently learns common influences by using the entire dataset. For EncodeX, a complete-pooling model could derive performance predictions for a new setting, which will be inaccurate for options with setting-dependent influences such as

conO. However, this model would correctly capture the common relative influence of varyO.

Our implementation of the complete-pooling strategy considers a single base influence α and a single vector of influences $\boldsymbol{\beta}$, regularized by the spread hyper prior b. It remains setting-agnostic:

$$\alpha \sim \mathcal{N}(0,1) \quad (2a) \qquad \boldsymbol{\beta} \sim \mathcal{L}(0,b) \quad (2d)$$

$$b \sim \text{Exp}(1) \quad (2b) \qquad \sigma^2 \sim \text{Exp}(1) \quad (2e)$$

$$\tilde{\Pi}^{\text{cp}}(s,\boldsymbol{c}) = \alpha + \boldsymbol{\beta c} \quad (2c) \qquad \mathcal{D} \mid \theta \sim \mathcal{N}\Big(\tilde{\Pi}^{\text{cp}}(s,\boldsymbol{c}),\, \sigma^2\Big) \quad (2f)$$

3.3.3 Partial Pooling (pp). Acknowledging that some option influences may vary across settings while others do not [17, 18, 34], *partial* pooling combines the generalizability of complete pooling with the flexibility of no pooling. Partial pooling is able to learn that EncodeX's option varyO has a consistent relative influence across settings, but allows setting-specific influences for options with varying influences, such as conO. To achieve this, we use the complete-pooling model as prior for the expected influences of the no-pooling models, adding a new level to the model. This results in our partial-pooling approach, HyPerf:

$$\alpha_\mu \sim \mathcal{N}(0,1) \quad (3a) \qquad \boldsymbol{\beta}_\mu \sim \mathcal{L}(0,b) \quad (3c)$$

$$\sigma_s^2 \sim \text{Exp}(\sigma_\mu^2) \quad \forall\, s \in \mathcal{S}^* \quad (3b) \qquad b \sim \text{Exp}(1) \quad (3d)$$

$$\alpha_s \sim \mathcal{N}(\alpha_\mu, \alpha_{\sigma^2}) \quad \forall\, s \in \mathcal{S}^* \quad (3e)$$

$$\boldsymbol{\beta_s} \sim \mathcal{N}(\boldsymbol{\beta}_\mu, \boldsymbol{\beta}_{\sigma^2}) \quad \forall\, s \in \mathcal{S}^* \quad (3f)$$

$$\tilde{\Pi}^{\text{pp}}(s,\boldsymbol{c}) = \alpha_s + \boldsymbol{\beta_s c} \quad (3g) \qquad \mathcal{D} \mid \theta \sim \mathcal{N}\Big(\tilde{\Pi}^{\text{pp}}(s,\boldsymbol{c}),\, \sigma_s^2\Big) \quad (3h)$$

Here, the upper level in Equation 3c captures the general influences across settings, whereas the lower level adjusts them with setting-specific deviations in Equation 3f. This strategy constitutes multitask learning: the model shares statistical strength across tasks (settings) through shared priors while allowing for setting-specific refinements where warranted by the data. This enables HyPerf to generalize across settings without assuming uniformity. The partial-pooling approach requires additional priors for the expected variance in the base (α_{σ^2}) in Equation 3e and in options' influences ($\boldsymbol{\beta}_{\sigma^2}$) in Equation 3f as well as an expected observation noise variance σ_μ^2 in Equation 3b. Imposing both Lasso regularization and a positivity constraint, we use the exponential distribution as it is a one-sided Laplace distribution: $\alpha_{\sigma^2}, \boldsymbol{\beta}_{\sigma^2}, \sigma_\mu^2 \sim \text{Exp}(1)$. Despite not modeling interactions between options, partial pooling achieves good results (see Section 5.1.2).

4 Study Design

This section outlines our research questions and describes the subject systems and setup of our experiments. Without loss of generality, we focus on workload as the external factor in our study due to its substantial influence on execution times and to provide comparability. We further elaborate on this choice in Subsection 5.6.

4.1 Research Questions

The overarching question is whether HyPerf's multilevel design achieves a prediction accuracy on-par with existing approaches while providing additional insights about the influence of options across settings. We address this overarching question with a number of fine-grained research questions.

RQ$_1$: In which circumstances is HyPerf beneficial for performance-influence learning under varying settings?

RQ$_1$ evaluates whether the promise of multilevel models of an increased sample efficiency holds by comparing HyPerf's predictive accuracy on training sets with varying sizes. We investigate three different aspects of modeling accuracy.

RQ$_{1.1}$: How does HyPerf's multilevel performance-influence modeling compare to no and complete pooling strategies?

RQ$_{1.1}$ assesses when HyPerf's sharing of information across settings is beneficial for accurate prediction.

RQ$_{1.2}$: How does HyPerf perform compared to state-of-the-art performance prediction models?

RQ$_{1.2}$ positions HyPerf relative to established black-box learners by evaluating its predictive accuracy and analyzing trade-offs between interpretability and accuracy.

RQ$_{1.3}$: How does HyPerf scale to very large software systems?

RQ$_{1.3}$ investigates whether HyPerf's Bayesian inference and predictions through MCMC sampling limit its applicability to systems with only a limited number of configuration options.

RQ$_2$: Does HyPerf produce interpretable models for reasoning about performance influences of options across settings?

Understanding option influences across settings is crucial for performance analysis and optimization. RQ$_2$ studies how HyPerf's unified multi-level model may enable novel reasoning about option influences across settings – not possible with existing methods.

RQ$_3$: Does HyPerf provide information to derive a coverage workload set?

RQ$_3$ is concerned with a practical question that none of the previous approaches can provide answers to. For the first time, we have a means with HyPerf to determine how many different settings practitioners should consider to understand the variation of performance influences across settings and for compiling practical benchmark suites. Hence, with RQ$_3$, we investigate how many settings are needed to cover the influence variation of most options, and whose options' influences cannot feasibly be covered in a finite set as their influence is different for every new setting.

4.2 Subject Systems, Settings & Execution

For our evaluation, we compile what is, to our knowledge, the largest dataset of its kind, comprising ten real-world subject systems across diverse domains, with varying numbers of options and workloads (see Table 1). We build upon an existing dataset of eight widely used subject systems from related work [34]. We extended the dataset with measurements for two video encoders, VP9 and x265. For these systems, we selected a subset of configuration options based on those used for VP9 and x264 in related work [21]. As workloads, we used video files from the "derf's collection"[2], which spans diverse characteristics in resolution, encoding complexity, and content, ensuring workload diversity and realistic variation in performance behavior. We measured encoding times on a Debian compute cluster with Intel Core i7-8559U CPUs and 32 GB RAM per node. Each configuration-workload pair was executed five times, repeating another five times if the coefficient of variation exceeded

[2] https://media.xiph.org/video/derf/

10%, following established practice [18, 21, 34, 37, 47]. Considering multiple workloads introduces a new independent experiment variable, requiring substantially more measurements to ensure a sound empirical analysis. For the two added software systems alone, we invested 97 days of computation time to capture performance variability across different workloads. Such efforts are needed to provide a careful evaluation, limiting risks to internal and external validity. We provide all data in our companion repository.

