Systematic Adoption of Genetic Programming for Deriving Software Performance Curves

Michael Faber
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
mail.michael.faber@gmail.com

Jens Happe
SAP Research
Karlsruhe, Germany
jens.happe@sap.com

ABSTRACT

Measurement-based approaches to software performance engineering apply analysis methods (e.g., statistical inference or machine learning) on raw measurement data with the goal to build a mathematical model describing the performance-relevant behavior of a system under test (SUT). The main challenge for such approaches is to find a reasonable trade-off between minimizing the amount of necessary measurement data used to build the model and maximizing the model's accuracy. Most existing methods require prior knowledge about parameter dependencies or their models are limited to only linear correlations. In this paper, we investigate the applicability of genetic programming (GP) to derive a mathematical equation expressing the performance behavior of the measured system (software performance curve). We systematically optimized the parameters of the GP algorithm to derive accurate software performance curves and applied techniques to prevent overfitting. We conducted an evaluation with a representative MySQL database system. The results clearly show that the GP algorithm outperforms other analysis techniques like inverse distance weighting (IDW) and multivariate adaptive regression splines (MARS) in terms of model accuracy.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; C.4 [**Performance of Systems**]; I.6.5 [**Simulation and Modeling**]: Model Development

General Terms

Performance Analysis, Performance Prediction, Genetic programming, Measurement-based

Keywords

Software Performance Engineering, Model Inference, Machine Learning, Black-box Approach

ICPE'12, April 22-25, 2012, Boston, Massachusetts, USA

1. INTRODUCTION

Software performance is an important quality attribute and directly influences the total cost of ownership (TCO), customer satisfaction and even the productivity of employees [23]. Thus, software performance is crucial in today's enterprise systems. Methods of software performance analysis help architects to detect bottlenecks and performance problems and to judge different design alternatives. For this purpose, software performance models are created to predict the behavior of software systems. Besides analytical, model-based and prototype-based approaches, the measurement-based approach takes measurement data and uses analysis methods, such as statistics or machine learning, to extract performance-relevant factors of the system under test (SUT). Those measurement-based approaches do not necessarily require an understanding of the system internals but often consider the SUT as a black box. This makes them well-suited for large enterprise systems that are historically grown and not implemented from scratch [24]. Moreover, they are often applied to existing systems, such as legacy systems or services provided by third parties, for which either no internals are known or no internals are to be modeled.

The main goal of this work is to apply machine-learning techniques in order to infer accurate performance models based on measurement data. Those models are hereafter referred to as *software performance curves*. They describe the performance behavior of a system in dependence of the system's configuration and the usage profile. As mentioned above, system performance depends on many factors, leading to a high dimensionality of the problem. This *curse of dimensionality* [11] leads to sparse measurements of the parameter space. To reduce the measurement errors and stabilize the observed metrics, all measurements have to be repeated sufficiently. Thus, meaningful measurement results are very expensive and only a very limited number of measurements are available for model inference. Analysis methods must find a reasonable trade-off between minimizing the amount of necessary measurement data used to build the model while maximizing the model's accuracy. Besides appropriate analysis methods, the process of efficiently selecting new measurement points is crucial for receiving accurate performance curves. This aspect is investigated in more detail in another work [28] of our research group.

Existing approaches use analysis methods such as MARS [5], piecewise polynomial regressions [15] or genetic optimization [10] to infer software performance curves. Problems of these techniques are the assumptions about the in-

put/output parameter dependencies or the resulting model accuracies. Genetic optimization and polynomial regression assume predefined structures for the input/output parameter dependency, expressed as mathematical equations, and focus on optimizing the coefficients of the equation. As this structure is not always known in practice, MARS tries to overcome this issue by approximating the dependency through combinations of piecewise-defined linear functions. However, this approach has problems when approximating functions which contain higher-order powers [11].

