Fast Multi-Parameter Performance Modeling

Alexandru Calotoiu*, David Beckingsale‡, Christopher W. Earl‡, Torsten Hoefler†
Ian Karlin‡, Martin Schulz‡, Felix Wolf*

*Technische Universität Darmstadt, Darmstadt, Germany
†ETH Zürich, Zurich, Switzerland
‡Lawrence Livermore National Laboratory, Livermore, USA
calotoiu@cs.tu-darmstadt.de, beckingsale1@llnl.gov, earl2@llnl.gov, htor@inf.ethz.ch,
karlin1@llnl.gov, schulzm@llnl.gov, wolf@cs.tu-darmstadt.de

***Abstract*—Tuning large applications requires a clever exploration of the design and configuration space. Especially on supercomputers, this space is so large that its exhaustive traversal via performance experiments becomes too expensive, if not impossible. Manually creating analytical performance models provides insights into optimization opportunities but is extremely laborious if done for applications of realistic size. If we must consider multiple performance-relevant parameters and their possible interactions, a common requirement, this task becomes even more complex. We build on previous work on automatic scalability modeling and significantly extend it to allow insightful modeling of any combination of application execution parameters. Multi-parameter modeling has so far been outside the reach of automatic methods due to the exponential growth of the model search space. We develop a new technique to traverse the search space rapidly and generate insightful performance models that enable a wide range of uses from performance predictions for balanced machine design to performance tuning.**

I. Introduction

Ever-growing application complexity across all domains, including but not limited to theoretical physics, fluid dynamics, or climate research requires a continuous focus on performance to productively use the large-scale machines that are being procured. However, designing such large applications is a complex task demanding foresight since they require large time investments in development and verification and are therefore meant to be used for decades. Thus, it is important that the applications be efficient and potential bottlenecks are identified early in their design as well as throughout their whole life cycle.

Continuous performance analysis starting in early stages of the development process is an indispensable prerequisite to ensure early and sustained productivity. This in turn, requires the availability of continuously updated performance models reflecting design updates and supporting the optimization process. Such models allow problems in applications to be detected early and their severity to be determined when the cost of eliminating the problems is still small. This is increasingly important since mitigating such problems can often take several person years. Despite this, though, current practice often looks different: since creating detailed analytical models is too time consuming, such models are often only built on back-of-the-envelope calculations, rough estimates, simple and manual spreadsheet calculations, or even only developer intuition.

Further, to ensure that current applications are performance-bug free, it is often not enough to analyze any one aspect such as processor count or problem size. The effect that the one varying parameter has on performance must be understood not only in a vacuum, but also in the context of the variation of other relevant parameters, including algorithm variations, tuning parameters such as tiling, or input characteristics.

In this work, we address this challenge by automating the model generation process for large and complex parameter spaces, making it feasible to have updated performance models available throughout the entire development process. In particular, we present a fast multi-parameter analysis approach for parallel applications: combining standard performance profiling [2] with an enhanced version of a lightweight automatic performance-modeling method [5], we generate human-readable empirical models with multiple parameters that allow insights into application performance. For example, our approach can determine exact models, such as the previously undocumented floating-point instruction count of the kernel `LTimes` in the particle-transport proxy application Kripke [11] depending on the processor count p, number of directions d, and number of groups g. The model is $f(p, d, g) = 5.4 \cdot d \cdot g$ millions of floating point operations (Mflop). The designer can then reduce the accuracy to simplify the handling, e.g., round to the next order of magnitude $f(p, d, g) \approx 10 \cdot d \cdot g$ Mflop, or even simply consider the asymptotic scaling $f(p, d, g) \in \mathcal{O}(d \cdot g)$ Mflop. Furthermore, the model certifies that the floating-point instruction count does not depend on the number of processors.

In this work, we describe our techniques and implementation to automatically generate multi-parameter performance models for parallel applications. In a detailed case study with two applications, we show how our modeling framework can be applied in practice to gain non-obvious insights into performance characteristics of real codes. The major contributions of our work are:

- An automated technique to generate empirical performance models of parallel applications as a function of an arbitrary set of input parameters.

- Two heuristics to accelerate the search for suitable performance models employed by our generation method, making our approach practically feasible. The first heuristic speeds up multi-parameter modeling, as it reduces the search space to only combinations of the best single-parameter models. The second heuristic speeds up model selection for single parameter models. Combined, the heuristics allow a search space of hundreds of billions of models to be reduced to under a thousand and to reduce the time required to obtain a multi-parameter model from over 6 years to under 6 milliseconds.
- A fully-functional implementation of the multi-parameter modeling technique using the two heuristics to demonstrate and validate our approach in practice using both synthetically generated data and measurements from three scientific applications.
- Two case studies where we highlight the utility of multi-parameter modeling to understand application behavior and to gain new insights into performance optimization opportunities while putting existing performance expectations to the test.

The remainder of the paper is organized as follows. After providing background on our prior work related to automatic performance modeling with a single parameter, we outline our multi-parameter modeling method, followed by details on the heuristics employed to reduce the search space. We then provide a validation of our heuristics. Finally, we present the two case studies, discuss the insights gained, and offer a conclusion.

II. Single-Parameter Modeling

Our multi-parameter performance-modeling approach builds upon a method to generate empirical performance models with a single model parameter, primarily scaling models, from performance measurements [5].

A. The Performance Model Normal Form

A key concept of our single-parameter approach is the *performance model normal form* (PMNF). The PMNF models the effect of a single parameter (predictor) x on a response variable of interest $f(x)$, typically a performance metric such as execution time or a performance counter. It is specified in Equation 1.

$$f(x) = \sum_{k=1}^{n} c_k \cdot x^{i_k} \cdot log_2^{j_k}(x) \tag{1}$$

The PMNF allows building a function search space, which we then traverse to find the member function that comes closest to representing the set of measurements. This assumes that the true function is contained in this search space. A possible assignment of all i_k and j_k in a PMNF expression is called a *model hypothesis*. The sets $I, J \subset \mathbb{Q}$ from which the exponents i_k and j_k are chosen and the number of terms n define the discrete model search space. Our experience suggests that neither I and J nor n have to be particularly large to achieve a good fit. We then automatically determine the coefficients of all hypotheses using regression and choose the hypothesis with the smallest error to find the most likely model function. Often a simple selection like the following will be sufficient to model an application: $n = 2$, $I = \left\{\frac{0}{4}, \frac{1}{4}, \ldots, \frac{12}{4}\right\}$, and $J = \{0, 1, 2\}$. It is possible to expand or modify the sets or the number of terms if clear expectations regarding the application behavior exist, but such prior knowledge is not required in the common case.

