Performance-Detective: Automatic Deduction of Cheap and Accurate Performance Models

Larissa Schmid
Karlsruhe Institute of Technology, KASTEL
Germany

Marcin Copik
ETH Zurich, D-INFK
Switzerland

Alexandru Calotoiu
ETH Zurich, D-INFK
Switzerland

Dominik Werle
Karlsruhe Institute of Technology, KASTEL
Germany

Andreas Reiter
University of Applied Sciences Karlsruhe, IDM
Germany

Michael Selzer
Karlsruhe Institute of Technology, IAM-MMS
Germany

Anne Koziolek
Karlsruhe Institute of Technology, KASTEL
Germany

Torsten Hoefler
ETH Zurich, D-INFK
Switzerland

ABSTRACT

The many configuration options of modern applications make it difficult for users to select a performance-optimal configuration. Performance models help users in understanding system performance and choosing a fast configuration. Existing performance modeling approaches for applications and configurable systems either require a full-factorial experiment design or a sampling design based on heuristics. This results in high costs for achieving accurate models. Furthermore, they require repeated execution of experiments to account for measurement noise. We propose *Performance-Detective*, a novel code analysis tool that deduces insights on the interactions of program parameters. We use the insights to derive the smallest necessary experiment design and avoiding repetitions of measurements when possible, significantly lowering the cost of performance modeling. We evaluate *Performance-Detective* using two case studies where we reduce the number of measurements from up to 3125 to only 25, decreasing cost to only 2.9% of the previously needed core hours, while maintaining accuracy of the resulting model with 91.5% compared to 93.8% using all 3125 measurements.

ICS '22, June 28–30, 2022, Virtual Event, USA

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

automatic performance modeling, empirical performance modeling, experiment design, configurable systems

ACM Reference Format:
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 *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3524059.3532391

1 INTRODUCTION

The costs of operating high-performance systems are millions of euros per year [8, 10]. While computing centers do their best to design and run clusters economically, developers must ensure that their applications scale efficiently on computing clusters. Modern software systems are configurable and allow users to set many parameters according to their needs. For example, in the Pace3D material simulation [21], users select algorithm settings and properties to simulate, impacting performance metrics such as response time and throughput [40, 50]. Choosing a set of parameters that yields the best performance is challenging. Developers and users often do not know how configuration options interact or how a single configuration option influences performance [19, 39]. An example of such a challenge is understanding application

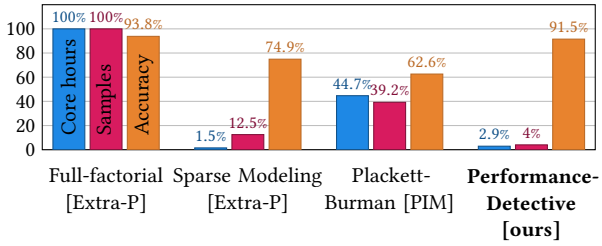


Figure 1: *Performance-Detective* vs Extra-P [12] and Performance-Influence Models (PIM) [39]. Cost-accuracy trade-offs on Pace3D with three parameters.

scalability, i.e., the interaction between problem size and the number of processes [4].

Performance modeling helps users understand application behavior by expressing application performance as functions of input parameters [20, 25]. Since the advent of High-Performance Computing (HPC), performance experts manually identify and model the parts of the application they consider critical to performance. Due to the high cost of performance experts, HPC developers often intuitively select a subset of the configuration space for evaluation. Automatic performance modeling generates models from empirical measurements that cover all configuration options. Even though configuration spaces are constrained and smaller than combinatorial explosion would suggest [34], the number of options still makes exhaustive measurements infeasible.

Modeling frameworks for applications with many configuration parameters can be classified into black-box and white-box approaches. The former suffers from high sampling costs and relies on heuristics to reduce the experiment design [39], introducing the risk of excluding interactions between options from the sample set. The likelihood of this problem can be decreased by using more samples – leading to a hard to quantify trade-off between model accuracy and the number of samples [17, 27, 36]. White-box approaches can support modeling of numerical options but do not use known interactions between options, resulting in an expensive full-factorial experiment design [13] or using heuristics to reduce the number of samples [48]. Other white-box procedures concentrate on binary and binary-encoded options [45, 46] and do not consider numerical parameters such as the problem size or the number of processes.

In this paper, we introduce *Performance-Detective*, a novel white-box modeling methodology that significantly lowers experimentation costs while maintaining the accuracy of the resulting models (Figure 1). In contrast to previous sampling optimizations that used imprecise heuristics, we use program information to *deduce* an optimized, minimal experiment design, removing measurement points that do not affect known interactions between non-functional parameters. We use *parametric performance models* obtained from the taint-based analysis [13] to understand the impact of parameters on program functions. Through a step-by-step analysis of parameter interactions, we derive conclusions on parameter interactions in the program's control flow. Applying the deduced conclusions to the experiment design removes measuring points unnecessary to model parameter interactions. Thus, *Performance-Detective* can reduce the dimensionality of experiments from exponential to polynomial while avoiding the risk of excluding parameter dependencies from the experiment. The experiment design of *Performance-Detective* is orthogonal to the modeling approach and can be used with black-box and white-box performance modeling.

The *deduced* experiment design makes performance modeling more affordable and is easily applicable alongside modern performance modeling systems. To quantify the increased efficiency and validate model correctness, we empirically evaluate *Performance-Detective* using a multi-physics solver (see Figure 1) and a particle transport application. *Performance-Detective* maintains accuracy of 91.4% while reducing the costs of measurements by a factor of up to 34 times, compared against both the Extra-P empirical performance modeling tool [12, 36] and Performance-Influence Models [39].

Contributions:

- *Performance-Detective*, a white-box measurement methodology using a novel *deductive* analysis that uses the results of performance tainting to derive an optimized minimal subset of required measurements out of a multi-dimensional configuration space[1].
- An extensive evaluation against state-of-the-art modeling workflows, proving high accuracy and reduced experiment size.
- We identify main loops within applications using our deductive analysis, and leverage this to reduce the cost of experiments even further, in essence applying classical analytical performance modeling techniques to modern automatic modeling approaches.
- Two case studies of modeling and evaluating applications extrapolated and interpolated test points.

2 FOUNDATIONS

The deduction process of *Performance-Detective* is based on a program's parametric profile obtained with taint-based performance modeling provided by Perf-Taint (Sec. 2.1). *Performance-Detective* overcomes the limitations of existing performance modeling tools (Sec. 2.2), enabling efficient and reliable white-box modeling.

2.1 Perf-Taint

Perf-Taint [13, 14] is a hybrid modeling tool that enhances the black-box and analytical modeling tools with program

[1]Workflow prototype and replication package: https://doi.org/10.5445/IR/1000146001

Figure 2: Perf-Taint: applying the taint-based analysis produces a JSON-like parametric performance profile.

information. Perf-Taint applies a sequence of static and dynamic analyses to the program to understand how its computational effort is affected by a change in input parameters. The result of the operation is a *parametric profile* of a program, consisting of a list of performance-relevant functions and parameters whose values can change the performance. Perf-Taint is built on top of LLVM [29] and supports distributed parallel applications implemented in C/C++ with MPI. Internally, the tool uses static LLVM loop analysis [16] and DFSan [15], a dataflow taint analysis library extended with support for control-flow tainting [49].