In this paper, we examine the ability of a machine learning technique, genetic programming (GP), to infer software performance curves. In contrast to genetic optimization, where a structure is already fixed (e.g., linear), GP does not make any assumptions about the input/output parameter dependency and optimizes the structure of the equation simultaneously with the coefficients (symbolic regression). We employed the Goal/Question/Metric (GQM) approach to derive a systematic plan for the optimization of the configuration of the genetic algorithm. This optimization is important when applying genetic algorithms to a certain problem domain (here the derivation of software performance curves). Then, we followed a four-stepped process to answer each question of the GQM plan. This process consists of i) the experiment definition, ii) experiment execution, iii) experiment analysis and iv) decision. The analysis step includes the application of statistical tests (Kruskal-Wallis Rank Sum Test and Wilcoxon Rank Sum Test) to choose the best alternative. To increase model accuracy and convergence speed, we experimented with providing domain knowledge (such as hypothesis about parameter dependencies) to the GP algorithm. However, the experiments showed that using domain knowledge has no significant influence on the resulting models. To improve the generalization of the result models, we applied techniques to prevent overfitting.

The evaluation contains a synthetic function and representative measurements of a MySQL database system. We compared the models created by the GP algorithm with those created by MARS and IDW. As training sets, we used randomly-distributed and equidistantly-distributed subsets of different sizes. The results for this evaluation reveal that GP outperforms other analysis techniques like MARS and IDW, not only in terms of the average relative error but also the dispersion of relative errors among single predictions is smaller. The main contributions of this paper are i) a generic and systematic approach to optimize and apply GP to any specific problem domain, ii) the adaption and implementation of GP to derive software performance curves, and iii) an evaluation using a synthetic function and measurements of a MySQL database to demonstrate the potential of the GP approach in contrast to other analysis techniques (MARS and IDW).

The remainder of the paper is structured as follows. Section 2 presents an overview of related work about measurement-based performance analysis and GP approaches. In Section 3, we provide foundations about the Software Performance Cockpit (SoPeCo) and GP. Section 4 describes our approach to adopt GP for the derivation of software performance curves. In Section 5, we evaluate the approach using a synthetic function and measurement data from a MySQL database. Our approach is discussed in Section 6. Finally, Section 7 concludes this paper.

2. RELATED WORK

The work related to our approach can be classified in the main categories of software performance engineering and genetic programming. First, we present approaches concerning measurement-based software performance analysis and model derivation. Second, we describe relevant work in the area of genetic programming.

The approaches in software performance engineering can be coarsely divided (according to [29]) into model-based (see [2] and [13] for detailed surveys) and measurement-based approaches (e.g., [5, 14, 15, 21]). In most measurement-based approaches, statistical inference or machine learning techniques are applied to derive predictions based on the measurement data. Courtois and Woodside [5] apply regression splines such as MARS to derive models and introduce a metric to determine the accuracy of the models. This allows to iteratively choose new measurement point using repelling forces until a desired model accuracy is reached. Lee et al. [15] compare polynomial regression and artificial neural networks (ANN) and suggest methods such as hierarchical clustering and correlation analysis to select the most relevant inputs. Their experiments revealed that both techniques lead to models with similar accuracies but differ in terms of their assumptions and the model transparency. A similar result is shown by the comparison in [19]: Psichogios et al. compared MARS with neural networks and found that "MARS is often more accurate and always much faster than neural networks". For a fair comparison of GP and ANN the latter requires similar adjustments in tuning like GP which is beyond the scope of this work. Thus, we chose MARS instead of ANN for the evaluation in Section 5. Sharma et al. [21] apply a machine learning technique, namely independent component analysis (ICA), to categorize workload requests and to identify their resource demands using only high-level measurement results (e.g., CPU/network usage or overall request rate). Zheng et al. [30] employ Kalman Filter estimators to track parameters which cannot be measured directly by using easy observable data such as response times. Kraft et al. [14] estimate service demands by applying linear regression and the maximum likelihood technique using only response time measurements. This approach avoids detailed instrumentations to receive samples for service demands.