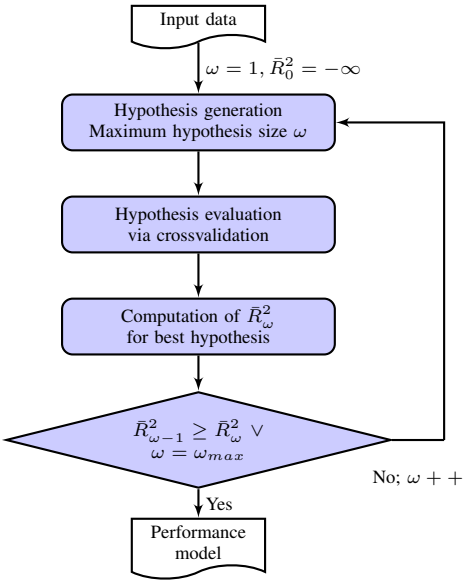

Fig. 1: Iterative model refinement process. Solid boxes represent actions or transformations, and banners their inputs and outputs [5].

For the above process to yield good results, the true function that is being modeled should not be qualitatively different from what the normal form can express. Discontinuities, piece-wise defined functions, and other behaviors that cannot be modeled by the normal form will lead to sub-optimal results.

B. Model Generation

Our single parameter model generator requires a set of performance profiles as input, representing runs with one changing parameter. These profiles can be obtained using existing performance measurement tools. Here, we use the performance measurement system Score-P [2], which collects several performance metrics, including execution time and various hardware and software counters, broken down by call path and process. Based on such profiles, we compute one model for each combination of target metric and call path, enabling a very fine-grained scalability analysis even of very complex applications. The instrumentation done by Score-P is direct, which on the one hand necessitates careful control of the runtime dilation, but on the other hand allows an accurate attribution of counters and timers to individual call paths. Other data sources from other performance measurement tools are equally possible and simply require some form of input format conversion. Past experience has shown

that as few as five different measurements for one parameter are enough for successful model generation. If noise can affect the measurements, experiments will be repeated until the data provides reasonably tight 95% confidence intervals: Confidence intervals larger than 5% of the absolute difference between measurements will often lead to noise being modeled rather than the desired behavior.

Running the number of required experiments can be managed by automated test frameworks [1] and the runtime required can be kept in check by carefully selecting the experiment range. For example in a weak scalability study, unless there is reason to believe that the application would behave qualitatively different at a larger scale, there is no reason to run experiments at massive scales. A qualitative difference in this context could be the result of changing an algorithm for a part of the application beyond a certain processor count. We mainly used this approach to study scalability by varying the number of processes—both in weak and strong scaling mode.

Once the profiles are available, we identify models in an iterative process, which is illustrated in Figure 1: in each step, we instantiate a number of model hypotheses of a certain size (i.e., number of terms) according to the PMNF defined above, and select the winner through cross-validation. We start the process with a one-term model, which we extend after each step. We continue this process until we reach the configurable maximum model size or discover onsets of over-fitting with the help of the adjusted coefficient of determination $\bar{R}^2$ [6]. $\bar{R}^2$ indicates which share of the variation in the data is explained by the model function and thus can be used to assess the quality of the fit. Its values are between 0 and 1 and values closer to 1 indicate a better quality of fit. The advantage of $\bar{R}^2$ is that it penalizes models for having more degrees of freedom and therefore helps to detect over-fitting. For further details of model generation including references to the statistical methods employed, we refer to Calotoiu et al. [5].

III. Multi-Parameter Modeling

Common questions asked by developers when trying to understand the behavior of applications are:

- How does application performance change when more processors are used?
- How does application performance change when the problem size is increased or decreased?

Changing the processor count while keeping everything else fixed is also known as strong scaling. The goal of many large-scale applications, however, is to solve larger problems using more processing power, leading to the concept of weak scaling. Weak scaling is often defined as the application's behavior when the problem size *per processor* is fixed and the processor count is varied. Creating an experimental setup in practice where the problem size per processor is fixed is not trivial as the problem decomposition is not arbitrary in the general case. When considering the pressure on applications to judiciously use computing resources both questions must be answered, and a new vital question arises:

- Are the effects of processor variation and problem size variation independent of each other or can they amplify each other?

For example, a weak-scaling run of the kernel `SweepSolver` in Kripke [11], a particle transport proxy application, has a runtime model for processor variation of $t(p) = O(p^{1/3})$ and a runtime model for varying the number of dimensions of $t(d) = O(d)$. The number of dimensions influences the problem size proportionally. It now needs to be determined how these two factors play together. Depending on their interaction, the application is scalable or not. For example, it would make a huge difference if the combined effect of processor variation and number of dimensions was $t(p, d) = O(p^{1/3} \cdot d)$ or $t(p, d) = O(p^{1/3} + d)$.

A. A Normal Form for Multiple Parameters

Below, we expand the original performance model normal form presented in Section II-A to include multiple parameters.

$$f(x_1, \ldots, x_m) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{m} x_l^{i_{k_l}} \cdot log_2^{j_{k_l}}(x_l) \qquad (2)$$

This expanded normal form allows a number m of parameters to be combined in each of the n terms that are summed up to form the model. Each term allows each parameter x_l to be represented through a combination of monomials and logarithms. The sets $I, J \subset \mathbb{Q}$ from which the exponents i_{k_l} and j_{k_l}, respectively, are chosen can be defined as in the one-parameter case.

B. Considerations for Multiple Parameters

Of course, if multiple parameters are considered, performance experiments have to be conducted for all combinations of parameter values and the total number of experiments that is required grows accordingly. While this might be manageable if the number of parameters considered is small enough (single digit) and/or the cost of an individual experiment is very small, another and more serious problem emerges even for two and three of parameters.