We present Perf-Taint on an example program with two functions in Figure 2. The function *important* is performance-relevant because its complexity changes with parameter x. Thus, both the parameter and the function are included in the performance profile on the right side. On the other hand, the function *unimportant* involves only a constant amount of computation. This information obtained from program analysis improves the overall modeling workflow by restricting the modeling to performance-relevant program elements In practice, many short-running functions are particularly affected by noise and could otherwise generate false models that would make the modeling process more difficult and error-prone. These can be automatically excluded from the modeling process if Perf-Taint identified them as constant.

2.2 Limitations of modeling frameworks

The state-of-the-art modeling approaches do not allow for efficient modeling of computing applications with many parameters and numerical options.

Performance-Influence Models (PIMs) [48] derive performance models on the function level, but they do not use parameter dependencies and require expensive tracing instead of profiling. White-Box PIMs [45, 46] do not support numerical options. Though numerical options can be encoded and discretized [46], the approach does not use a learning method and, therefore, cannot predict inter- or extrapolating training configurations.

Extra-P [12] relies on the *full-factorial* experiment design where all parameter combinations must be considered, leading to a combinatorial explosion of the number of measurements. Heuristics have been proposed to optimize the experiment design [36], but they introduce a risk of missing parameter interactions and negatively affecting model quality. Furthermore, the impact of the noise present in measurements makes modeling with more than three parameters particularly challenging.

Perf-Taint [13] does not propose a deterministic and effective methodology for user analysis of the experiment design. While Perf-Taint detects no function depending on both x1 and x2 in the example in the listing below, it does not use parameter knowledge to reduce the sampling set.

```
void f(int x) {
  for(int i : irange(0, x))
    calculate();
}
void g(int x) {
  for(int i : irange(0, x))
    calculate();
}
f(x1); g(x2);
```

In the next code example, all functions are called in the main loop. If the main loop is executed often enough, repetitions of experiments are not necessary as the repetition of the calculations already accounts for measurement noise.

```
for (int i = 0; i < iters; i++) {
  f(x1); g(x2);
}
```

However, Perf-Taint does not make use of this to suggest fewer experiments. Even though 25 and 5 measurements are sufficient to model the performance in these examples, Extra-P and Perf-Taint still require and suggest 125 measurements in both cases. Even though heuristics for sampling can lower experiment size, they do so at the cost of reducing model accuracy. While more samples can increase the accuracy, there is no strategy for selecting them, leading to a full-factorial experiment design.

3 APPROACH

Figure 3 depicts the overall workflow of performance modeling that we will detail in the following subsections. We start by analyzing the system to trace the influence of configuration options on single functions (Section 3.1). *Performance-Detective* then uses the insights about which parameter influences the performance of which function to deduce a minimal experiment design that exploits insights about the interplay of options (Section 3.2). After executing the experiments (Section 3.3), the performance of single functions can be modeled using any learning methodology. Finally, we derive a whole system model by summing up single models.

We assume that the system's performance varies in functions that contain non-constant loops, i.e., a configuration option determines how often they are executed and how many iterations they include. First, we distinguish between functional and non-functional configuration options: while the functional options define the problem to be solved, non-functional options are free to vary. Then, we distinguish between configuration options that affect performance and those that do not. In

	Performance-relevant	Not Performance-relevant
Functional	Problem size	Physical constant
Non-functional	Number of processes	

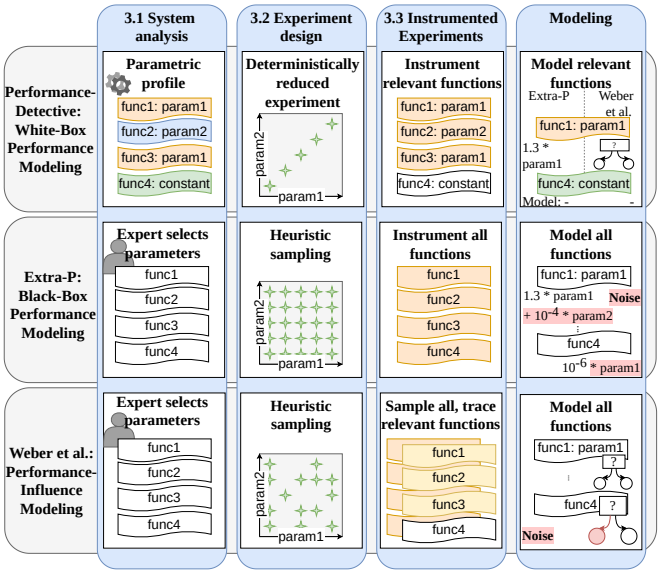


Figure 3: Modeling workflow of *Performance-Detective* compared to Extra-P [9, 36] and Weber et al. [48]. Based on the system analysis, *Performance-Detective* deterministically reduces the experiment design.

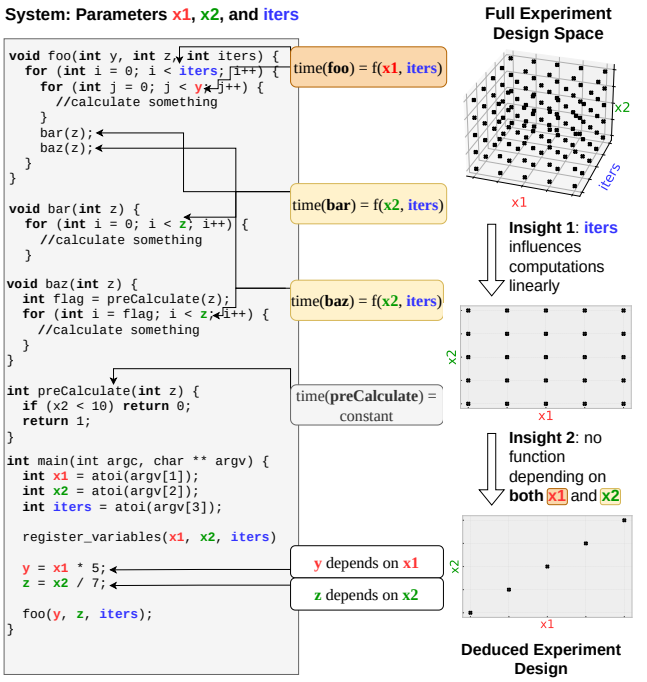


Figure 4: Running example. The system takes the configuration options `x1`, `x2`, and `iters`.

modeling, we ignore options that affect neither the result nor the performance of a program. However, the exact distinction between functional and non-functional options is subjective and domain-specific [46].

We will use the example program in Figure 4 as a running example. It takes the three parameters `x1`, `x2`, and `iters` as configuration options and does calculations based on them.

3.1 System analysis

In the first step, we analyze the system to find out how configuration options influence its performance. We do this using the Perf-Taint approach introduced in Section 2.1. The analysis of Perf-Taint outputs parameter dependencies for each performance-relevant function: a function always depends on variables that determine the loop iteration count in the function.