In this part, we present approaches about genetic programming which compare different alternatives for generating constants, fitness functions, crossover operators and preventing overfitting. We used these approaches as a basis for our optimizations of the GP algorithm. Ryan and Keijzer [20] investigate the effects of different constant mutation types in the problem domain of symbolic regression. They state that, for symbolic regression, constant mutation types must find a good balance between the effort needed for the mutation and the probability of rejecting the whole individual in future generations. Ryan and Keijzer compare four different mutation types and evaluate their influence on the overall performance of the GP algorithm. Their experiments reveal, that the decision which operator performs best depends on the complexity of the problem which should be approximated. We used this approach as a basis and reproduced the experiments for the domain of deriving software performance curves. Ferrucci et al. examine the influence of different fitness functions [8] and conclude that the choice of an appropriate fitness function is crucial as it

leads the whole evaluation process. Our experiments also reveal significant differences when comparing different fitness functions. Gustafson et al. suggest a way to improve a GP approach for the symbolic regression domain in [9]. They investigate the dissimilarity and diversity of the solutions during the evolutionary process. Finally, they suggest to prevent crossovers between parents having the same fitness value. Their experiments showed significant improvements after applying this improvement. However, in our experiments, we saw no improvements for the adjusted crossover operator. Panait and Luke compare six alternatives on how to evolve robust programs with GP [18]. Robust programs are solutions which generalize correctly from the learning data. During their experiments, they identified three methods which perform significantly better than the others for the domain of symbolic regression. We applied one of these methods (random per generation) and combined it with a cross-validation approach to prevent the effects of overfitting.

3. FOUNDATIONS

This section places our work in the context of the Software Performance Cockpit and introduces the concepts of machine learning and genetic programming.

The Software Performance Cockpit (SoPeCo) [26, 27] is a framework with the goal to make measurement-based software performance engineering more practicable. Enterprise software systems are usually quite complex and their performance depends on a variety of influencing factors. Different parts and layers of the system (e.g., operating system or middleware) require detailed knowledge and mostly separate tools for instrumentation and monitoring. The SoPeCo handles this challenge through a plug-in-based architecture which allows the encapsulation of domain knowledge and implementations within adapters. Westermann et al. defined three different responsibilities and tasks: The *System, Benchmark and Tool Experts* develop adapters for generating load and connecting parts of the software systems such as middleware or monitoring tools with the SoPeCo. *Analysis Experts* provide adapters which enable various statistical analysis of measurement results. *Performance Analysts* model the test scenario using the configuration meta-model provided by the framework. This includes aspects such as the use of different adapters (e.g., workload driver, monitoring) and providing system information where the adapters are deployed. For further information about the SoPeCo, we recommend [26] and [27].

When executing the configuration model created by the Performance Analyst, the SoPeCo automatically triggers systematic measurements and collects the reported metrics such as response time. A subsequent analysis adapter derives a *software performance curve* [28] which describes dependencies between system's configuration, its workload and the performance expressed through timing behavior, throughput or resource utilization. A performance curve might be realized through mathematical equations expressing these dependencies: Assuming that x_1 and x_2 are two performance-relevant factors acting as independent variables (inputs) and *responseTime* the dependent variable (output). A random performance curve might then be defined as $\hat{f}(x_1, x_2) = 2 * x_1 + 0.1 * x_2^2$. Such curves can be derived through statistical or machine learning techniques. Within this work, we applied genetic programming, as a common machine learning technique, for the derivation of software performance curves.

Machine learning (ML) in general refers to the process of deriving knowledge from given training data. ML techniques create prediction models which estimate the outcome for a given set of features. Depending on the training data, ML distinguishes supervised and unsupervised learning techniques. Unsupervised techniques only use the features and no outcome variables for the training. Mostly, these techniques are used to cluster data. Supervised techniques require pairs of features and outcome variables for the training. Features have different types such as quantitative, qualitative or ordered categorical. The prediction of a quantitative output is called regression, whereas the prediction of qualitative output is often referred to as classification [11]. Due to the quantitative nature of performance metrics, we focus on regression techniques.