Looking at Equation 2, the combinatorial explosion of the search space for model hypotheses that multi-parameter modeling generates becomes apparent. This shows the need for efficient methods for traversing the search space. The maximum number of terms should always allow at least an additive combination of all parameters so $n \geq m$. For convenience, we will use $n = 3$ throughout the following example, which is required for three parameters. With $n = 3$ and I, J defined as in Section II-A, the model search space when one parameter is considered can be built as follows: For each of the terms there are $|I| \cdot |J|$ possible options, i.e., 54 in this case. The order of terms is irrelevant and the same term cannot be repeated, therefore the cardinality of the search space is the binomial coefficient $\binom{|I| \cdot |J|}{n}$, i.e., 24,804 models in total. If we now consider two parameters, each term has $(|I| \cdot |J|)^2$ possible options, i.e., 2916 in total. The model search space in this case contains 4,128,234,660 candidates. Let us assume that 300,000 hypotheses can be evaluated in a second, a rate drawn from our

experience on current commodity personal computers. Even with the simplifying assumption that evaluating a hypothesis with multiple parameters would take as much as evaluating a hypothesis with only one parameter, it would still take more than three hours to select the best model for a single combination of metric and kernel. With three parameters the model search space contains around $6.51 \cdot 10^{14}$ candidates, and with m parameters $\binom{(|I|\cdot|J|)^m}{n}$, making the search for the best fit a daunting task. Spending six years to compute the best model with three parameters for one metric of one kernel is not something any developer would consider. Obviously, one does not need many parameters to make the traversal of such a multi-parameter search space practically infeasible. While this problem is embarrassingly parallel, the resource requirements for such a performance modeling process will far outweigh any gains obtained through optimization of the target application. To overcome the challenge that the size of the search space presents, we speed up the search process using novel heuristics, which we describe in Section IV.

IV. Fast Multi-Parameter Modeling

We build on the concepts of single-parameter modeling from our prior work, but extend and optimize them to match the new requirements posed by modeling multiple arbitrary parameters. Note that the original method is only capable of modeling the scalability as a function of one parameter, usually the number of processes. This single-parameter modeling allowed several simplifying assumptions that do not hold for general multi-parameter modeling, for example, that we can search all models for this one parameter. Thus, we first develop a new search method for the model space of a single arbitrary parameter and then derive an effective method to combine all single-parameter models into a single model for all parameters.

A. Improved Single-Parameter Modeling

To model multiple parameters without the time to solution becoming prohibitive, with billions of candidates being generated for as few as three parameters, the existing approach to model detection is no longer sufficient: we must find a way to reduce the search space of model hypotheses. The following method is only applicable to single-parameter modeling. It complements the hierarchical search outlined in the next subsection to speed up the entire modeling process.

Reducing a search space is often related to finding some ordering of the search space, i.e., finding a way to rank the possible hypotheses, and our method is no exception. We use the following sets of modeling terms for $n = 1$: $I = \{\frac{0}{4}, \frac{1}{4}, \ldots, \frac{12}{4}\}$, and $J = \{0\}$ as an example to demonstrate our hypothesis ranking approach. This example generates a small set of hypotheses when modeling a single parameter x: $\{x^0; x^{1/4}, \ldots, x^{12/4}\}$. Based on our experience with performance modeling, we make the following observation: if we rank the hypothesis functions by the magnitude of their first derivative at the observation with the largest parameter value then the respective error function of the ranked hypotheses is unimodal, i.e., a function that has a unique minimum or maximum point and is monotonically decreasing/increasing towards it. The unique minimum of this discrete function will be the best matching term. This is intuitive: the regression and cross-validation approach we use always finds the best possible coefficients for each of the terms to fit the available data, but the results will be better the closer the hypothesis function is to the true function.

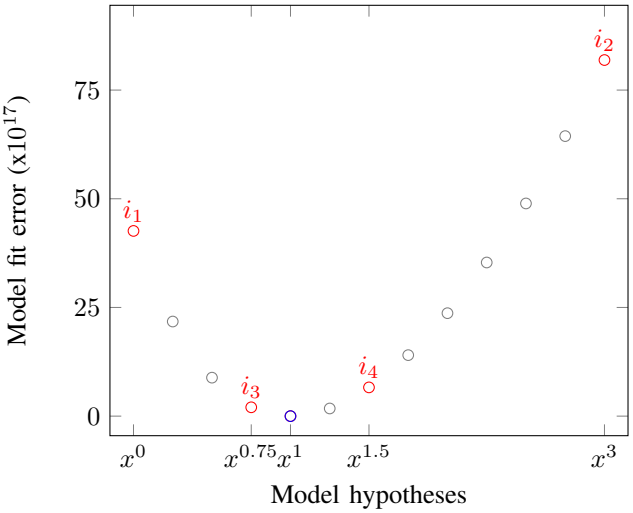

Fig. 2: Model fit error for different model hypotheses. The fit error of model hypotheses decrease towards the one with the smallest error, in this case x^1. The figure further shows the golden-section-search interval reduction. The search interval starts as $[i_1, i_2]$ and becomes $[i_1, i_4]$ after one step of the golden section search method.

Following this observation, we sort all hypotheses by their slopes at the measurement with the largest parameter value. In the particular example from above, this is trivial, as all hypotheses are simple monomials and thus their order (for any value) is the ascending order of the exponent. In the case of more complex hypotheses the order may change depending on the measurement chosen. We select the measurement with the largest parameter value, as users are most often interested in understanding and predicting the behavior at and beyond the upper range of a given parameter.

As an example, we will attempt to model the effect that varying the group number has on floating point instructions in the `LTimes` kernel of the Kripke application. We will consider the following pairs of parameter x and measurement t as in input: $(32, 1209.6)$, $(64, 2419.2)$, $(96, 3628.8)$, $(128, 4838.4)$, $(160, 6048)$. Figure 2 shows the residual sum of squares error of various model functions fitted via the least squares method to our five data points. The x^1 hypothesis is a perfect fit without error, and the error of the model fit as a function of its rank (the index in the sorted hypothesis list) has a minimum at the index of the best model. Since this minimum is also unique, this function is unimodal in the analyzed range.