Example. Figure 4 shows functions' dependencies in the colored boxes. Function `preCalculate` is not influenced by the configuration options and is identified as constant. `foo` is dependent on `x1`. `bar` and `baz` iterate over the parameter `z` and depend on `x2`. We observe that `iters` has a multiplicative influence on the runtime of all functions and deduce that it linearly affects all computations. We do not consider `foo` to depend on `x2` because we exclusively measure time spent in functions (cf. Section 3.3).

3.2 Experiment design

Based on the derived dependencies from the system analysis, *Performance-Detective* deduces the minimal experiment design. From a black-box view of the system in Figure 4, *Performance-Detective* starts with a full-factorial experiment design of the three parameters `x1`, `x2`, and `iters`. Since *Performance-Detective* detects that `iters` influences the runtime of `foo`, `bar`, and `baz` linearly, measuring variations of `iters` will not provide us additional insights into the system's performance. Therefore, *Performance-Detective* can exclude this parameter from the experiment design. While the linear influence is easy to see in the example, this may not be the case for more complex real-world programs. To verify that the influence of the `iters` parameter is linear in such a case, we can measure execution times of loop iterations. If the runtime does not change significantly between them, i.e., the coefficient of variation is sufficiently small between iterations, we verify that the influence of the `iters` parameter is indeed linear.

Furthermore, we observe that `x1` and `x2` influence the performance of distinct functions and do not interact with each

other. Therefore, additional samples measuring their interactions will not provide further insights, and *Performance-Detective* can exclude them from the experiment design. In general, we assume that if options influence different sets of functions, they do not interact with each other, and *Performance-Detective* can vary them simultaneously. Also, we do not consider options linearly influencing system runtime, such as the number of main loop iterations.

Extra-P requires a full-factorial measurement setup [9] (Section 2.2). Even the sparse modeling approach with heuristics [36] requires at least five measurements per parameter to capture interactions between parameters. In contrast, *Performance-Detective* can detect the lack of interaction and strike out measurement configurations where only one parameter is changed, and the others are kept constant. This improvement reduces the dimensionality of the experiment compared to a full-factorial setup and provides further savings when compared to sparse modeling.

In the example above, Performance-Detective *deterministically reduces the number of measurements from 125 in a full-factorial experiment design to only five measurement points without sacrificing accuracy.*

3.3 Profiling

We use an instrumentation-based approach to capture the total time spent in a function during one execution of the program, summarizing time across all invocations. We measure time spent in a function exclusively, i.e., not including time spent in calls to other instrumented functions. We instrument functions identified as performance-relevant by Perf-Taint, i.e., containing loops dependent on configuration options and the main function. Previous works suggested repeating each sample five times to account for measurement noise. However, when *Performance-Detective* identifies a linear dependency of parameter on the main calculation, *we can skip the repeated measurements and even halt execution after gathering at least five iterations of the main calculation loop.* While this approach is common in analytical modeling, it has been outside the reach of automated modeling due to the difficulty of identifying configuration parameters that control the number of executions of the main loop.

Example. In Figure 4, *Performance-Detective* disregards the constant function `preCalculate` for instrumentation. We instrument `foo`, `bar`, and `baz`, as they depend on annotated input parameters, and `main` to capture the total runtime of the application. In exclusive measurement, the time obtained for `foo` does not include the time spent in `bar`. However, the time of `baz` includes the time spent in `preCalculate`.

3.4 Limitations

If all parameters are intertwined, no pruning of parameter combinations to measure is possible. Also, we do not regard binary options as switching between them results in performance jumps. We assume that performance-relevant behavior is located in computational loops and MPI communication routines. Modeling of recursion is not supported, but recursive computations are rare in HPC [13].

4 MODELING WORKFLOWS

Performance-Detective is orthogonal to the instrumentation and learning methodology, and it can reduce the costs of experiments in different performance modeling toolchains. We consider two state-of-the-art modeling workflows: the black-box, empirical Extra-P performance modeling tool (Sec. 4.1) and the white-box Performance-Influence Models (Sec. 4.2).

4.1 Extra-P

Extra-P [12] is a performance modeling tool that expresses the effect of configuration parameters $x_i, i \in \{1, \ldots, m\}$ on a performance metric $f(x_1, \ldots, x_m)$. The result, such as $t(n, p) = 10 \cdot n \cdot \log(p)$, is a familiar, human-readable function.

In practice, the configuration parameters most often analyzed are problem size and process count, and the metric of interest is usually the runtime. This allows developers to identify performance bottlenecks in their applications, especially when using it in conjunction with tools such as Score-P [26] that allows automatic instrumentation and measurement at the granularity of individual function calls.

The core assumption of the methods is that the complexity of most algorithms can be expressed using a small number of building blocks, summarized in Equation 1. The *Performance Model Normal Form (PMNF)* models a metric as a combination of polynomial and logarithmic expressions of configuration parameters. This limits the possible search space sufficiently to allow for a fast traversal while still being sufficiently flexible to cover the overwhelming majority of applications.

$$f(x_1, \ldots, x_m) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \cdot \log_2^{j_{kl}}(x_l) \qquad (1)$$

An issue not addressed in the state-of-the-art approaches is modeling more than three parameters. In practice, noise makes detecting parameters with a smaller impact on metrics of interest effectively impossible for even four parameters.

4.2 Performance-Influence Models

Performance-Influence Models (PIMs) [39] describe how configuration options and their interactions influence the performance of a system. They support a theoretically unlimited number of binary and numerical parameters. PIMs can be created using different sampling and learning techniques.

The models are learned in a black-box manner by providing the configurations used and measured total system runtime as inputs. Due to a high number of supported techniques, it is unclear which combination of methods will provide the best accuracy. However, there are works [17, 24] investigating the interplay of sampling and learning techniques and the influence of the sample size on model accuracy.

To further overcome these trade-offs between measurement effort and accuracy, Velez et al. [45, 46] proposed White-Box Performance-Influence Modeling for binary and binary-encoded configuration options. They analyze the options' influence on the system's execution and help find an optimal sample set. However, numerical options such as the problem size are not supported since continuous parameters would be tedious to encode. Weber et al. [48] present a two-step white-box process for creating PIMs on the function level. First, they use samples of the application execution to learn a PIM for every function. Then, they use tracing to derive more accurate models for functions that could not be learned with a specified accuracy. They use the extended Plackett-Burmann design [47] for sampling the numerical options and decision trees for learning models.

5 CASE STUDIES

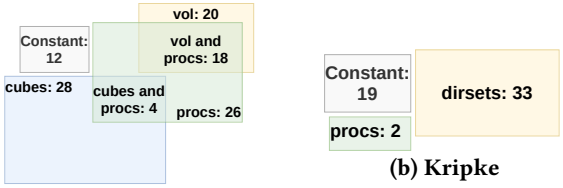
We illustrate the deduction process of *Performance-Detective* presented in Section 3 with two case studies: Kripke, a 3D Sn particle-transport proxy application, and a real-world case study from Pace3D (Parallel Algorithms for Crystal Evolution) [21], a multi-physics framework that simulates how a material reacts to outside influences. For both case studies, we only consider the execution of the main calculation loops as this is where most of the work happens.