One field of machine learning are *evolutionary algorithms* (EA) which are approaches to solve optimization or search problems. EAs use an iterative approach to approximate an optimal solution. During each iteration (*generation*), the *population*, consisting of a certain number of *individuals*, evolves. This evolution is performed by *reproducing*, *mutating* and *crossing-over* individuals of the previous generation. Each individual represents a candidate solution and has a *fitness* value expressing the quality of the solution. The aim of EAs is to maximize this fitness over many generations. The main principles mentioned above and the concept of "survival of the fittest" (as described by Charles Darwin) are copied from nature. For more information, we recommend [4] and [7]. *Genetic programming* (GP) is a special application of EA and aims at deriving computer programs or mathematical equations. The individuals in GP are usually represented as tree structures and recombinations are tree operations such as randomly exchanging subtrees between two trees [12].

4. APPROACH

In this section, we present our approach to meet the challenge of deriving accurate software performance curves from measurements. In Section 4.1, we depict the overall idea of applying GP for the derivation of software performance curves and provide a simple illustrative example. In Section 4.2, we describe the initial configuration of the GP algorithm, formalize the training set and provide three definitions describing the model error. Next, we present all investigated aspects to adopt the GP algorithm to the specific problem using GQM plans (Section 4.3) and describe a generic process for the adoption of GP algorithms (Section 4.4). Finally, we provide a detailed example to show how we applied the process to systematically fulfill all goals defined in the GQM plan (Section 4.5).

4.1 Overview and Idea

The aim of our approach is to derive accurate software performance curves using measurement data. Since the retrieval of measurement data is very expensive in terms of time and effort, it is desirable to have an algorithm which is capable of deriving accurate performance curves using a small amount of data. To derive the performance curve, we use genetic programming which does not make any assumptions about the input/output parameter dependency

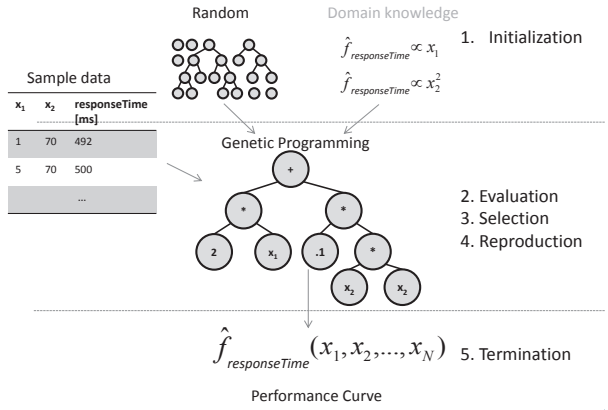

Figure 1: Overview of our approach

and optimizes the structure of the equation simultaneously with the coefficients (symbolic regression).

Figure 1 depicts the idea of our approach that applies genetic programming to software performance engineering. In the first step, GP is initialized with randomized data. The initialization may be improved by domain knowledge or prior statistic analyses. We investigated whether and how domain knowledge can be reused to improve the convergence speed and accuracy of our algorithm. However, our experiments inducting domain knowledge did not influence the results. Similarly, we evaluated the influence of prior statistical analyses that serve as input for the initialization. In our experiments, we did not observe any improvement in terms of convergence speed or model accuracy by these means. After the initialization, the genetic algorithm begins to evolve the individuals. The evolution starts with an evaluation of individuals by using the measurement data (Step 2). Then, the algorithm selects and reproduces fit individuals (Step 3 and 4) and repeats steps 2-4 for a given number of iterations (generations). Finally, the algorithm terminates (Step 5) when a given termination criteria, such as the desired accuracy level or runtime constraints, are fulfilled. The result of the algorithm is a software performance curve expressed through a mathematical equation.