Table 1: Overview of our ten subject systems: eight from literature [34], two are original measurements. $|\mathcal{S}^*|$: # settings; $|C^*|$: # measured configurations per workload.

System (Version)	Domain	$\lvert O\rvert$	$\lvert C^*\rvert$	$\lvert\mathcal{S}^*\rvert$
JUMP3R (1.0.4)	Audio encoder	16	4 196	6
DCONV (1.0.0-α7)	Image scaling	18	6 764	12
H2 (1.4.200)	Database	16	1 954	8
BATIK (1.14)	SVG rasterizer	10	1 919	11
XZ (5.2.0)	Data compression	33	1 999	13
LRZIP (0.651)	Data compression	11	190	13
X264 (baee40…)	Video encoder	25	3 113	9
Z3 (4.8.14)	SMT solver	12	1 011	12
VP9 (1.13.0)	Video encoder	24	302	35
X265 (3.5)	Video encoder	26	354	35

Implementation & Execution. We implemented HyPerf using `NumPyro` 0.12.1 [1, 36] in Python 3.10.4, which infers the model posterior using *Monte-Carlo Markov Chain* (MCMC) sampling. To train a model, we ran three parallel chains of the No-U-Turn sampler [16], each with 1 000 warm-up and 1 000 retained samples, which ensures robust posterior exploration and accurate credible interval estimation. We run our experiments on a server with two AMD EPYC 7302 16-core processors and 256 GB RAM. We used `Arviz` 0.16.1 [27] to compute credible intervals.

5 Results

In this section, we present the results for each RQ, along with their operationalization and a discussion.

5.1 Data Efficiency ($RQ_{1.1}$)

5.1.1 Operationalization. To assess the sample efficiency of HyPerf under varying settings, we train models for each subject system with varying training set sizes and compare their prediction accuracy. This naturally constitutes a multitask learning problem, where each setting represents a distinct but related task. Specifically, we consider the Bayesian models for no pooling ($\tilde{\Pi}^{\text{np}}$), complete pooling ($\tilde{\Pi}^{\text{cp}}$), and HyPerf (i.e., the partial pooling model $\tilde{\Pi}^{\text{pp}}$). As a baseline, we include frequentist Lasso regression [41] to represent frequentist point estimate models. We adapt it to no-pooling (equivalent to Mühlbauer et al. [34]) and complete pooling, denoted as $\hat{\Pi}^{\text{np}}_{\text{Lasso}}$ and $\hat{\Pi}^{\text{cp}}_{\text{Lasso}}$, respectively. We omit transfer-learning methods from the baselines as they adapt a pre-trained base model to each setting individually. This pairwise transfer hinders learning patterns that generalize across environments. For each subject system, we select training datasets $\mathcal{D}^{\text{train}}$ of various sizes, such that there is the same amount of data for each setting, obtained via independent random sampling. We construct training sets of sizes $\alpha \cdot |O|$ with $\alpha \in \{1/8, 1/4, 1/2, 3/4, 1, 2, 3\}$, giving rise to a series of datasets ${}^{1/8}\mathcal{D}^{\text{train}}_s \subset \ldots \subset^3 \mathcal{D}^{\text{train}}_s$, whose size is relative to the software system's number of configuration options $|O|$. This range reflects realistic measurement budgets when each setting must be sampled individually. For instance, in z3 (12 options and 12 workloads; see Table 1), $\alpha = 1/8$ yields two configurations per setting, but still produces 24 total observations. To ensure a fair evaluation, we reserve a separate testing set $\mathcal{D}^{\text{test}}_s = \mathcal{D}_s \setminus^3 \mathcal{D}^{\text{train}}_s$, $\forall s \in \mathcal{S}^*$ that comprises all configurations that are not included in the largest training set.

To assess the prediction accuracy of models with scalar predictions, related work computes the *mean absolute percentage error* (MAPE) [8, 9, 13, 32, 39]. This metric relies on the *absolute error* (AE), a measure of the deviation of the predicted scalar performance $\hat{\Pi}$ from the actual observed performance Π. The MAPE is obtained by averaging all absolute errors, divided by the observed performances, across all execution contexts:

$$\text{MAPE}(\mathcal{X}^*) = \frac{100}{|\mathcal{X}^*|} \sum_{x \in \mathcal{X}^*} \left| \frac{\Pi(s, \boldsymbol{c}) - \hat{\Pi}(s, \boldsymbol{c})}{\Pi(s, \boldsymbol{c})} \right|, \quad (s, \boldsymbol{c}) \in \mathcal{X}^* \tag{4}$$

To extend MAPE for models that predict probability distributions, we replace the scalar-based AE with the *continuous ranked probability score* (CRPS) [33]. CRPS generalizes the concept of AE to random variables by comparing the predicted *cumulative probability density function* (CDF), $F_{\tilde{\Pi}(s,\boldsymbol{c})}$, to the ground truth $\Pi(s, \boldsymbol{c})$. As $\Pi(s, \boldsymbol{c})$ is scalar, we express it as a stepwise CDF using the indicator function $\mathbf{1}_{\{\Pi(s,\boldsymbol{c})\}} : \mathbb{R} \mapsto \{0, 1\}$:

$$\mathbf{1}_{\{\Pi(s,\boldsymbol{c})\}}(t) = \begin{cases} 1 & \text{if } t \geq \Pi(s, \boldsymbol{c}), \\ 0 & \text{if } t < \Pi(s, \boldsymbol{c}). \end{cases} \tag{5}$$

CRPS is then computed by integrating the squared difference between the predicted CDF and this step function over all possible performance values v:

$$\text{CRPS}(x) = \int_{-\infty}^{\infty} \left(F_{\tilde{\Pi}(s,\boldsymbol{c})}(v) - \mathbf{1}_{\{\Pi(s,\boldsymbol{c})\}}(v) \right)^2 \mathrm{d}v, \quad (s, \boldsymbol{c}) = x \tag{6}$$

This accounts for the full predictive distribution rather than just a single estimate. To extend MAPE accordingly, we define *probabilistic MAPE* (pMAPE), which replaces AE with CRPS:

$$\text{pMAPE}(\mathcal{X}^*) = \frac{100}{|\mathcal{X}^*|} \sum_{x \in \mathcal{X}^*} \left| \frac{\text{CRPS}(x)}{\Pi(s, \boldsymbol{c})} \right|, \quad (s, \boldsymbol{c}) \in \mathcal{X}^* \tag{7}$$

As pMAPE reduces to MAPE for scalar predictions, we apply it to all models. We repeat the training set generation along with model training 30 times to account for the probabilistic nature of Bayesian inference and random training set sampling. We report the mean pMAPE across all repetitions.