Pace3D: Pressure calculation with projected conjugate gradient method. The calculation consists of two steps: First, an approximate solution is calculated on a coarse grid. This approximate solution is then used to calculate the real solution on the fine grid. The grids represent the material, with each cell corresponding to one cube of the material. The solver iterates until it reaches a convergence criterion or a given maximum number of iterations. We consider the number of processes and the number of coarse grid cubes as non-functional parameters. We also use the size of the material to predict how much material each process should get to achieve the best performance. Additionally, we consider the maximum number of iterations.

Kripke. Angular fluxes are calculated using different numbers of directions and groups. Kripke was built to research how different data layouts, programming paradigms, and architectures influence performance [1]. We consider the number of direction sets as well as the number of processes as parameters. Additionally, we are interested in the number of iterations.

```
for (int i = 0; i < cubes; i++) {
  //spacing = vol / cubes
  for (int j = 0; j < spacing; j++) {
    //complexity = cubes · spacing
    //complexity = cubes · (vol / cubes)
    //complexity = vol
  }
}
```

Listing 1: Program iterating over the spacing



(a) Pace3D **(b) Kripke**

Figure 5: Number of functions depending on the annotated parameters per case study as Venn diagrams.

5.1 System analysis

We analyze the scenarios with Perf-Taint [13]. The only necessary modifications to the source code are annotating and registering the variables corresponding to the parameters of interest. We run the analysis on a small problem size using only a few iterations, assuming that the computations and analysis results are representative, and validate the results with larger measurements in Section 6.2.

5.1.1 Pace3D. We annotate variables in the code corresponding to the number of processes (*procs*), material size (*vol*), coarse grid size (*cubes*), and spacing of the coarse grid (*spacing*). Additionally, we annotate the number of iterations for the fine as well as for the coarse grid. This results in a total of 21 lines of code added.

The coarse grid spacing is a dependent parameter (not linearly independent) and is calculated by dividing the material size by the coarse grid size ($spacing = vol/cubes$). Hence, we can replace the spacing by $vol/cubes$, substituting it by the independent parameters it is defined by. Listing 1 shows an example where we can conclude that the material size determines the performance of the loops.

From the results, *Performance-Detective* identifies that the maximum number of iterations on the fine grid determines the runtime of the main calculation. To verify that the influence is linear, we trace the iterations of the loop for one execution of the program and check whether the runtime of a single loop iteration changes. To do so, *Performance-Detective* calculates the coefficient of variation between them. In our

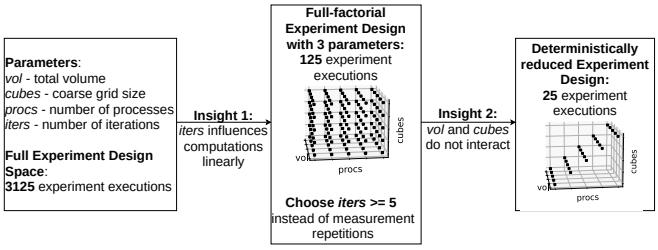


Figure 6: Deducing the minimal experiment design for Pace3D.

scenario, the coefficient of variation is 0.05 for 100 loop iterations. We conclude that the influence of *iters* is indeed linear.

Furthermore, the analysis results show that 12 functions are constant, and 54 are dependent on the annotated parameters. For the 2 functions dependent on the spacing, we can automatically assess that they in fact depend on the total volume, using replacement as shown in the example in Listing 1. The results, as shown in Figure 5a, show that the material size and the coarse grid size influence different functions.

5.1.2 Kripke. To annotate the number of processes (*procs*) as well as the number of direction sets (*dirsets*) and iterations (*iters*), we add a total of five lines of code. Using the analysis results, *Performance-Detective* analyzes that the number of iterations determines the runtime of the main calculation. As before, to verify that the influence is linear, we trace the loop iterations of one program execution. As the coefficience of variation between the measured 10 iterations is *0.0067*, we conclude that the influence of *iters* is linear. Moreover, the analysis shows (cf. Figure 5b) that 19 functions are constant and 35 are dependent on the annotated parameters, with 33 depending on *dirsets* and 2 on *procs*. The number of processes influences different functions than the number of direction sets.

5.2 Minimal Experiment Design

Figure 6 shows the process of *Performance-Detective* deducing the experiment design for Pace3D based on the insights gained from analyzing the system. From a black-box view at the system, *Performance-Detective* starts with four parameters and a full-factorial experiment design, leading to 625 measurement points. To account for measurement noise, each point has to be executed five times, leading to a total of 3125 experiment executions.

As we know that *iters* linearly influences the runtime of the main computation, *Performance-Detective* can exclude *iters* from the parameter space (*Insight 1*). Also, we can skip repetitions of the execution of measurement points and choose a sufficiently high value (≥ 5) for *iters* instead. This reduces the experiment design to 125 points that need to be executed only once. We also know that the functions affected by the coarse grid size are distinct from those affected by the total size. Thus, *Performance-Detective* can strike out configurations aiming to find interactions between them from the experiment design (*Insight 2*). This means that *Performance-Detective* varies *vol* and *cubes* simultaneously and only the number of processes independently of them. As *procs* interacts with *vol* and *cubes* according to the analysis, *Performance-Detective* includes *procs* as interacting parameter into the experiment design. Figure 7 (Sec. 6.2) shows the resulting training data points for Pace3D as crossed circles.

For Kripke, *Performance-Detective* deduces the experiment design similarly: As *iters* has a linear influence on the runtime of the main computation, it can exclude it from the parameter space (*Insight 1*) and we can skip repetitions of experiments. Furthermore, *Performance-Detective* can remove configurations aimed at finding interactions between *procs* and $dirsets$ as they influence distinct sets of functions (*Insight 2*). Thus, *Performance-Detective* varies *procs* independently of $dirsets$, resulting in 5 measuring points with $(p, dirsets) = (8, 8), (16, 16), (32, 24), (64, 32), (128, 64)$.

Performance-Detective reduced the full experiment design space of 3125 experiment executions to only 25 executions needed for Pace3D, and from 625 to 5 for Kripke.

6 EVALUATION

To evaluate *Performance-Detective*, we assess the accuracy of the performance models generated for the case studies presented in Section 5. We evaluate the reduction of repetitions by inclusion of iterations and variation of independent parameters individually. Therefore, we formulate the following research questions:

- RQ1 What is the model accuracy when generating it from a single measurement with a high number of iterations?
- RQ2 What is the model accuracy when generating it from a minimal experiment design by varying independent parameters simultaneously?

To answer the RQs, we compare our models with models generated following conventional experiment and sampling designs. The expected outcome of the evaluation is maintained accuracy while reducing the dimensionality of experiment design and not repeating experiment executions, resulting in significantly decreased cost for measurements.