1) Modified golden section search: The observation above justifies a modified golden section search as a means to traverse the model hypothesis search space. This method is a way to quickly narrow down the range of values in which the extremum of a unimodal function is found. Starting with the complete search space, we recursively refine the interval in which the extremum can be found as follows: we first divide the total search space into subintervals by choosing two additional points between the extreme points of the interval. For optimal performance, the points in the interval are selected using the golden ratio $\phi = 1.618$. We then evaluate the model fit at all four points and from there pinpoint the interval that contains the extremum. We then repeat the same approach on this interval recursively, until only one hypothesis remains.

As an example, a step of the method is displayed in Figure 2 using data from the `LTimes` kernel of Kripke: the two end points of the interval $[i_1, i_2]$ and one point in the interval, i_3, such that $\frac{i_2 - i_3}{i_3 - i_1} = \phi$. In this situation $e(i_1) > e(i_3) < e(i_2) \wedge e(i_1) > e(i_2)$. A new point i_4 is then chosen in the interval $[i_3, i_2]$ using the same ϕ as before. Indices only take integer values, so the i_3 and i_4 must be rounded before the hypothesis error function e can be evaluated.

The evaluation of i_4 indicates where the search should continue. If $e(i_4) \geq (i_3)$ due to the monotonicity of e the minimum cannot be in the interval $[i_4, i_2]$. Therefore the search has to be continued in the interval $[i_1, i_4]$. Should $e(i_4) < e(i_3)$ the search must be continued in $[i_3, i_2]$. After a finite number of recursive search interval contractions only one hypothesis can be selected, and that will be the one with the optimal fit out of all hypotheses available in the search space.

2) Limitations: If the true function we are trying to model has a behavior very different from what can be modeled based on the normal form then it is possible that the above observation no longer holds. Examples include discontinuous functions and functions with multiple behaviors depending on the parameter values. While we have not encountered such cases so far, they can occur and in the worst case a model which is not the model with the best fit could be selected. Nevertheless, if a model has a large fit error, an unsatisfying value for $\hat{R}^2$ would alert the user before he could draw any wrong conclusions.

3) Benefits: Golden section search allows the model hypothesis space to be searched faster. The dependence between the cardinality of the hypotheses set and the number of steps needed to find the best model goes down from linear to logarithmic. The benefits therefore increase the larger the search space becomes. For example, in the case of the single parameter search described in Section III-B, which created a search space of 24,804 candidates, the number of steps required drops to 25, a reduction of almost three orders of magnitude.

The advantage of the golden section search over similar approaches, such as ternary search, is the reuse of previous measurements. At any given step only one new point has to be evaluated. Needing as few such evaluations as possible is crucial, as this is a computationally intensive part of the process.

B. Combining Multiple Parameters

Our approach for multi-parameter modeling is based on the assumption that the best single parameter models for each individual parameter form the best multi-parameter model together, only their combination is unknown. This is—just like the previous assumption—intuitive: If the best model for the process count is $c_1 \cdot \log x_1$ and the best model for problem size is $c_2 \cdot {x_2}^2$ we expect that the best multi-parameter model will either be $c_3 \cdot \log x_1 \cdot {x_2}^2$ or $c_4 \cdot \log x_1 + c_5 \cdot {x_2}^2$ depending on whether the effects of the two parameters are combined or independent of each other. We do not expect it to be $c_6 \cdot {x_1}^3 \cdot \sqrt{x}_2$, or any other model unrelated to the best single parameter models.

1) Hierarchical search: Using the assumption above, we first obtain single parameter models for each individual parameter using the golden section search method previously described. Once we have these models, all that is left is to compare all additive and multiplicative options of combining said models into one multi-parameter model and choosing the one with the best fit.

The size of the search space for this approach is as follows: given m parameters and one n-term model for each of them. We must combine all subsets of terms of each single-parameter model with each subset of terms of each other single parameter model. The number of subsets of a set of n elements is 2^n, so the total size of the search space is $2^{n \cdot m}$.

Again using the example from Section III-B, assuming the single-parameter models for all three parameters have been computed and that all models have three terms each (the worst case scenario for search space cardinality in this case), the number of hypotheses that have to be tested is $2^{3*3} = 512$. Adding the 3 times 25 steps needed to generate the single-parameter models, we will need to look at most at 587 models to find the best fit, compared to the $6.51 \cdot 10^{14}$ in the unoptimized approach.

2) Discussion: The total size of the search space, $2^{n \cdot m}$, can seem daunting at first, but is insignificant when compared to the unoptimized search space. Considering that two terms have proven sufficient to successfully model applications, the search space becomes 4^m. While the number of parameters cannot be arbitrarily large, our approach practically removes model generation as a bottleneck, as collecting sufficient experimental measurements will prove impractical long before our modeling approach will experience any issues.

C. Data Collection

To create multi-parameter models, we need to have sufficient input data that allows accurate single-parameter models for all parameters to be generated, as required by the hierarchical search described in Section IV-B1. For this reason, the set of parameter assignments used in experiments must be symmetric. Assume each of v measurements is a tuple of $(x_{1,i}, \ldots, x_{m,i}, t_i)$, consisting of m input parameter values plus the metric t of interest (e.g., the completion time). Then

TABLE I: Evaluation of heuristics using synthetic functions.