5.1.2 Results. Comparing the pMAPE values of Lasso and Bayesian models in Figure 2, we see that Bayesian models (in blue) consistently outperform Lasso models for a given pooling strategy. As both models share the same model structure, the improvement of Bayesian models can be attributed to our chosen prior distributions, which lead to a better regularization than Lasso.

Table 2 lists the pMAPE for each software system and pooling strategy for Bayesian models. Apparently only for the smallest training set of $1/8|O|$, the no-pooling strategy excels over the others for

Table 2: pMAPE across all subject systems and training set sizes. The best pMAPE for each training set is shaded blue.

$\lvert\mathcal{D}^{\text{train}}\rvert$	$^1/_8\lvert\mathcal{O}\rvert$			$^1/_4\lvert\mathcal{O}\rvert$			$^1/_2\lvert\mathcal{O}\rvert$			$^3/_4\lvert\mathcal{O}\rvert$			$1\lvert\mathcal{O}\rvert$			$2\lvert\mathcal{O}\rvert$			$3\lvert\mathcal{O}\rvert$		
Model	$\tilde{\Pi}^{\text{cp}}$	$\tilde{\Pi}^{\text{np}}$	$\tilde{\Pi}^{\text{pp}}$	$\tilde{\Pi}^{\text{cp}}$	$\tilde{\Pi}^{\text{np}}$	$\tilde{\Pi}^{\text{pp}}$	$\tilde{\Pi}^{\text{cp}}$	$\tilde{\Pi}^{\text{np}}$	$\tilde{\Pi}^{\text{pp}}$	$\tilde{\Pi}^{\text{cp}}$	$\tilde{\Pi}^{\text{np}}$	$\tilde{\Pi}^{\text{pp}}$	$\tilde{\Pi}^{\text{cp}}$	$\tilde{\Pi}^{\text{np}}$	$\tilde{\Pi}^{\text{pp}}$	$\tilde{\Pi}^{\text{cp}}$	$\tilde{\Pi}^{\text{np}}$	$\tilde{\Pi}^{\text{pp}}$	$\tilde{\Pi}^{\text{cp}}$	$\tilde{\Pi}^{\text{np}}$	$\tilde{\Pi}^{\text{pp}}$
H2	14.1	13.8	14.2	10.9	11.7	10.4	9.4	10.4	8.7	8.9	9.4	7.9	8.7	8.5	7.4	8.2	6.9	6.7	8.1	6.6	6.5
VP9	162.2	156.6	132.4	85.8	133.5	58.1	52.7	128.5	37.4	40.4	115.5	31.2	36.5	99.4	29.6	29.9	51.6	27.8	27.5	37.1	29.3
BATIK	39.5	33.4	30.9	22.7	18.5	27.9	15.7	15.9	15.4	14.7	15.2	13.1	14.0	14.0	10.9	13.3	10.9	9.9	12.9	9.6	9.3
DCONV	13.6	13.3	14.5	9.5	10.1	8.8	7.5	8.9	6.4	6.8	8.2	5.5	6.4	7.5	4.8	5.8	5.4	4.1	5.6	4.5	3.9
JUMP3R	53.8	49.9	71.9	41.1	42.3	44.8	23.7	36.7	24.0	19.8	33.2	20.4	17.8	28.3	17.5	16.2	18.3	15.7	15.6	16.0	15.0
LRZIP	283.7	221.8	271.3	189.6	153.8	201.1	170.5	140.6	177.9	162.5	142.4	157.2	149.7	141.7	144.3	134.4	148.0	131.6	132.4	151.1	131.2
x264	99.0	106.2	116.6	60.5	102.3	62.5	51.9	83.6	62.0	51.5	61.7	59.6	50.4	50.4	54.3	49.3	47.7	50.9	49.8	48.6	50.7
x265	88.0	81.8	79.0	53.7	66.1	47.4	42.5	61.2	44.5	40.5	58.2	42.5	40.0	55.1	41.3	39.3	43.8	39.6	39.3	37.2	39.0
XZ	70.8	63.2	74.4	63.2	63.8	69.8	60.0	62.3	70.0	59.9	61.2	67.0	60.2	59.5	64.0	60.5	58.6	62.2	60.7	59.0	61.8
z3	595.4	581.9	587.6	319.9	270.6	307.1	262.0	255.0	252.2	245.2	241.2	208.9	228.2	193.7	167.3	228.9	147.0	152.0	232.1	144.1	151.4
Mean	142.0	132.2	139.3	85.7	87.3	83.8	69.6	80.3	69.8	65.0	74.6	61.3	61.2	65.8	54.1	58.6	53.8	50.0	58.4	51.4	49.8

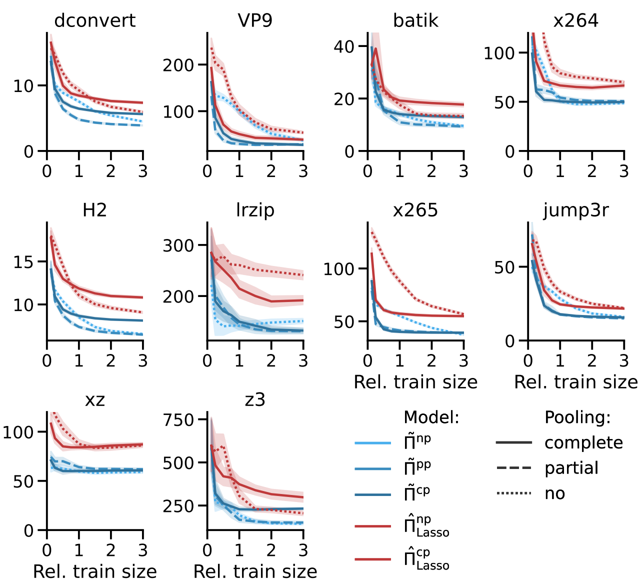


Figure 2: pMAPE results for Bayesian models ($\tilde{\Pi}$) and Lasso ($\hat{\Pi}$). The color encodes the model type, whereas the line style encodes the pooling strategy. The shaded area represents the standard deviation across 30 repetitions.