6.1 Instrumented Experiments

Table 1 shows the hardware and software systems used for measuring. We instrument the application using the list of important functions generated by Perf-Taint and repeat the measurement of each configuration five times to assess RQ1 and RQ2 separately. The coefficient of variation between

CPU	Intel Xeon Gold 6230 2.1GHz
Cores	40 on 2 sockets
Memory	96 GB
GCC	10.2 (Pace3D), 11.2 (Kripke)
MPI	OpenMPI 4.0.5 (Pace3D), OpenMPI 4.1.2 (Kripke)
Software	Score-P 7.0 [26] (Pace3D), Score-P 7.1 (Kripke), Perf-Taint, Extra-P

Table 1: Measurement environment

Modeling approach	#executions	Mean error Pace3D		Mean error Kripke	
		interpolate	extrapolate	interpolate	extrapolate
Extra-P	5	8.31 %	9.31 %	4.56 %	18.32 %
Extra-P	1	8.05 %	8.74 %	4.33 %	15.17 %
Decision Trees	5	21.61 %	60.15 %	21.79 %	25.68 %
Decision Trees	1	22.62 %	62.80 %	22.38 %	25.31 %

Table 2: Mean error of models generated from *Performance-Detective* experiment design using a single application execution vs five repetitions

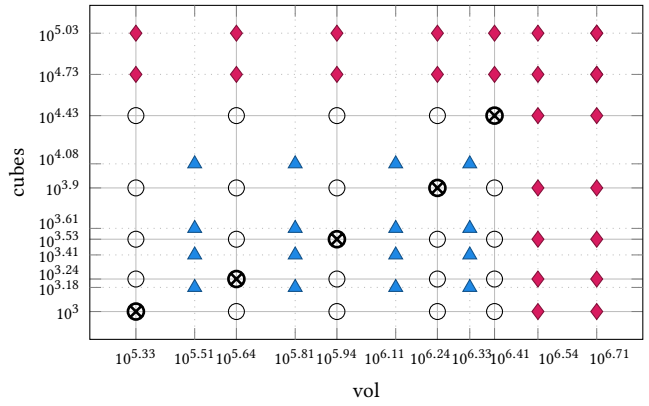


Figure 7: Overview of training and test points on logarithmic scales. Training data *Performance-Detective*: ⊗ , training data full-factorial: ○ and ⊗ . Test data interpolated: ▲ , test data extrapolated: ♦ .

the repetitions of the same configuration is 0.1 or less for all configurations. We always use the filter file containing important functions gained from system analysis to instrument only relevant functions to compare the predictions of all models. Otherwise, the evaluation would be less meaningful because the models generated when instrumenting all functions cannot predict the actual execution time as they have more profiling overhead that we cannot remove from the measurements.

6.2 RQ1: Modeling using a single measurement

To evaluate whether the inclusion of iterations in the model can simplify the experiment design, saving us repetitions of the measurements, we generate a model from a single execution of each measurement and compare the predictions of this model with test points. For evaluating the accuracy, we use the mean time of the five repeated executions of each test measurement point. To obtain a prediction of the total execution time, we sum up the predictions for the single functions and evaluate them against the execution time of the evaluation configurations.

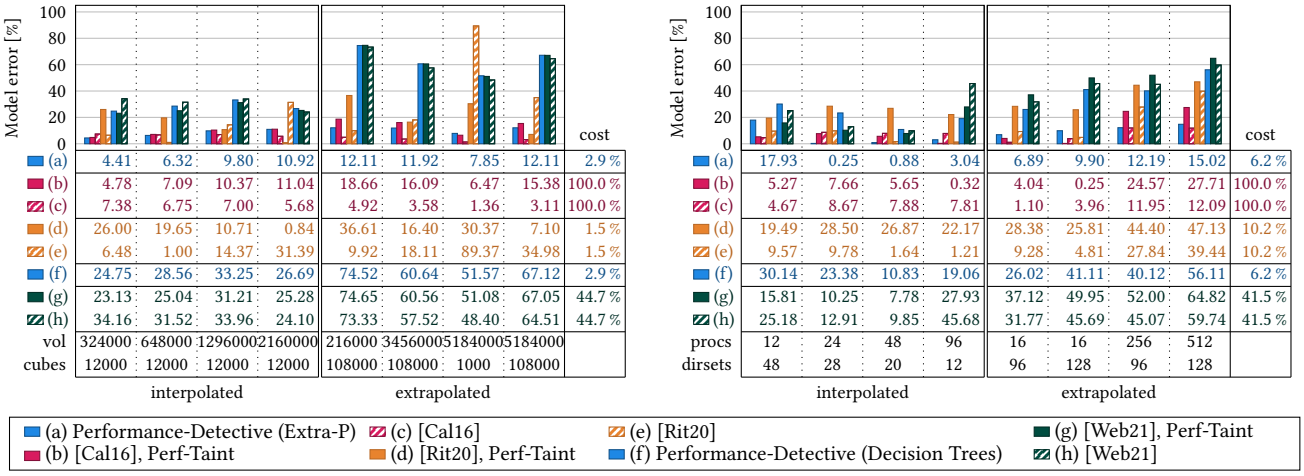
We use two testing sets, containing either inter- or extrapolated measurement points. Figure 7 shows an overview of the test data sets used for Pace3D. For the interpolated test data of both case studies, we measure configurations in between the training data points. For Pace3D, we do not interpolate the number of processes as this would not provide insight into the quality of the minimal experiment design: We still treat the number of processes as a separate parameter and vary it separately from *vol* and *cubes*, as it interacts with both according to the analysis. For the respective independent parameters, we use points between each value used for training, as shown in Figure 7. This results in 80 measuring points for Pace3D and 16 for Kripke with the interpolated values $procs = (12, 24, 48, 96)$ and $dirsets = (12, 20, 28, 48)$.

For testing how well the model predicts extrapolated configuration points, we extrapolate parameters identified as independent and measure them together with the respective other parameter using a value used for the training set. This means that we measure the extrapolated values 54,000 and 108,000 for *cubes* together with the extrapolated values 3,456,000 and 5,184,000 for *vol* as well as with all the values for *vol* used for the training set, shown in Figure 7, for Pace3D. Following this approach results in 119 measuring points for the extrapolated test data set of Pace3D and 24 for Kripke, for which we extrapolate *procs* to 256 and 512 and *dirsets* to 96 and 128.

The results in Table 2 show that the accuracy remains about the same when using only one execution as for the models generated from the mean times of five repetitions. This suggests that it is sufficient to execute the measurement of each configuration point only once if the main calculation is executed sufficiently often. For Pace3D, we set the number of iterations of the main calculation to 100, and to 10 for Kripke. However, even five iterations per application run should be enough compared to one iteration per repetition of the experiment. While the total time measured in the main calculation stays the same for both variants, we can save the initialization overhead by executing it more often in only one execution.