Search type	Heuristic	Exhaustive
Optimal models identified	95,480 [95.5%]	96,120 [96.1%]
Lead-order term identified (including coefficient)	4,520 [4.5%]	3880 [3.9%]
Lead-order term not identified	0 [0%]	0 [0%]
Modeling time	1.5 hrs.	107 hrs.

symmetric means that for each input parameter x_i there must be a set of k measurements where x_i is varied while all other parameters remain constant. For example, the three-parameter tuple set (1,10,22,t_1), (2,10,22,t_2), (1,11,22,t_3), (2,11,22,t_4), (1,10,44,t_5), (2,10,44,t_6), (1,11,44,t_7), (2,11,44,t_8) is symmetric. The removal of any single tuple would render the set non-symmetric. Symmetry is required by our method because it allows to fix any single parameter and look at it in isolation, considering the other parameters constant. For this, we project out all but one parameter and calculate the average metric value across all tuples with the same assignment for the chosen parameter. For example, if we model the first parameter of the previous example, we would use $(1, 0, 0, (t_1 + t_3 + t_5 + t_7)/4)$ and $(2, 0, 0, (t_2 + t_4 + t_6 + t_8)/4)$ as the basis for the single-parameter model of the first parameter. Overall, this strategy requires a full factorial design of $v = k^m$ measurements if each parameter is tested in k configurations. We empirically observed that $k = 5$ is sufficient in practice. Thus, the number of parameter assignments to be tested is 5^m. Depending on the run-to-run variation, the measurement of each parameter assignment must be repeated up to five times. Therefore, the total number of required measurements is between 5^m and $5^{(m+1)}$.

V. Evaluation of Heuristics

To evaluate the heuristics presented in Section IV, we quantify the speedup of the model search in comparison to an exhaustive traversal of the same search space. Furthermore, we determine the frequency at which our heuristics lead to models that differ from the ones the exhaustive search produces. In those cases where the models we discover are different, we analyze these differences and discuss their impact on the quality of the results. Because traversing the entire search space for three or more parameters is prohibitively time consuming even with a very small number of potential terms, we allow only at most two model parameters for the purpose of this comparison.

The evaluation is divided into two parts. First, we examine how close the models generated both through exhaustive search and with the help of heuristics are to inputs derived from synthetically generated functions. This allows our results to be compared with a known optimal model. Second, we compare the results of both approaches when applied to actual performance measurements of scientific codes, which factors in the effects of run-to-run variation.

A. Synthetic Data

We generate 100,000 test functions by instantiating our normal form from Eq.1 with random coefficients $c_l \in (0, 100)$ and i_l and j_l randomly selected from the sets I and J, obtaining functions of the type represented in Eq. 3.

$$f(x) = c_0 + c_1(x^i \cdot log_2^j(x))^{0|1} \cdot (y^k \cdot log_2^l(y))^{0|1} + c_2(\cdot x^i \cdot log_2^j(x))^{0|1} \cdot (y^k \cdot log_2^l(y))^{0|1} \quad (3)$$

To create the input sets for our model generator, we evaluated the functions at 5^m points with $m = 2$ being the number of parameters. To these inputs, our model generator responded in three different ways:

1) **Optimal models.** The most common result (ca. 95%) is that the heuristically determined model, the model determined through an exhaustive search, and the known optimal model are identical.
2) **Lead-order term and its coefficient identified, smaller term not modeled by either method.** Another scenario is encountered when the optimal model has the form $c_1 \cdot f(x) \cdot f(y) + c_2 \cdot f(y)$, where $c_1 \cdot f(x) \cdot f(y) \gg c_2 \cdot f(y)$ in the considered parameter ranges. The optimal model $100 \cdot x^3 \cdot log_2(y) + 2 \cdot log_2(y)$ is a example of this case. Neither modeling approach is capable of detecting the smaller term and they both only model the lead-order term. The effect on the quality of the resulting models is very small, and an attempt to model such small influences will often lead to noise being modeled instead.
3) **Lead-order term and its coefficient identified, smaller additive term only modeled by exhaustive search.** This behavior appears when the optimal model has the form $c_1 \cdot f(x) + c_2 \cdot f(y)$, where $c_1 \cdot f(x) \gg c_2 \cdot f(y)$. In this case the heuristic approach fails to identify the parameter with a smaller effect. The contribution of one parameter leads to the single parameter model for the other parameter to have a very large constant component. As this constant is larger than the variation caused by the parameter with the smaller effect, the modeling process attributes the variation to potential noise and conservatively selects the constant model. The effect on the quality of the resulting model is again negligible.

Table I displays the number of times the modeling identified the entire function correctly and the times only the lead-order term was identified correctly. The lead-order term was correctly identified in all test cases. The difference in time required to obtain the 100,000 models is significant: 1.5 hours when using the heuristics compared to 107 hours when trying out all models.

B. Application Measurements

In addition to synthetic data, we evaluate our heuristics with three scientific applications: Kripke, Cloverleaf, and BLAST. Below, we briefly describe them along with the input decks used. All tests we report were run on Vulcan, an IBM BG/Q system at Lawrence Livermore National Laboratory

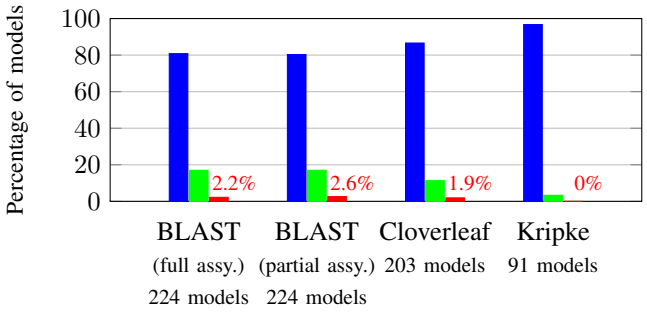

Fig. 3: Comparison of performance models obtained for scientific applications using either our heuristics or a full traversal of the search space. For each application, we show the percentage of times where the resulting models were identical (left bar), where only the lead-order terms and their coefficients were the same (center bar), and where the lead order terms were also different (right bar).

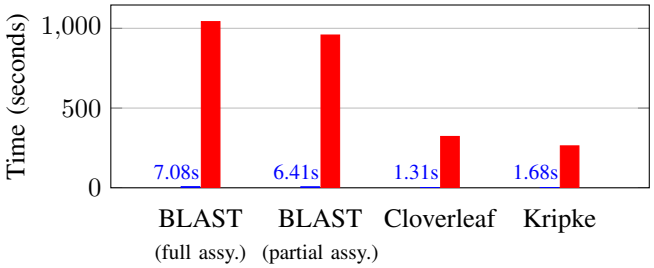

Fig. 4: Time required to obtain performance models of scientific applications via heuristics (left bar) and via exhaustive traversal of the entire search space (right bar).

with 24,576 nodes in 24 racks. Each node is powered by an IBM PowerPC A2 processor with 16 cores/64 hardware threads and features 16 GB of main memory. The system uses IBM's Compute Node Kernel (CNK), as well as an IBM MPI implementation based on MPICH2. We use Score-P [2] to acquire all of our metrics for the chosen applications. In particular, we measured execution time, total number of instructions, number of floating point instructions, and MPI bytes sent and received.