Returning to the example provided in the previous section, we illustrate the genetic programming approach. The goal of the GP algorithm is to find the software performance curve $\hat{f}(x_1, x_2)$, which predicts the dependent variable *responseTime* using provided measurement data. Assume that the algorithm receives results of 50 independent response time measurements with different input configurations (values for x_1, x_2). To evaluate the fitness of each individual, the algorithm calculates the averaged relative error based on the provided training data (see Section 4.2). New individuals are created by recombining the genes (represented as trees) of two individuals. The trees comprise *operators* (e.g., +, -, *, /) serving as inner nodes and *constants* and *variables* (here x_1, x_2) serving as leaves. When the evolution of individuals finishes, the algorithm returns the fittest individual representing the software performance curve identified by the algorithm. The exemplary individual in the center of Figure 1 depicts one possible representation for the software performance curve ($\hat{f}(x_1, x_2) = 2 * x_1 + 0.1 * x_2^2$) in the internally-used tree representation.

4.2 Background

In this section, we present the initial setting of the GP algorithm which we used as a starting point for our optimizations. We also formalize the training set and introduce three definitions expressing the error of inferred models that we used as fitness functions and for judging the model quality.

The idea and concepts of evolutionary algorithms (e.g., GP) are very intuitive and general but its configuration is not. For example, the size of the population and amount of generations, the selection of parent individuals, probabilities for crossover and mutation, and the fitness function highly influence the efficiency of the algorithm (in terms of solution quality and convergence speed) and therefore must be chosen carefully. While some rules of thumb exist on how to set parameters for broad problem domains (e.g., in [12]), many systematic experiments are necessary to adjust the configuration towards a certain problem. We based the initial settings for the configuration parameters of the GP algorithm on the well-established suggestions by Koza [12] about applying GP to symbolic regression. The population had a size of $M = 1024$ and was evolved for $G = 51$ generations. The maximum tree depth was restricted during the run to $D_c = 17$ and during the initialization to $D_i = 6$. As function set we used arithmetic operations $(+, -, *, /)$. Probabilities for the crossover operator was set to $p_c = 0.9$ and for reproduction to $p_r = 0.1$. As a method for parent selection, we used the tournament selection with $k = 7$. In tournament selection, a group of k individuals is randomly selected and pair-wisely compared. The fittest individuals of two tournaments are chosen for the crossover. We used no structural mutation and selected inner nodes for crossovers with a probability of $p_{ip} = 0.8$.

The training set with N samples for a d-dimensional problem can be formulated as an $N \times (d+1)$-matrix $\mathbf{T}$, where the first d columns of each row represent the input parameters and the $d+1$-th column contains the corresponding target metric (e.g., response time). Let $\vec{x_i}$ denote the transposed vector containing columns 1 to d of the i-th row of matrix $\mathbf{T}$ and thus representing the independent parameters for one tuple of training data. The sum of absolute residuals (SAR) based on the training set $\mathbf{T}$ can be used as a fitness measure and is defined as:

$$SAR_{\hat{f}}(\mathbf{T}) = \sum_{i=1}^{N} |\hat{f}(\vec{x_i}) - t_{i,d+1}| \quad . \tag{1}$$

with $\hat{f}$ being the prediction model, $\vec{x_i}$ the vector containing the input parameters and $t_{i,d+1}$ the corresponding target metric.

The relative error is defined as

$$RE(y, \hat{y}) = \begin{cases} |\frac{y-\hat{y}}{y}|, & \text{if } y \neq 0 \\ |y - \hat{y}|, & \text{otherwise} \end{cases} \quad , \tag{2}$$

where y is the measured value and $\hat{y}$ is the predicted value. For the case that the measured value is 0, we used the absolute error as an approximation for the relative error.

We can now define the averaged relative error as

$$ARE_{\hat{f}}(\mathbf{T}) = \frac{\sum_{i=1}^{N} RE(t_{i,d+1}, \hat{f}(\vec{x_i}))}{N} \quad , \tag{3}$$

where $\mathbf{T}$, $\vec{x_i}$, N and $\hat{f}$ are defined as above.