most subject systems. In absolute numbers, these training sets have one to four training samples per workload. For such small numbers, the standardization in our data preparation (Subsection 3.1) fails to find a common scale for performance values, giving an edge to the no-pooling strategy, where no common scale is needed. For training set sizes of $^1/_4|\mathcal{O}|$ to $1|\mathcal{O}|$, we see a shift towards partial pooling and complete pooling as the best strategies. Because complete pooling merges all setting data, it excels when a system's training data contain many settings and relatively low variation across settings. For a relative sample size of $^3/_4|\mathcal{O}|$, four subject systems can be best modeled with complete pooling, while partial pooling yields best results for six systems. However, given a larger sampling budget, complete pooling loses its edge to no-pooling in cases where the data for each setting contain enough information individually to accurately learn all influences. While Figure 2 reveals substantial accuracy differences between complete and no-pooling models, HyPerf's partial pooling (blue, dashed line) consistently either outperforms the other pooling strategies or is close to the best strategy.

5.1.3 Discussion. Answering $RQ_{1.1}$, HyPerf's partial pooling provides reliable accuracy for datasets larger than $^1/_8|\mathcal{O}|$. Notably, HyPerf achieves strong pMAPE results despite not incorporating interaction terms between options in its current form (see also Section 3.3.3). With ample training data, that is, $|\mathcal{D}| \gg |\mathcal{O}|$, no-pooling may be considered. However, the benefit of investing in additional measurements for a new workload is often uncertain in practice. If the new setting is similar to known ones, partial pooling is advantageous since there is little new to learn. So, while no-pooling can, on average, increase accuracy, it may not be the most economical strategy. In essence, by combining the flexibility of Bayesian multi-level models with enhanced sample efficiency, HyPerf offers a good trade-off for stable and setting-dependent influences.

5.2 Predictive Accuracy ($RQ_{1.2}$)

5.2.1 Operationalization. HyPerf 's novel capabilities are no free lunch [48] and may come at a cost in predictive accuracy. To assess this trade-off, we compare HyPerf to three state-of-the-art performance-prediction approaches that prioritize predictive power over interpretability. We include Random Forest (RF) [3] as a strong ensemble learner known for capturing complex, nonlinear interactions between configuration options [20]. Specifically, we use `RandomForestRegressor` from the SCIKIT-LEARN library with 100 trees, squared-error splits, and bootstrap sampling. We further include DeepPerf [13], a deep-learning approach that automatically composes and tunes feed-forward networks under ℓ_1-regularization, balancing model expressiveness and sparsity. We adopt DeepPerf's reference implementation, perform 50 hyperparameter optimization trials, and enable early stopping after 20 epochs without improvement. Lastly, we include Divide-and-Learn (DaL) [10], a recent approach that partitions the configuration space into diverse regions and fits a sparsity-regularized deep model to each region individually. Each prediction is computed by routing a configuration to its closest local model. We follow the authors' implementation. We evaluate each method with both no-pooling and complete-pooling (i.e., one model per setting and a single model shared across all settings). We replicate the experiments of RQ_1 for each pooling variant. To keep model training tractable, given the

computational cost of these complex learners, we reduce the number of repetitions and restrict training to smaller sets of size $\alpha \cdot |O|$ with $\alpha \in \{1/8, 1/4, 1/2, 1, 2\}$. Despite this reduced training budget, the setup remains sufficient to reveal differences in accuracy.

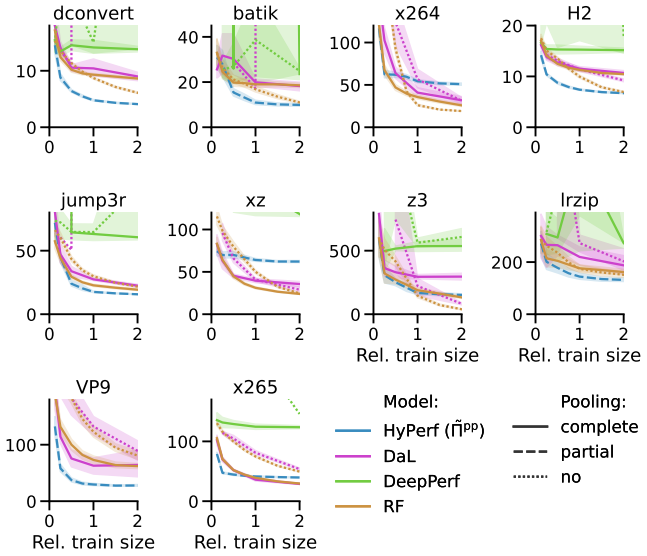


Figure 3: pMAPE results for HyPerf, Random Forest, DeepPerf and DaL. The color encodes the model type, whereas the line style encodes the pooling strategy. The shaded area represents the standard deviation across 5 repetitions.

5.2.2 Results. Figure 3 presents the prediction accuracy of all models across varying training sizes and subject systems. HyPerf consistently achieves a low pMAPE, especially with small training sizes ($\alpha \leq 1$), often outperforming competing approaches. Its partial pooling enables it to leverage shared information across workloads, yielding robust predictions even under sparse data. Exceptions include x264 and z3, where higher variability limits this benefit. In contrast, DaL and DeepPerf exhibit large variance in low-data regimes and occasionally fail to train completely, causing visible irregularities in Figure 3. Pooling trends reveal additional insights. For DaL and DeepPerf, complete pooling (solid lines) often performs competitively across training sizes in systems such as VP9, sws, and xz, suggesting that option influences in these systems are largely stable across workloads. In settings with more diverse workload-specific behavior, such as H2, x264, and z3, no-pooling (dotted lines) tends to outperform complete pooling when $\alpha > 1/2$. Across all training sizes, HyPerf's partial pooling (dashed lines) reliably balances sample efficiency and flexibility, adapting to both high and low variance in option influences across settings.

5.2.3 Discussion. Overall, HyPerf achieves strong and consistent prediction accuracies across subject systems, especially when training data are limited. In these cases, it often outperforms state-of-the-art single-point estimation approaches such as DeepPerf and DaL, which are prone to unreliable results or complete training failures. While both models can achieve lower pMAPE with large training sets, particularly when option influences are stable across settings, they are substantially more expensive to train (up to 50 times for DeepPerf compared to HyPerf).

5.3 Scalability ($RQ_{1.3}$)

5.3.1 Operationalization. To investigate whether HyPerf remains operable and insightful in extreme-scale scenarios, we apply it to the *TuxKConfig* dataset [2], which captures the Linux kernel's configuration space across seven versions (4.13–5.8), interpreted as settings in our multi-level model. Each version contains over 12 000 binary options, with a total of more than 243 000 measured configurations. We model the kernel's binary size.