Experiment Design	Modeling approach	Perf-Taint?	Cost Pace3D		Mean error Pace3D		Cost Kripke		Mean error Kripke	
			core hours	#experiments	test set interpolated	test set extrapolated	core hours	#experiments	test set interpolated	test set extrapolated
Performance-Detective	Extra-P	✓	10.9	25	8.05 %	8.74 %	5.3	5	4.3 %	15.2 %
Full-factorial	Extra-P	✓	367.55	625	9.5 %	10.7 %	85.9	125	7.3 %	17.0 %
Full-factorial	Extra-P	–	367.55	625	6.2 %	6.3 %	85.9	125	6.7 %	11.2 %
Sparse	Extra-P	✓	5.54	80	8.9 %	17.7 %	8.8	50	24.6 %	34.2 %
Sparse	Extra-P	–	5.54	80	15.0 %	31.8 %	8.8	50	7.7 %	16.7 %
Performance-Detective	Decision Trees	✓	10.9	25	22.6 %	47.7 %	5.31	5	22.4 %	25.3 %
Plackett-Burman	Decision Trees	✓	164.37	245	20.1 %	45.4 %	35.66	50	15.8 %	35.1 %
Plackett-Burman	Decision Trees	–	164.37	245	27.0 %	44.2 %	35.66	50	21.8 %	31.2 %

Table 3: Mean error of different models for interpolated and extrapolated test points.



(a) Accuracy of models for the Pace3D case study for excerpts of the interpolated and extrapolated data set with 64 processes.

(b) Accuracy of models for the Kripke case study for excerpts of the interpolated and extrapolated data set.

Figure 8: Accuracy of different models for evaluation testing points.

6.3 RQ2: Varying independent parameters simultaneously

The central question in our evaluation is whether we can maintain accuracy of the model when generating it from a minimal experiment design. To answer this, we compare the accuracy of our *Performance-Detective* model to models created with conventional experiment or sampling designs. For the Extra-P multi-parameter modeler [9], this means using a full-factorial setup with 125 measuring points for Pace3D and 25 for Kripke. For the Extra-P sparse modeler [36], we use 5 values for each parameter while keeping the others constant and one interaction point for each parameter combination, resulting in 16 points for Pace3D and 10 for Kripke. To learn PIMs, we use the extended Plackett-Burman Design [47] with 49 samples (Pace3D) or 10 samples (Kripke) and five levels, effectively being a subset of the full-factorial measuring points. We use the script of Weber et al. [48] to learn PIMs on a function level based on known parameter dependencies. However, we do not use their two-step process but only one profiling step using compiler instrumentation. In contrast to them, we use mean values derived from the repetitions of the experiment for modeling. They build a separate model for each repetition of the experiment.

As we showed in Section 6.2 that the accuracy of the models generated from a single application execution per measurement point is comparable to the accuracy of the model generated from repeating measurements five times for each point, we will continue with these models for *Performance-Detective*.

Table 3 shows the required number of measurements of each approach and the cost in total core hours. The cost of individual samples are not uniform across the configuration space, as the impact the parameters have on performance can mean some samples are orders of magnitude more expensive than others. We therefore consider the total core hours to be the more important metric. To evaluate the predictions'

accuracy, we use the same test configurations as in Section 6.2, inter- and extrapolating the training data points. Table 3 depicts the mean error of the models from the different approaches.

6.3.1 Interpolated test data. Figure 8 shows the comparison of prediction error among models learned using Extra-P (depicted as [Cal16] and [Rit20]) and Decision Trees (depicted as [Web21]) both with and without the information about dependencies from Perf-Taint. We compare our approach to the full-factorial setup and the sparse modeling approach.

One would expect all approaches to be reasonably effective at predicting configuration options within the range of measurements already available. In the case of Extra-P, using the full set of measurements leads to accurate models, with model errors seldom passing 10%, and the use of Perf-Taint has a relatively small impact on results. The results of the sparse modeler are worse overall, but for Pace3D, they are significantly improved by the use of Perf-Taint. *Performance-Detective* achieves results comparable to the full factorial experiment design of Extra-P, while requiring only a fraction (2.9% and 6.18%, respectively) of the cost.

The performance-influence models, however, show a consistently higher model error rate of over 20%. When using *Performance-Detective*, the quality of the models remains approximately the same, while the cost is reduced by a factor of 15 for Pace3D and 7 for Kripke.

6.3.2 Extrapolated test data. Figure 8 shows an overview of the accuracy of the different Extra-P (depicted as [Cal16] and [Rit20]) and PIM models ([Web21]), both using information about dependencies from Perf-Taint and without it.

When extrapolating, predicting the runtime is more challenging as any errors are quickly magnified. Overall, we observe the same trends as for interpolated evaluation points, with a couple of differences. For Pace3D, the sampling approach of Extra-P generates significantly higher errors when not used in conjunction with Perf-Taint. However, for Kripke, the errors of the sampling approach are increased when used together with Pef-Taint. The performance-influence models show a larger model error rate, this time of over 40% for Pace3D and over 30% for Kripke when using the Plackett-Burman sampling design. However, the approach for learning performance-influence models does not extrapolate, as we can see in our data: It only predicts the time measured for the highest training value of the respective extrapolated parameter.

While providing information about the dependencies derived by Perf-Taint can be beneficial to the accuracy of the sparse Extra-P modeler, it decreases accuracy for the full-factorial experiment design. This is because with a full-factorial design, there is a lot of measurement data and Perf-Taint only removes wrong dependencies that model noise. The accuracy of the sparse modeler increases because it has less measurement data and very few information about parameter interplay.

6.3.3 Discussion. For both case studies, we can maintain the accuracy of the model generated with the experiment design deduced by *Performance-Detective* (25 and 5 experiment executions) as compared to a full-factorial design (625 and 125 experiment executions) and Plackett-Burman sampling (245 and 50 experiment executions). While for Pace3D, the sparse modeler is even less expensive than *Performance-Detective*, it has a lower accuracy than *Performance-Detective* (mean accuracy of 85.9% and 74.9% with and without Perf-Taint as compared to 91.5%) with especially worse predictions for extrapolated test points (mean accuracy of 17.7% and 31.8% with and without Perf-Taint compared to 9.3%).

For the PIMs, we observe a generally significant prediction error in both evaluation sets, which does not change much depending on using our minimal experiment design or sampling, and usage of Perf-Taint. This indicates that the learning method decision trees is not well-suited to model the performance of these applications.

Across evaluation scenarios, we observe that Performance-Detective *always dramatically decreases the cost of experiments, while not meaningfully degrading model quality.*

6.4 Threats to validity

A threat to external validity is that we ran the taint analysis on a small problem size. This was possible for the scenarios shown because the calculations are the same and still representative. However, this is not guaranteed in the general case, as sometimes different branches can be active depending on problem size. We can detect this by using control-flow tainting, provided by Perf-Taint. The only impact this scenario would have is that the taint analysis will have a higher cost.

Regarding internal validity, not inlining functions defined as inline is a potential issue. Functions are not inlined if we detect them as being performance-relevant. This could distort the measurements, as not inlining might incur a performance penalty on the applications as a whole. The overhead introduced by the profiling itself is a further source of distortion. We mitigate this by only instrumenting functions detected as performance-relevant, and keep the overhead as low as possible.

7 RELATED WORK

Parametric Performance Models. The difficulty of modeling modern software systems with many parameters has been addressed by parametric software performance models [5, 28, 32, 35] and Performance-Influence Models (PIMs) [39].

The parametric software performance modelling approach Palladio [5, 35] allows the modelling of parametric dependencies. However, automated extraction of these models [28, 32] assumes that the system consists of software components with well-defined interfaces, which is usually not the case in HPC applications.