Part I: Parameter Optimizations
G1: Constants
G2: Fitness Function
G3: Multi-Dimensional Regression Problems
G4: Training Sets
G5: Extended Function Set
G6: Crossover
G7: Population Size and Number of Generations
Part II: Domain Knowledge
G8: Influence of One Domain Function
Part III: Overfitting
G9: Apply Efficient Technique to Prevent Overfitting

Table 1: Overview of all addressed goals

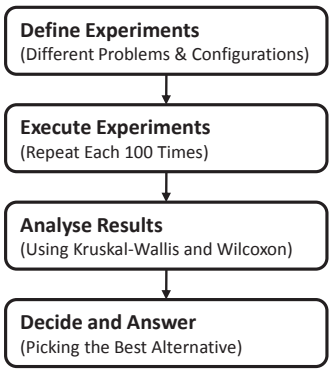

Figure 2: Process to answer questions through experiments

4.3 Examined Goals

In this section, we present the different aspects of the GP algorithm which we adapted and optimized. We used a Goal/Question/Metric plan for a systematic planning of all experiments.

The Goal/Question/Metric (GQM) [3] is a framework for systematic experimentation in software engineering. For this purpose, it defines a process of answering well-defined goals in a top-down fashion. The contribution of the framework is twofold: It helps to derive experiments in a goal-oriented manner and subsequently allows the systematic evaluation and interpretation of their results in order to answer the overall goals.

Table 1 depicts the goals we investigated within this work. The nine goals are organized in three groups: The first group summarizes all goals concerning parameter optimizations of the GP algorithm. To answer the question assigned to these goals, we used the process described in the Section 4.4. The second part investigates the introduction of domain knowledge. Overfitting, as a common problem in the machine learning domain, is addressed in part three.

The accuracy of performance curves highly depends on finding accurate coefficients. An appropriate generation and mutation of constants is addressed Goal G1. Furthermore, the fitness function is essential in genetic algorithms since it steers the whole evolutionary process. Goal G2 addresses the selection of an appropriate fitness function. Software performance is influenced by factors like the system's usage and configuration. Hence, Goal G3 investigates the ability of the algorithm to solve multi-dimensional regression problems. The training data is the only information for algorithm to build the performance models. Goal G4 investigates the influence of necessary preprocessing steps and the training set size. The expressiveness of the algorithm depends on the available operators (inner nodes) to build the models. Goal G5 was to extend the arithmetic function set $(+, -, *, /)$ with other functions such as `pow` or `log`. The crossover operator is responsible for building new individuals during the evolutionary process. Gustafson et al. improved the solution quality in symbolic regression domains by introducing constraints when picking the parent individuals for the crossover [9]. Goal G6 applies these algorithmic improvements to our approach. Goal G7, being the last goal concerning the parameter optimizations, investigates a reasonable population size and number of generations. After the adjustment of the GP algorithm, we enhanced it to use domain knowledge (Goal G8), such as parameter dependencies that are typical for software performance or queuing formulas. Overfitting is a common problem in all fields of machine learning which also needs to be addressed in our scenarios. Thus, we applied and evaluated different techniques to prevent overfitting (G9).

4.4 Generic Process for Adoption of GP Algorithms

In this section, we describe the systematic process which we used for answering the questions derived in the GQM plan. Even though the process is intuitive it is fundamental to describe the decision mechanism detailed since our adjustments of the GP algorithm and thus one of the main contribution of the paper are based on this process. The process is depicted in Figure 2 and described in the following.

Define Experiments.

The first step for answering a question about a reasonable parameter setting is the definition of experiments. The definition basically includes two parts. The first part is the selection of appropriate prediction problems and training sets. It is important, that the chosen prediction problems and training sets are representative for real problems in the target domain for which the GP algorithm should be optimized. The second part is the identification and implementation of different alternative configurations for the investigated question (e.g., different constant mutation types).