Kripke [11] is an open-source 3D Sn deterministic particle transport code. It calculates angular fluxes and stores them in a flexible hierarchy of data structures (direction sets, group sets, and zones). Kripke was designed as a research tool to explore how data-layouts affect performance, especially on different architectures and with different programming models. For this test, we varied two parameters: the number of directions per set (16, 32, 64, 128, 256, and 512), and the number of groups per set (32, 64, 96, 128, and 160).

BLAST [12] is an arbitrary-order finite-element hydrodynamics research code under development at Lawrence Livermore National Laboratory. It is used to explore the costs and benefits of high-order finite element methods for compressible hydrodynamics problems on modern and emerging architectures. BLAST implements two different algorithmic approaches that produce the same answer: *full assembly*, which assembles and solves a global matrix, and *partial assembly*, which stores only physics data and solves the linear system matrix free. Using the Sedov test problem [15] as input, we kept the number of degrees of freedom in the problem fixed and ran 2D tests varying two parameters: order (1, 2, 4, 8, and 16) and number of MPI ranks (64, 256, 1024, 4096, and 16384) using 64 ranks per node. These tests allowed models for both code options to be produced, considering various parameters of interest to the application team, including FLOP scaling with order, data motion scaling with order, and code scaling with processor count.

CloverLeaf [9] is a 2D structured hydrodynamics mini-application that solves the Euler equations using an explicit, second-order method. It was developed to investigate the use of new programming models and architectures in the context of hydrodynamics. We use a modified version of Sod's shock tube [17] as input problem, where we increase the vertical size of the domain and allow the problem to evolve in both dimensions. We ran CloverLeaf in a weak-scaling configuration, where the problem size per node remains fixed. We vary two parameters: the per-node problem size (1, 2, 4, 8, 16) and the number of MPI ranks (64, 256, 1024, 4096, and 16384) using 16 MPI ranks per node. This allowed models to be produced that capture key concerns for CloverLeaf developers: how problem size and processor count impact the scalability of the application.

Real data sets come with new challenges, such as not knowing the optimal model, and indeed no guarantees that the assumptions required for our method hold, namely that the optimal model is described by one and only one function and that the function is part of the search space. Fig. 3 shows the results of both applying the heuristics and searching the entire solution space. As expected, in the overwhelming majority of cases the two approaches provide the same result (84%), or at least present the same lead-order term (14%). In about 2% of the cases the models differ. The reason is that noise and outliers occurring in real data sets are not limited to any arbitrary threshold compared to the effect of different parameters on performance. The projection used by the heuristics to generate single-parameter models out of multi-dimensional data diminishes noisy behavior to a higher degree than the exhaustive search does. Therefore, in these rare cases, the heuristic approach results in models with a more conservative growth rate than the ones identified through an exhaustive search. The optimal model is not necessarily the one identified by the exhaustive search, as noise could be modeled alongside the parameter effects.

In all three cases, the model generation for an entire application took only seconds (cf. Fig. 4) and was at least a hundred times faster than the exhaustive search. Generating performance models for an entire application means one model per call path and target metric. The search space reduction in all three cases was five orders of magnitude (from 4,250,070 model hypotheses down to 66 per call path and target metric).

C. Discussion

The evaluation with synthetic and real data demonstrates that our heuristics can offer results substantially faster than an exhaustive search—without significant drawbacks in terms of result quality. For three or more parameters, the size of the search space would have prevented such a comparison altogether, which also means that the exhaustive search presents no viable alternative beyond two parameters.

VI. Application Case Studies

In the following, we present the type of insights our approach can deliver. In two case studies, Kripke and BLAST, we look at how performance modeling with multiple parameters can help developers understand and validate the behavior of an application.

A. Kripke

For Kripke we now increase the number of model parameters to three, enabling more complex behavior to be captured. The number of directions per set and number of groups per set are varied as in Section V-B. In addition, we vary the number of MPI ranks (8, 64, 512, 4,096, and 32,768). Each rank has 8 OpenMP threads, which means the rank counts correspond to using 1, 8, 64, 512, and 4,096 nodes on Vulcan, using all 64 hardware threads available on a node. These parameter settings represent a realistic range for actual use cases, while remaining tractable. Although we assume that system noise affects our Blue Gene system to a lesser degree, we repeat each test five times to verify its impact experimentally. We ran 750 tests (150 different parameter settings times 5 repetitions each). We determined the confidence intervals and found that there is little to no noise. Had we have known for sure that the system has little to no noise, 150 measurements would have sufficed. This is also true if measurements are restricted to deterministic countable metrics.

The analysis of Kripke covers three parameters: the number of MPI ranks p, the number of directions per direction set d, and the number of groups per group set g. We are particularly interested in the behavior of the `LTimes`, `LPlusTimes`, and `SweepSolver` kernels, as the combination of these three kernels encapsulate the physics simulated by Kripke. Table II lists selected performance models we generated for these kernels. The `LTimes` kernel computes the spherical harmonic moments of the the angular flux for each element in each group and for each direction. Given that in our weak scaling experiments the number of elements per rank is kept constant, we expect the number of floating-point instructions per rank to remain constant as well. However, we should discover linear relationships with respect to both directions and groups, and their effects to be multiplicative. The model we found is $5.4{\cdot}10^6{\cdot}d{\cdot}g$. $\hat{R}^2 = 1$ indicates that this model is exact. Given the expected availability of floating-point processing power on current and future supercomputers, linear growth is not necessarily a bottleneck, but the combined influence of the two parameters could become challenging. The kernels `LPlusTimes` and `SweepSolver` are structured similarly to the `LTimes` kernel, except that the calculations in their innermost loop are different. Nonetheless, as far as the number of floating-point instructions is concerned, all three kernels belong to the same complexity class.