In white-box PIMs, machine learning and heuristic methods are used to iteratively learn models representing the influences and interactions of various application parameters. However, white-box PIMs are either limited to binary-encoded arguments or require expensive trace measurements (cf. Sections 2.2, 4.2). Other methods of learning efficient performance models for highly configurable systems use Fourier transformations to reduce the number of samples and parameter combinations [18, 51].

HPC Performance Modeling. Analytical performance models can be created manually through source code inspection and guidance by performance engineers [20, 25]. Unfortunately, such models require significant effort, and excluding empirical data makes the models prone to underestimate the effects of hardware congestion and network performance.

Extra-P [12] is the state-of-the-art workflow for empirical and parametric performance modeling, extended with multi-parameter modeling [9], validation of high-performance libraries [38], prototyping hardware requirements for HPC applications [11], and with white-box modeling [13, 14]. We present a wider discussion in Sections 2.1 and 4.1.

PALM [43] constructs performance models from annotated application source code and enhances them with user-provided insights, program analysis, and measurements. Aspen [41] is a domain-specific language for the manual specification of program operations. COMPASS [31] uses Aspen annotations to generate application models statically, but it requires user intervention when the kernels cannot be analyzed automatically. In contrast, *Performance-Detective* does not require user input and manual analysis steps.

Online learning improves the accuracy of static and compiler-based performance models [6, 7]. Machine learning methods have been applied successfully to model performance [22, 30, 42] and decrease the negative effects of measurement noise [37]. *Performance-Detective* provides a complete and white-box workflow that requires neither heuristics nor approximations to construct performance models, and we provide validated parametric dependencies of models.

Auto-tuning. Auto-tuning applies an optimization method to achieve one or more goals, such as minimizing runtime [2] or floating-point operations per second [44] of an application. To find the best configuration among the search space, global or local search methods are employed [3, 23, 33]. In contrast to auto-tuning approaches that find the best configuration for a given optimization goal, *Performance-Detective* enables efficient modeling of the whole configuration space and interactions among configuration options.

8 CONCLUSION

We have shown that we can significantly lower the cost of automatic performance modeling of applications with multiple configuration parameters. We deduce a minimal experiment design with *Performance-Detective* by exploiting automatically derived insights about parameter interplay and main loops and thus reduce the number and cost of required measurements while achieving comparable accuracy to methods costing more than an order of magnitude more compute hours. With *Performance-Detective*, we model the Pace3D real-world multi-physics solver using 25 rather than 3125 measurements, require 34 times fewer core hours and achieve and still maintain a model accuracy of 91.5% compared to 93.8% and 90.6% when all measurements are used. Furthermore, we model Kripke, reducing needed measurements from 125 to 5, leading to 16 times fewer core hours needed, while maintaining an accuracy of 89.2% compared to 90.6% using all measurements.

ACKNOWLEDGMENTS

Larissa Schmid was supported by the Ministry of Science, Research and the Arts Baden-Württemberg (Az: 7712.14-0821-2). This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement PSAP, No. 101002047), and from the Schweizerische Nationalfonds zur Förderung der wissenschaftlichen Forschung (SNF, Swiss National Science Foundation) through Project 170415. The authors acknowledge the financial support by the Federal Ministry for Economic Affairs and Energy of Germany in the project ProStroM (project number 03ETB026C). This work was supported by KASTEL Security Research Labs. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

REFERENCES

[1] A. J. Kunen, T. S. Bailey, and P. N. Brown. 2015. Kripke - a massively parallel transport mini-app. In *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method (ANS MC '15)* (Nashville, Tennessee).

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 303–315. https://doi.org/10.1145/2628071.2628092

[3] Prasanna Balaprakash, Stefan M. Wild, and Paul D. Hovland. 2013. An Experimental Study of Global and Local Search Algorithms in Empirical Performance Tuning. In *High Performance Computing for Computational Science - VECPAR 2012*, Michel Daydé, Osni Marques,

and Kengo Nakajima (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 261–269.

[4] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. 2008. A Regression-Based Approach to Scalability Prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing* (Island of Kos, Greece) *(ICS '08)*. Association for Computing Machinery, New York, NY, USA, 368–377. https://doi.org/10.1145/1375527.1375580

[5] Steffen Becker, Heiko Koziolek, and Ralf Reussner. 2009. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software* 82, 1 (2009), 3–22. https://doi.org/10.1016/j.jss.2008.03.066 Special Issue: Software Performance - Modeling and Analysis.

[6] Arnamoy Bhattacharyya and Torsten Hoefler. 2014. PEMOGEN: Automatic Adaptive Performance Modeling During Program Runtime. In *Proc. of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. ACM, New York, NY, USA, 393–404. https://doi.org/10.1145/2628071.2628100

[7] Arnamoy Bhattacharyya, Grzegorz Kwasniewski, and Torsten Hoefler. 2015. Using Compiler Techniques to Improve Automatic Performance Modeling. In *Proc. of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT'15)* (San Francisco, CA, USA). San Francisco, CA, USA, 1–12.

[8] Christian Bischof, Dieter an Mey, and Christian Iwainsky. 2012. Brainware for green HPC. 27, 4 (2012), 227–233. https://doi.org/10.1007/s00450-011-0198-5

[9] Alexandru Calotoiu, David Beckinsale, Christopher W. Earl, Torsten Hoefler, Ian Karlin, Martin Schulz, and Felix Wolf. 2016. Fast Multi-parameter Performance Modeling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)* (Taipei, Taiwan, 2016-09). IEEE, 172–181. https://doi.org/10.1109/CLUSTER.2016.57

[10] Alexandru Calotoiu, Marcin Copik, Torsten Hoefler, Marcus Ritter, Sergei Shudler, and Felix Wolf. 2020. ExtraPeak: Advanced Automatic Performance Modeling for HPC Applications. In *Software for Exascale Computing - SPPEXA 2016-2019* (Cham) *(Lecture Notes in Computational Science and Engineering)*, Hans-Joachim Bungartz, Severin Reiz, Benjamin Uekermann, Philipp Neumann, and Wolfgang E. Nagel (Eds.). Springer International Publishing, 453–482. https://doi.org/10.1007/978-3-030-47956-5_15

[11] Alexandru Calotoiu, Alexander Graf, Torsten Hoefler, Daniel Lorenz, Sebastian Rinke, and Felix Wolf. 2018. Lightweight Requirements Engineering for Exascale Co-design. In *Proc. of the 2018 IEEE International Conference on Cluster Computing (CLUSTER), Belfast, UK*. IEEE, 1–11.

[12] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. 2013. Using automated performance modeling to find scalability bugs in complex codes. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013-11). 1–12. https://doi.org/10.1145/2503210.2503277 ISSN: 2167-4337.