To consider realistic constraints, we train HyPerf with a minimal budget of $\alpha = 0.01$ (i.e., 120 samples per version). We retain 500 MCMC samples after 500 warm-up iterations—a pragmatic compromise to ensure feasibility while enabling posterior analysis. Due to the large data size, we subsample a test set of 10 000 configurations and skip multicollinearity reduction since the dataset is already preprocessed. Missing values (e.g., from version-specific constraints) are set to zero, treating unavailable options as inactive. We report training time, prediction time, and memory usage.

5.3.2 Results and Discussion. Training took approximately 5 hours and used up to 70 GB of memory. Predicting 10 000 configurations completed in under two minutes. These results suggest that HyPerf remains operational on large-scale systems, with manageable training cost and fast prediction. Users can further adjust the cost of inference by tuning the number of MCMC samples. Compared to the effort of compiling and measuring hundreds of kernels, the computational cost of training remains modest. More importantly, HyPerf's Bayesian inference provides insights into model fitting. The Pareto-smoothed leave-one-out diagnostic (p_{loo}) [43] revealed that few parameters were effectively learned—indicating that priors dominated due to limited data per option. With thousands of options, understanding whether generalization fails due to data scarcity or model assumptions becomes challenging—a distinction HyPerf's Bayesian diagnostics help make.

5.4 Reasoning on Multi-Factorial Variance (RQ_2)

5.4.1 Operationalization. To study HyPerf's capabilities for capturing the variation in the performance influence of options across settings, we train and analyze one HyPerf model for each software system, trained on its largest $3|O|$ dataset to minimize uncertainty among the inferred performance influences. The multi-level structure of these models enable novel quantitative and qualitative analyses, which we discuss next.

Upper Level. HyPerf's upper level enables analyses on the *general* influences of options, that is, on the performance influence patterns that are consistent across settings. This, however, requires a prior step, because, after Bayesian inference updates an option's hyper prior into its hyper posterior, each option's general influence is split into its hyper posterior for the expected value $\boldsymbol{\beta}_{\mu,o}|\mathcal{D}$ and its spread $\boldsymbol{\beta}_{\sigma^2,o}|\mathcal{D}$. By inserting both into Equation 3f for HyPerf's specific influences, we compute $\boldsymbol{\beta}_o^{\text{gen}}$, an option's general influence across settings. In essence, it reflects HyPerf's updated best guess for an option's influence for any unseen setting.

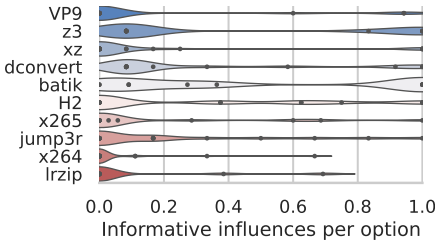


Figure 4: Proportion of configuration options whose performance influence varies substantially across workloads in different software systems.

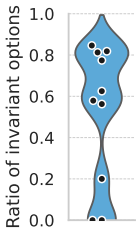


Figure 5: Share of options with invariant influence per system.

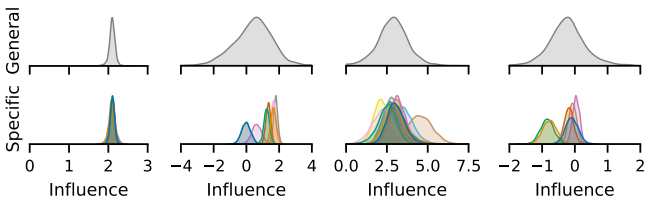


Figure 6: Comparing general influences versus their workload-specific ones for selected edge cases.

A wider spread in β_o^{gen} indicates more variance in setting-specific influences ($\beta_o|\mathcal{D}$). To quantify the spread of a given general influence, we compute its 95%-*credible interval* (CI) and that interval's width. This allows us to determine whether there is uniform variance in the general influences or whether specific options exhibit significantly wider spreads. We use *inter quartile range* outlier detection [15] to identify options O_{out} for which the hyper posterior spread substantially deviates from the norm. These outliers indicate options with an unusual level of variance in their performance influence, requiring special attention in new settings. This method adapts to a model's overall magnitude of spread in its β_o^{gen}, which can vary significantly across models.

Multi-level. The presence of both general and specific influences in the same model allows us to identify workload-specific influences that substantially deviate from their general influence. These unique workloads are particularly insightful, hinting at new use cases of the software system or performance bugs that manifest only for certain workloads. However, HyPerf's robust hyper priors may treat such deviations as outliers, potentially not reflecting them in the general influences and being missed through the upper-level analysis. From an information-theoretic perspective, specific influences add information to the model if they deviate from their general influence. The *Kullback-Leibler Divergence* (KLD) [26] quantifies this information gain[3]. Consequently, we compute the KLD for each pair of general and corresponding specific influence. The higher the KLD, the more information that specific influence contributes to the model. Consequently, we deem specific influences as *informative* if their KLD is above a threshold of 1.0, which has proved robust in our experiments. Finally, we consider an option's performance influence *invariant* if no specific influence is informative.

5.4.2 Results. Overall, our multi-level analysis reveals that informative influences, i.e., workload-specific influences that deviate from general influence, are not evenly distributed across options. Notably, 57 % of options exhibit no informative workload-specific influence, suggesting their effect is stable across all workloads. An example is x264's option noasm, which shows identical general and specific influences in Figure 6a. In contrast, 10 % of options exhibit informative influences under every workload, indicating that no consistent general influence can be identified.

[3]As a closed-form posterior PDF is unavailable, we approximate it using sampling [35].

Figure 4 shows how informative influences are distributed across options and systems. Each point in the violin plot represents the proportion of informative setting-specific influences for one option. Notably, a large number of options has 5 to 20 % informative influences. Such options have a common influence among the majority of workloads, but there are notable workload-specific deviations. The violin plot in Figure 5 visualizes these option influences across all subject systems. Here, a point represents a software system, which is positioned according to the relative number of options with invariant influence. Counting the points on the upper half of the plot shows that more than 50 % of options have an invariant performance influence for seven subject systems, with x264 exhibiting the highest share of options with invariant influence of 88%. At the same time, Figure 5 reveals a cluster consisting of batik, Density Converter, z3, among whose options more than 90 % have informative workload-specific influences.