However, the `SweepSolver` kernel's runtime model is different from the other two kernels: The number of parallel MPI ranks appears in the model. The difference stems from the fact that in addition to floating-point calculations, the `SweepSolver` uses MPI to pass data between ranks and ensures that dependencies between ranks are maintained. Theoretically, the processor count should not affect the number or size of MPI messages sent by each processor, except for a logarithmic term in the message number due to optimizations in the inter-processor communication scheme. The models we have generated are in agreement with this theory and indicate that the $p^{1/3}$ term is caused by waiting on other processors, as shown by the model of the `MPI_Testany` function. The `MPI_Testany` function is called from `SweepSolver` using spin waiting. The $p^{1/3}$ term stems from the three dimensional data decomposition across processes. It represents the diagonal of the process cube, and the waiting time caused by the wavefront traveling along it. The spin waiting causes the performance of these two kernels to compound each other. This is why both kernels show a much smaller $p^{1/3} \cdot d \cdot g$ term, (about 2 orders of magnitude smaller than the lead-order term), representing the interaction caused by the spin-waiting.

Bailey and Falgout [3] show that the theoretical lower bound on the `SweepSolver` kernel for 3D simulations is $\mathcal{O}(p^{1/3} + d \cdot g)$. The key difference between the theoretical lower bound and Kripke's actual runtime performance is the small multiplicative effect caused by the spin waiting. Although the coefficient is quite small, the contribution could become more pronounced at larger configurations.

Since the study above considers three parameters, relying on an exhaustive search would not have been a competitive option. The model generator would have taken more than five hundred years. In contrast, our heuristics-based model generation took less then a minute. This corresponds to a search space reduction of twelve orders of magnitude.

B. BLAST

BLAST [12] has provided us with the opportunity to study the effects of a parameter, the order, that does not define the input problem size and analyze its interaction with the processor count. When used with a fixed number of degrees of freedom, order is independent of problem specification. That is, for a fixed processor count, changing order does not meaningfully change initial conditions of the simulation, nor the resolution of the degrees of freedom within the mesh. Changing order does change the calculations used within the simulation and the flexibility of the mesh (how likely the mesh is to tangle). In general, a higher order increases the number of calculations and increases the flexibility of the mesh. A discussion of balancing the costs and benefits of increasing order are beyond the scope of this paper. We used the same setup and measurements as in Section V-B.

TABLE II: Selected multi-parameter performance models for different kernels of Kripke and BLAST.

Kernel	Metric	Model	$\hat{R}^2$
Kripke			
LTimes	Floating point instr. [10^6]	$5.4 \cdot d \cdot g$	1
LPlusTimes	Floating point instr. [10^6]	$5.4 \cdot d \cdot g$	1
SweepSolver	Floating point instr. [10^6]	$2.16 \cdot d \cdot g$	1
LTimes	Time [seconds]	$12.68 + 3.67 \cdot 10^{-2} \cdot d^{5/4} \cdot g$	0.989
LPlusTimes	Time [seconds]	$9.82 + 9.62 \cdot 10^{-3} \cdot d \cdot g^{3/2}$	0.991
SweepSolver	Time [seconds]	$4.91 + 4.83 \cdot 10^{-3} \cdot p^{1/3} \cdot d \cdot g + 0.90 \cdot d \cdot g$	0.994
MPI_Testany	Time [seconds]	$6.81 + 0.8 \cdot p^{1/3} + 4.76 \cdot 10^{-3} \cdot p^{1/3} \cdot d \cdot g$	0.996
SweepSolver	Bytes sent per msg. = Bytes recv. per msg. [10^6]	$4.8 \cdot d \cdot g$	1
SweepSolver	Messages sent = Messages received	$11250 + 900 \cdot log(p)$	1
BLAST – full assembly			
MPI_Isend	Bytes sent per msg.	$1.95 \cdot 10^4 + 81.8 \cdot \log p \cdot o^{7/4} + 4.62 \cdot 10^3 \cdot o^{7/4}$	0.999
BLAST – partial assembly			
MPI_Isend	Bytes sent per msg.	$7.63 \cdot 10^3 + 1.31 \cdot 10^2 \cdot \log p$	0.871

When modeling the two different algorithms for BLAST, we have gained new insights into how their parallel communication behavior differs. We model the bytes sent and received in non-blocking fashion and display the results in Table II. We observe that both approaches grow logarithmically with the number of processors in weak scaling mode. The order of the solver has no effect on the partial assembly algorithm but a significant effect on the full assembly algorithm, as indicated by the $o^{7/4}$ component. The developer analyzed our result, and did not expect the order to have such a pivotal effect, or that the order should have such a different effect on the two algorithms. These insights will help the developers better run and optimize the code as they are now aware of the additional cost the order has in full assembly mode. This result also showcases the compounding effect on performance that parameters have and the need to understand their interactions.

VII. Related Work

There is a broad spectrum of existing methods and tools to support the creation of performance models. While Hammer et al. [8] and Lo et al. [13] focus on roofline models, Zaparanakus et al. generate simple performance models for sequential algorithms [21]. The PMaC tool suite creates scaling models of parallel applications [14]. Finally Goldsmith et al. apply clustering and linear regression analysis to derive performance-model coefficients from empirical measurements [7]. Although already quite powerful, none of the above methods supports true multi-parameter models. If different parameters are considered at all, then only one at a time or with a fixed relationship between them.

PALM [19] supports the creation of true multi-parameter models but requires the user to annotate the source code with micro-models that apply only to small sections of the code. Following extensive and detailed per-function measurements, the underlying framework then automatically combines these micro-models into structural macro-models. The approach is related to Aspen [18], a dedicated language to specify such micro-models. Our approach provides a higher degree of automation without prior source-code annotation. Vuduc et al. select the best implementation of a given algorithm by automatically generating a large number of candidates and then choosing the one that offers the best performance according to an empirically derived model—potentially with multiple parameters [20]. However, automatic is only the linear regression to determine the model coefficients. The model hypothesis itself must be chosen manually, which is why their approach, as far as performance modeling is concerned, cannot be considered truly automatic.