[13] Marcin Copik, Alexandru Calotoiu, Tobias Grosser, Nicolas Wicki, Felix Wolf, and Torsten Hoefler. 2021. Extracting Clean Performance Models from Tainted Programs. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 403–417. https://doi.org/10.1145/3437801.3441613

[14] Marcin Copik and Torsten Hoefler. 2019. perf-taint: Taint Analysis for Automatic Many-Parameter Performance Modeling. *ACM Student Research Competition at ACM/IEEE Supercomputing* (2019). https://sc19.supercomputing.org/proceedings/src_poster/src_poster_pages/spostg110.html

[15] dfsan 2019. Clang 9 Documentation - DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html.

[16] Dan Gohman. 2009. ScalarEvolution and Loop Optimization. Talk at LLVM Developer's Meeting.

[17] Alexander Grebhahn, Norbert Siegmund, and Sven Apel. 2019. Predicting Performance of Software Configurations: There is no Silver Bullet. (2019). arXiv:1911.12643 http://arxiv.org/abs/1911.12643

[18] Huong Ha and Hongyu Zhang. 2019. Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 470–480. https://doi.org/10.1109/ICSME.2019.00080

[19] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Ciudad Real, Spain) *(ESEM '16)*. Association for Computing Machinery, New York, NY, USA, Article 23, 10 pages. https://doi.org/10.1145/2961111.2962602

[20] Torsten Hoefler, William Gropp, William Kramer, and Marc Snir. 2011. Performance modeling for systematic performance tuning. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[21] J. Hötzer, A. Reiter, H. Hierl, P. Steinmetz, M. Selzer, and Britta Nestler. 2018. The parallel multi-physics phase-field framework Pace3D. *Journal of Computational Science* 26 (2018), 1–12. https://doi.org/10.1016/j.jocs.2018.02.011

[22] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. 2005. An Approach to Performance Prediction for Parallel Applications. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing* (Lisbon, Portugal) *(Euro-Par'05)*. Springer-Verlag, Berlin, Heidelberg, 196–205.

[23] Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. 2012. A multi-objective auto-tuning framework for parallel codes. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1109/SC.2012.7

[24] 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

[25] Darren J Kerbyson, Henry J Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J Wasserman, and Mike Gittings. 2001. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. 37–37.

[26] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91.

[27] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. 2019. Tradeoffs in modeling performance of highly configurable software systems. 18, 3 (2019), 2265–2283. https://doi.org/10.1007/s10270-018-0662-9

[28] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. 2010. Using Genetic Search for Reverse Engineering of Parametric Behavior Models for Performance Prediction. *IEEE Transactions on Software Engineering* 36, 6 (2010), 865–877. https://doi.org/10.1109/TSE.2010.69

[29] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, Washington, DC, USA.

[30] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. 2007. Methods of inference and learning for performance modeling of parallel applications. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Jose, California, USA) *((PPoPP '07))*. ACM, 249–258.

[31] Seyong Lee, Jeremy S Meredith, and Jeffrey S Vetter. 2015. Compass: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 405–414.

[32] Manar Mazkatli, David Monschein, Johannes Grohmann, and Anne Koziolek. 2020. Incremental Calibration of Architectural Performance Models with Parametric Dependencies. In *IEEE International Conference on Software Architecture (ICSA 2020)*. Salvador, Brazil, 23–34. https://doi.org/10.1109/ICSA47634.2020.00011

[33] Philip Pfaffe, Martin Tillmann, Sigmar Walter, and Walter F. Tichy. 2017. Online-Autotuning in the Presence of Algorithmic Choice. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1379–1388. https://doi.org/10.1109/IPDPSW.2017.28

[34] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. 2010. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10* (Cape Town, South Africa), Vol. 1. ACM Press, 445. https://doi.org/10.1145/1806799.1806864

[35] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. 2016. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, Cambridge, MA. http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures

[36] Marcus Ritter, Alexandru Calotoiu, Sebastian Rinke, Thorsten Reimann, Torsten Hoefler, and Felix Wolf. 2020. Learning Cost-Effective Sampling Strategies for Empirical Performance Modeling. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (New Orleans, LA, USA, 2020-05). IEEE, 884–895. https://doi.org/10.1109/IPDPS47924.2020.00095

[37] Marcus Ritter, Alexander Geiß, Johannes Wehrstein, Alexandru Calotoiu, Thorsten Reimann, Torsten Hoefler, and Felix Wolf. 2021. Noise-Resilient Empirical Performance Modeling with Deep Neural Networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 23–34. https://doi.org/10.1109/IPDPS49936.2021.00012

[38] Sergei Shudler, Alexandru Calotoiu, Torsten Hoefler, Alexandre Strube, and Felix Wolf. 2015. Exascaling Your Library: Will Your Implementation Meet Your Expectations?. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 165–175. https://doi.org/10.1145/2751205.2751216

[39] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015* (Bergamo, Italy). ACM Press, 284–294. https://doi.org/10.1145/2786805.2786845

[40] Connie U. Smith. 1993. Software performance engineering. In *Performance Evaluation of Computer and Communication Systems*, Lorenzo Donatiello and Randolph Nelson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–536.

[41] K. L. Spafford and J. S. Vetter. 2012. Aspen: A Domain Specific Language for Performance Modeling. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 84, 11 pages.

[42] Jingwei Sun, Guangzhong Sun, Shiyan Zhan, Jiepeng Zhang, and Yong Chen. 2020. Automated Performance Modeling of HPC Applications Using Machine Learning. *IEEE Trans. Comput.* 69, 5 (2020), 749–763. https://doi.org/10.1109/TC.2020.2964767

[43] N. R. Tallent and A. Hoisie. 2014. Palm: Easing the Burden of Analytical Performance Modeling. In *Proc. of the 28th ACM International Conference on Supercomputing* (Munich, Germany) *(ICS '14)*. ACM, New York, NY, USA, 221–230. https://doi.org/10.1145/2597652.2597683

[44] Sébastien Varrette, Frédéric Pinel, Emmanuel Kieffer, Grégoire Danoy, and Pascal Bouvry. 2020. Automatic Software Tuning of Parallel Programs for Energy-Aware Executions. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, and Konrad Karczewski (Eds.). Springer International Publishing, Cham, 144–155.

[45] 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. https://doi.org/10.1007/s10515-020-00273-8

[46] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-box analysis over machine learning: Modeling performance of configurable systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1072–1084.

[47] JC Wang and CF Jeff Wu. 1995. A hidden projection property of Plackett-Burman and related designs. *Statistica Sinica* (1995), 235–250.

[48] Max Weber, Sven Apel, and Norbert Siegmund. 2021. White-Box Performance-Influence Models: A Profiling and Learning Approach. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (Madrid, Spain, 2021-05). IEEE, 1059–1071. https://doi.org/10.1109/ICSE43902.2021.00099

[49] Nicolas Wicki. 2020. Control Flow Taint Analysis for Performance Modeling in LLVM. Bachelor's Thesis.

[50] Murray Woodside, Greg Franks, and Dorina C. Petriu. 2007. The Future of Software Performance Engineering. In *Future of Software Engineering (FOSE '07)* (Minneapolis, MN, USA, 2007-05). IEEE, 171–187. https://doi.org/10.1109/FOSE.2007.32

[51] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance Prediction of Configurable Software Systems by Fourier Learning (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015-11). 365–373. https://doi.org/10.1109/ASE.2015.15