Our investigation at the upper model level has identified ten general-influence outliers across several subject systems: xz, Density Converter, jump3r, x264, lrzip, and H2. The remaining subject systems exhibited no outliers, suggesting uniform variance in their option's influences across workloads. To gain deeper insights, we conducted a qualitative analysis on the five options with the largest credible intervals, investigating whether their variance roots in their functional dependence on the workload: H2's option MVSTORE enhances performance especially for concurrent query workloads, leading to distinct specific influences shown in Figure 6b. The lrzip option max-resident-set-size limits the available RAM during run-time, directly influencing performance relative to the workload size. This leads to a general influence with an exceptional CI width of 4.6, depicted in Figure 6c. Notably, the same option was identified as an outlier for x264. Variance in the general influence of option threads (Density Converter) suggests that increasing the number of threads does not benefit all workloads. Option RECOMPILE_ALWAYS of H2 affects workloads with repetitive queries more severely by preventing caching. Lastly, the influence of Mono (jump3r) depends on whether the workload is stereo or mono. The clustered specific influences seen in Figure 6d reflect this behavior. All this indicates that the outliers detected with HyPerf are reasoned by the functionalities of these options.

5.4.3 Discussion. Our findings highlight the novel insights that HyPerf is able to provide and emphasize the importance of considering influence variance across settings. Answering RQ_2, our results show that, depending on the system, 0% to 88% of options'

influences are invariant across workloads. For options with varying influences, we frequently observe one of two patterns: either each workload exhibits a distinct influence, or most workloads share a common influence with only a few causing deviations. This insight enables focused optimization efforts by identifying which options need workload-specific testing versus those that can be configured uniformly. We furthermore observe that it is common for a software system to have an option whose influence varies substantially more compared to other options. Our results confirm previous studies [29, 34], while providing finer granularity and qualitative insights. In particular, they demonstrate HyPerf's reasoning capability by detecting when a setting actually varies an option's influence. This capability stems from HyPerf's method to model distinct sources of uncertainty in its information loss quantification.

5.5 Coverage Sets of Workloads

During performance testing configurable software systems, detecting workload-specific performance issues is crucial, yet testing every known workload is impractical, as changes in other factors, such as hardware, quickly lead to a combinatorial explosion. To make testing practical, we need a systematic method for identifying a minimal *coverage* set of workloads $\mathcal{S}^\circ \subset \mathcal{S}^*$ that captures essential performance variations across options' performance influences, creating a tailor-made benchmark suite. While non-Bayesian methods can compute only distances between specific influences to judge whether they provide sufficient coverage of each other, HyPerf's inferred PDFs enable a more informative comparison by capturing uncertainty from measurement noise and data sparsity. When comparing PDFs for coverage assessment, we consider the asymmetric nature of the relationship: A workload whose performance influence has a broader distribution naturally provides better coverage of behaviors captured by several workloads with narrower distributions, while the opposite would miss important performance variations. A key challenge arises when performance influences of options lack a common mode [19, 29, 34]. Such options would require considering all workloads in $\mathcal{S}^*$ for complete coverage, which would defy the purpose of coverage. Thus, selecting workloads demands a careful balance between coverage completeness and practicality.

5.5.1 Operationalization. To answer RQ_3, we compute the information loss when using one workload-specific influence to cover the influence under another workload. While symmetric measures such as the Jensen-Shannon divergence [31] exist, we specifically chose KLD because its asymmetry aligns with the directionality of coverage relationships. To find a minimal coverage set, we first compute the KLD for each pair of specific influences bidirectionally across all options, forming information loss matrices, such as the one for Mono (JUMP3R) depicted in Figure 7. The ith column in this matrix quantifies the information loss when using the ith workload to cover the performance influence of the given option instead of the jth workload, as indicated in the jth row. Each cell represents a coverage candidate. Such a candidate provides coverage if its KLD is low enough. In RQ_2, a KLD threshold of 1.0 was selected to classify influences as informative (i.e., not being covered by its general influence). However, coverage between workloads requires a stricter criterion. Our experiments suggest that a KLD threshold of 0.5 effectively distinguishes sufficient coverage, although practitioners may adapt this based on their requirements. Coverage candidates above this threshold appear red in Figure 7. For instance, while Figure 6d suggests two clusters of specific influences for JUMP3R's option Mono, its information loss matrix in Figure 7 indicates that, in fact, a third workload is needed for complete coverage. However, it is unclear whether these three workloads suffice to cover specific influences for *all* options. To build the *complete* coverage set $\mathcal{S}^\bullet$, we iteratively add the workload that reduces the cumulative information loss the most until all specific influences are covered. Once this reduction falls below 10% of the information loss of the initial workload $\mathcal{S}^\bullet_1$, we mark the set of the previous iteration as our *practical* coverage set $\mathcal{S}^\circ$. Thus, the practical coverage set $\mathcal{S}^\circ$ aims to provide sufficient coverage of workload-specific influences at minimal size, whereas the complete coverage set $\mathcal{S}^\bullet$ includes all workloads required to cover every specific influence. Our termination criterion adapts to the scale of information loss in a given model and prevents unnecessarily large coverage sets. Our companion website contains the full algorithm. As a baseline for evaluating the practical ($\mathcal{S}^\circ$) and complete ($\mathcal{S}^\bullet$) coverage sets, we compute the information loss for each individual workload.

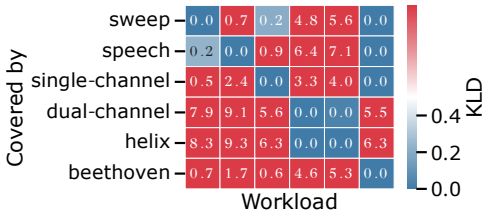


Figure 7: Representation matrix for JUMP3R's Mono option, comparing how each workload-specific influence (in rows) covers the others (in columns) using KLD. Blue cells (KLD < 0.5) indicate sufficient coverage.

5.5.2 Results. Our analysis reveals substantial improvements in reducing information loss through information-loss-based workload selection. Compared to other workloads, our initially computed set of workloads reduce information loss by, on average, 31%, in some cases even halving it. While these results witness the effectiveness of choosing a bellwether workload (i.e., a single workload that provides the best coverage), our results in Table 3 show that adding a second workload decreases the information loss compared to the initial workload by 52%, on average. For instance, the red swarm plot in Figure 8 illustrates that other workloads yield 282 % higher information losses compared to the optimal first workload for x265, while the second workload $\mathcal{S}^\circ_2$ cuts the best initial loss by 50%. Most subject systems require only three workloads for practical coverage, with BATIK requiring four, and JUMP3R and x264 only two. On average, these coverage sets achieve an information loss reduction of 77 %, compared to the best initial workload, and 84 % compared to $\overline{\mathcal{S}^*}$, the average across all considered workloads. In Figure 8, the dashed line marks the coverage set size of three workloads until the information loss (blue line) substantially declines in each iteration (blue dots). The resulting coverage sets cover nearly all workload-specific influences for all options, with the number of

influences with un-covered specific influences never exceeding 4. This suggests that HyPerf's information-loss quantification reliably excludes options with no common influence across workloads.