Finally, there are also automated methods for multi-parameter performance modeling. For example, Siegmund et al. analyze the interaction of different configuration options of an application and model how this affects the performance of the application as a whole [16]. The main difference is the supported diversity of model functions. While they allow only linear, quadratic or logarithmic functions, we allow a flexible combination of polynomials and logarithms. Furthermore, we apply heuristics to traverse the model search space more quickly, which is especially helpful in view of the higher model diversity we provide. Finally, we construct performance models for every function (with calling context) in an application - not just for the application as a whole. This allows optimization efforts to be channeled to where they will be most effective. Hoefler et al. generate multi-parameter performance models online [4]. Although already supported by prior static analysis, the online nature of their approach limits the size of the search space and thus the diversity of models quite significantly, and thus adversely affects model accuracy. Finally, another multi-parameter approach was presented by Jayakumar et al. [10]. They extract execution signatures from their target applications, representing different execution phases, and match them with reference kernels stored in a database. If such a match exists, they use the performance model belonging to the reference kernel to predict execution times for varying numbers of processors and input sizes. If no match can be found for an execution phase, they apply static analysis to derive performance models. Model parameters

include the core count p and only one input-size defining variable n at a time, which the user has to identify manually. The spectrum of model functions is quite small and restricted to n, $log(n)$, n/p, $log(n)/p$.

VIII. Conclusion

We demonstrate that automatic performance modeling with multiple parameters is feasible. Within seconds, we generated accurate performance models for realistic applications from a limited set of performance measurements. The models both confirmed assumption the developers had earlier, providing further validation for our heuristics, and offered new unexpected insights into application behavior. Given that the resources needed for the model generation itself are now negligible, the number of model parameters is only constrained by the amount of measurements a user can afford. From a practical perspective, we believe that this will allow the key parameters of many applications to be captured in empirical performance models.

Speaking in general terms, our method enables a more complete traversal of the performance space compared to other performance analysis methods at relatively low cost, making application performance tuning more effective. This will not only benefit application developers but also the designers of emerging systems, who can now project application requirements more precisely along several parameter dimensions and balance their systems accordingly. Auto-tuning methods that still rely on long series of performance tests, which are not only time consuming but also expensive, can profit as well.

Acknowledgments

A part of this work was performed under the auspices of the U.S. Department of Energy under Grant No. DE-SC0015524 and by Lawrence Livermore National Laboratory under Grant No. DE-AC52-07NA27344. Moreover, support of the German Research Foundation (DFG) and the Swiss National Science Foundation (SNSF) through the DFG Priority Programme 1648 *Software for Exascale Computing* (SPPEXA) as well as the German Federal Ministry for Education and Research (BMBF) under Grant No. 01|H13001G is also gratefully acknowledged.

References

[1] JuBE: Jülich Benchmarking Environment. http://www.fz-juelich.de/jsc/jube.

[2] D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. S. Shende, M. Wagner, B. Wesarg, and F. Wolf. Score-P: A unified performance measurement system for petascale applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, pages 85–97. Gauß-Allianz, Springer, 2012.

[3] T. S. Bailey and R. D. Falgout. Analysis of massively parallel discrete-ordinates transport sweep algorithms with collisions. In *International Conference on Mathematics, Computational Methods & Reactor Physics, Saratoga Springs, NY*, 2009.

[4] A. Bhattacharyya, G. Kwasniewski, and T. Hoefler. Using compiler techniques to improve automatic performance modeling. In *Proc. of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT'15)*, pages 1–12, San Francisco, CA, USA, 2015.

[5] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. Nov. 2013. IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC13).

[6] D. S. Carter. Comparison of different shrinkage formulas in estimating population multiple correlation coefficients. *Educational and Psychological Measurement*, 39(2):261–266, 1979.

[7] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring Empirical Computational Complexity. In *Proc. of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 395–404, New York, NY, USA, 2007. ACM.

[8] J. Hammer, G. Hager, J. Eitzinger, and G. Wellein. Automatic loop kernel analysis and performance modeling with kerncraft. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, PMBS '15, pages 4:1–4:11, New York, NY, USA, 2015. ACM.

[9] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Accelerating Hydrocodes with OpenACC, OpeCL and CUDA. In *Proceedings of the 3rd International Workshop on Performance Modeling, Benchmarking and Simulation*, pages 465–471, Nov. 2012.

[10] A. Jayakumar, P. Murali, and S. Vadhiyar. Matching application signatures for performance predictions using a single execution. In *Proc. of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2015)*, pages 1161–1170, May 2015.

[11] A. J. Kunen. Kripke - user manual v1.0. Technical Report LLNL-SM-658558, Lawrence Livermore National Laboratory, August 2014.

[12] S. Langer, I. Karlin, V. Dobrev, M. Stowell, and M. Kumbera. Performance analysis and optimization for blast, a high order finite element hydro code. *Proceedings of the 2014 NECDC*, 2014.

[13] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker. Roofline Model Toolkit: A practical tool for architectural and program analysis. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 129–148. Springer, 2014.

[14] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole. Modeling and Predicting Performance of High Performance Computing Applications on Hardware Accelerators. *Int. J. High Perform. Comput. Appl.*, 27(2):89–108, May 2013.

[15] L. I. Sedov. Propagation of strong shock waves. *Journal of Applied Mathematics and Mechanics*, 10:241–250, 1946.

[16] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 284–294, New York, NY, USA, 2015. ACM.

[17] G. A. Sod. A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws. *Journal of Computational Physics*, 27(1):1–31, Apr. 1978.

[18] K. L. Spafford and J. S. Vetter. Aspen: A domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 84:1–84:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[19] N. R. Tallent and A. Hoisie. Palm: Easing the Burden of Analytical Performance Modeling. In *Proc. of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 221–230, New York, NY, USA, 2014. ACM.

[20] R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical Models for Empirical Search-Based Performance Tuning. *Int. J. High Perform. Comput. Appl.*, 18(1):65–94, Feb. 2004.

[21] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. *SIGPLAN Not.*, 47(6):67–76, June 2012.