Table 3: Coverage set characteristics, listing average information loss ($\overline{\mathcal{S}^*}$), initial loss ($\mathcal{S}_1^\circ$), improvement with second workload ($\mathcal{S}_{1\mapsto 2}^\circ$), final loss ($\mathcal{S}^\circ$), sizes of complete ($|\mathcal{S}^\bullet|$) and practical coverage set ($|\mathcal{S}^\circ|$), and the number of options with un-covered specific influences ($|O_{\text{unc}}|$).

System	Information loss with				Set size		$\lvert O_{\text{unc}}\rvert$
	$\overline{\mathcal{S}^*}$	$\mathcal{S}_1^\circ$	$\mathcal{S}_{1\mapsto 2}^\circ$	$\mathcal{S}^\circ$	$\lvert\mathcal{S}^\circ\rvert$	$\lvert\mathcal{S}^\bullet\rvert$	
JUMP3R	47	29	82%	4	2	3	2
DCONV	207	154	79%	28	2	6	4
H2	89	54	54%	11	3	6	4
BATIK	219	193	55%	31	4	7	4
XZ	154	106	47%	12	3	6	3
LRZIP	36	16	37%	3	3	4	2
X264	51	28	44%	10	2	3	1
Z3	339	257	44%	78	3	10	4
VP9	286	157	44%	44	3	9	3
X265	257	162	36%	39	3	8	4
Avg.	168.6	115.9	52.2%	26.3	2.8	6.2	3.1

5.5.3 Discussion. Our results for RQ_3 demonstrate that HyPerf is indeed able to determine minimal workload sets that provide adequate coverage of performance variations across configuration options, achieved through systematic quantification of information loss. The practical impact is significant: Identifying workloads that provide essential coverage and workloads that add minimal new information is crucial for efficient (regression) performance testing. Our computed savings in execution time of workloads, when comparing 3 vs. up to 35 workloads, is substantial. Considering 6–35 workloads per system at once through HyPerf's multi-task design already enables a more robust analysis compared to prior approaches. When a new workload emerges, HyPerf allows practitioners to assess the information gain from measuring a small sample and even to understand which option's influences deviate from the existing test suite. Our results challenge the assumption that a single bellwether setting suffices [24]. Adding a second workload already halves information loss, indicating important uncovered performance influences. Nevertheless, we encourage practitioners to adapt both the selection procedure and the KLD threshold based on their specific coverage requirements and resource constraints.

5.6 Threats to Validity

We address threats to *internal validity* arising from measurement noise in our training data by repeating each performance experiment, at least, five times, using the median. Each performance experiment was run on a dedicated compute node with a minimum of background processes and only necessary packages installed.

Increasing *external validity*, we study ten diverse, previously studied subject systems that were selected from different domains to ensure broad applicability. We focus on workload variability as our external factor, because it strongly influences performance, it can be varied without affecting other factors, and is easier to modify to aid reproducibility. This choice enables direct comparison with prior work [18, 19, 28, 30, 34]. As HyPerf is agnostic to the specifics of factors, its demonstrated capabilities hold for other qualitative setting changes, although effectiveness may vary.

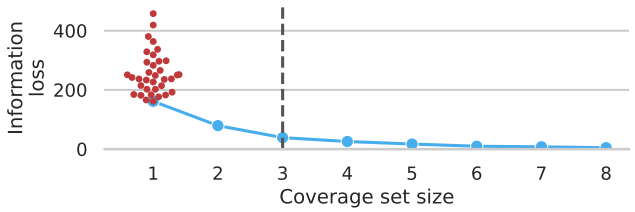


Figure 8: Coverage set building for x265. Red shows information losses of individual workloads; blue shows reduction by HyPerf. Dashed line indicates practical set size.

A potential threat to *construct validity* lies in the choice of options and workloads. Despite our dataset's breadth, some options may have limited performance impact, and certain workloads may exhibit similar behavior, simplifying the modeling task.

6 Conclusion

Software configurability allows users to tailor systems to different settings, yet existing approaches of performance-influence modeling largely ignore external factors such as workload variability, limiting their practical effectiveness. This is where our approach comes into play: With HyPerf, a *Bayesian multi-level performance modeling approach*, we propose to employ a hierarchical model structure to model common performance influences of configuration options on the upper level and setting-specific influences at the lower level of the model. This ensures sample efficiency by generalizing across settings while preserving their specific performance characteristics, which is not possible with existing approaches.

Tested on ten real-world systems across up to 35 workloads, HyPerf consistently matches or outperforms state-of-the-art methods (Lasso, Random Forest, DeepPerf, DaL) in predictive accuracy while reducing training time by up to a factor of 50. HyPerf remains computationally feasible even on very large systems—with thousands of options and multiple settings—training in hours and predicting in minutes, while its Bayesian diagnostics flag when adaptive priors are needed under extreme sparsity. Our multi-level analysis reveals that more than half of configuration options exhibit invariant influences, while 10 % share no common influence across workloads, enabling focused optimization efforts. Crucially, HyPerf is able to compute coverage workload sets, cutting required performance tests from as high as 35 to 3 while still capturing 77 % of performance variance. This and the ability to quantify information loss in workload coverage provides a robust method to create efficient benchmark suites—a significant advancement in performance engineering of configurable software systems.

Acknowledgments

This work has been supported by the German Research Foundation (DFG) under Grants SI 2171/3-2 and AP 206/11-2 and through Collaborative Research Center TRR 248 – CPEC (389792660), by the German Federal Ministry of Education and Research (BMBF, SCADS22B), as well as by the Saxon State Ministry for Science, Culture and Tourism (SMWK) for ScaDS.AI.

References

[1] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6.

[2] Heraldo Borges, Juliana Alves Pereira, Djamel Eddine Khelladi, and Mathieu Acher. 2025. Linux Kernel Configurations at Scale: A Dataset for Performance and Evolution Analysis. *arXiv preprint arXiv:2505.07487* (2025).

[3] Leo Breiman. 1996. Bagging Predictors. *Machine Learning* 24 (1996), 123–140.

[4] Jiezhu Cheng, Cuiyun Gao, and Zibin Zheng. 2022. HINNPerf: Hierarchical Interaction Neural Network for Performance Prediction of Configurable Systems. *ACM Transactions on Software Engineering and Methodology* 32, 2, Article 46 (2022), 30 pages.

[5] Jamal I Daoud. 2017. Multicollinearity and Regression Analysis. In *Journal of Physics: Conference Series*, Vol. 949. IOP Publishing, 012009.

[6] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 379–390.

[7] Yi Ding, Yi Ding, Ahsan Pervaiz, Michael Carbin, and Henry Hoffmann. 2021. Generalizable and Interpretable Learning for Configuration Extrapolation. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 728–740.

[8] Johannes Dorn, Sven Apel, and Norbert Siegmund. 2023. Mastering Uncertainty in Performance Estimations of Configurable Software Systems. *Empirical Software Engineering* 28, 2 (2023), 33.

[9] Daniel Friesel and Olaf Spinczyk. 2022. 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)*. ACM, 170–180.

[10] Jingzhi Gong, Tao Chen, and Rami Bahsoon. 2025. Dividable Configuration Performance Learning. *IEEE Trans. Softw. Eng.* 51, 1 (Jan. 2025), 106–134. https://doi.org/10.1109/TSE.2024.3491945

[11] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-Aware Performance Prediction: A Statistical Learning Approach. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311.

[12] 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 (2018), 1826–1867.

[13] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1095–1106.

[14] Huong Ha and Hongyu Zhang. 2019. Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 470–480.

[15] Victoria Hodge and Jim Austin. 2004. A Survey of Outlier Detection Methodologies. *Artificial Intelligence Review* 22, 2 (2004), 85–126. https://doi.org/10.1023/B:AIRE.0000045502.10941.a9

[16] Matthew D. Hoffman and Andrew Gelman. 2014. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *The Journal of Machine Learning Research (JMLR)* 15 (2014), 1593–1623.

[17] 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 International Conference on Automated Software Engineering (ASE)*. IEEE, 497–508.

[18] 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 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 71–82.

[19] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. 2017. Transfer Learning for Improving Model Predictions in Highly Configurable Software. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 31–41.

[20] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2020. The Interplay of Sampling and Machine Learning for Software Performance Prediction. *IEEE Software* 37, 4 (2020), 58–66. https://doi.org/10.1109/MS.2020.2987024

[21] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-Based Sampling of Software Configuration Spaces. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1084–1094.

[22] Ugur Koc, Austin Mordahl, Shiyi Wei, Jeffrey S. Foster, and Adam A. Porter. 2021. SATune: A Study-Driven Auto-Tuning Approach for Configurable Software Verification Tools. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 330–342.

[23] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2019. Tradeoffs in Modeling Performance of Highly Configurable Software Systems. *Software and Systems Modeling* 18, 3 (jun 2019), 2265–2283.

[24] Rahul Krishna and Tim Menzies. 2019. Bellwethers: A Baseline Method for Transfer Learning. *Transactions on Software Engineering* 45, 11 (2019), 1081–1105.

[25] Rahul Krishna, Vivek Nair, Pooyan Jamshidi, and Tim Menzies. 2021. Whence to Learn? Transferring Knowledge in Configurable Systems Using BEETLE. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2956–2972.

[26] S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79–86.

[27] Ravin Kumar, Colin Carroll, Ari Hartikainen, and Osvaldo Martin. 2019. ArviZ a unified library for exploratory analysis of Bayesian models in Python. *Journal of Open Source Software* 4, 33 (2019), 1143. https://doi.org/10.21105/joss.01143

[28] Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. Deep Software Variability: Towards Handling Cross-Layer Configuration. In *Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, 1–8.

[29] Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. 2023. Input Sensitivity on the Performance of Configurable Systems: An Empirical Study. *Journal of Systems and Software* 2021 (2023), 111671.

[30] Luc Lesoil, Helge Spieker, Arnaud Gotlieb, Mathieu Acher, Paul Temple, Arnaud Blouin, and Jean-Marc Jézéquel. 2024. Learning Input-Aware Performance Models of Configurable Systems: An Empirical Evaluation. *Journal of Systems and Software* 208 (2024), 111883.

[31] Jianhua Lin. 1991. Divergence measures based on the Shannon entropy. *IEEE Transactions on Information theory* 37, 1 (1991), 145–151.

[32] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Jean-Marc Jézéquel, and Djamel Eddine Khelladi. 2021. Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size. *Transactions on Software Engineering* 48, 11 (2021), 4274–4290.

[33] James E. Matheson and Robert L. Winkler. 1976. Scoring Rules for Continuous Probability Distributions. *Management Science* 22 (1976), 1087–1096.

[34] Stefan Mühlbauer, Florian Sattler, Christian Kaltenecker, Johannes Dorn, Sven Apel, and Norbert Siegmund. 2023. Analyzing the Impact of Workloads on Modeling the Performance of Configurable Software Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2085–2097.

[35] Fernando Perez-Cruz. 2008. Kullback-Leibler Divergence Estimation of Continuous Distributions. In *2008 IEEE International Symposium on Information Theory*. IEEE, 1666–1670.

[36] Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. *arXiv preprint arXiv:1912.11554* (2019).

[37] Larissa Schmid, Marcin Copik, Alexandru Calotoiu, Dominik Werle, Andreas Reiter, Michael Selzer, Anne Koziolek, and Torsten Hoefler. 2022. Performance-detective: automatic deduction of cheap and accurate performance models. In *Proceedings of the 36th ACM International Conference on Supercomputing*. ACM, Article 3.

[38] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 284–294. https://doi.org/10.1145/2786805.2786845

[39] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines. *Software Quality Journal* 20 (2012), 487–517.

[40] Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. 2018. VarXplorer: Lightweight Process for Dynamic Analysis of Feature Interactions. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018, Madrid, Spain, February 7-9, 2018*. ACM, 59–66.

[41] Robert Tibshirani. 1996. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* 58 (1996), 267–288.

[42] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. 2017. Transferring Performance Prediction Models Across Different Hardware Platforms. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 39–50.

[43] Aki Vehtari, Andrew Gelman, and Jonah Gabry. 2017. Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC. *Statistics and Computing* 27, 5 (2017), 1413–1432. https://doi.org/10.1007/s11222-016-9696-4

[44] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. 2020. Configcrusher: Towards White-Box Performance Analysis for Configurable Systems. *Automated Software Engineering* 27, 3 (2020), 265–300.

[45] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 18:1–18:33.

[46] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 213–226.

[47] Max Weber, Sven Apel, and Norbert Siegmund. 2021. White-Box Performance-Influence Models: A Profiling and Learning Approach. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1059–1071.

[48] David H. Wolpert. 1996. The Lack of A Priori Distinctions Between Learning Algorithms. *Neural Computation* 8, 7 (10 1996), 1341–1390.

[49] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance Prediction of Configurable Software Systems by Fourier Learning. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 365–373.