Execute Experiments.

The second step is the execution of the experiments. Since the GP approach is non-deterministic, we had to repeat all experiments for a sufficient number of times (n). This number is a trade-off between the execution time for the experiments and stabilized results. Using an initial experiment and the calculation of confidence bands we determined $n = 100$ as sufficient for our experiments. We observed that some evolutions lead to models with very high errors. Such outliers are common in symbolic regression. In order, to keep the experiment runtime to evaluate the effect of changes to the GP algorithm in a reasonable scope, we decided to remove the worst 10% of each 100 runs before analysing the

results. When the GP algorithm is applied to a real problem, it will be executed repeatedly and thus removing the outliers during our experiments does not diesturb the results. All boxplots and statistical tests are based on the cleaned data. Due to the non-determinsm, we implemented the final analysis adapter such that it internally repeats the GP algorithm for x times and only returns the model of the best run.

Analyze Results.

The third step is the analysis of the experiment results. For this purpose, we use statistical tests to determine if the target metric (mostly the averaged relative error of the models) is significantly influenced by different configuration alternatives. The analysis results did not follow a normal distribution (also observed in [18]) and thus we used the Kruskal-Wallis Rank Sum Test and Wilcoxon Rank Sum Test [6]. They either lead to the decision that significant differences in the target metric exist or that the alternatives do not significantly influence the target metrics.

Decide and Answer.

The last step is the decision for one of the questioned alternatives based on the results of the statistical tests. After we answered one question, we picked the next from the GQM plan presented in the section above. We neglected mutual dependencies between configuration parameters of the GP algorithm and sequentially answered the questions.

We suggest this four-stepped process in combination with a GQM plan as a reasonable approach for optimizing the parameters of genetic programming algorithms in order to apply GP to a specific problem domain.

4.5 Detailed Investigation of One Question

In this section, we illustrate the process shown in Figure 2. We focus on Goal G1, which addresses the generation of constants. We derived five questions concerning the constant types, the ranges and their mutation types and probabilities:

- *Q1:* Which mutation type is the best for integer constants?
- *Q2:* Which mutation type is the best for float constants?
- *Q3:* How does the combination of both constant types (integer and float) perform?
- *Q4:* What is the optimal mutation rate?
- *Q5:* Are the constant intervals sufficient to generate big constants?

In the following, we address the first question of the list above to present the application of the process comprising experiment definition, experiment execution, result analysis and decision.

Concerning constant mutation, Koza [12] assumes that new constants are created during the evolution as combinations of other existing constants and functions. Thus, no mutation was considered, and all constant values remained unchanged after their initialization during the whole evolutionary process. Ryan and Keijzer on the other hand investigated the influence of different constant mutation types [20] (see Section 2). We based the three investigated mutation types on the ideas of Ryan and Keijzer.

Formula
$f_0(x_0) = 12 * x_0$
$f_1(x_0) = -41 * x_0$
$f_2(x_0) = 24 * x_0 + 3$
$f_3(x_0) = 54 * x_0^2 - 34 * x_0 + 7$
$f_4(x_0) = \frac{53}{97} * x_0^2$

Table 2: One-dimensional problems with focus on integer coefficients

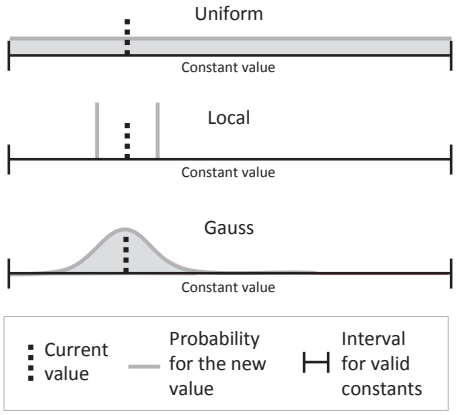

Figure 3: Investigated alternatives for the mutation of constants

4.5.1 Definition of Experiments

The experiment definition (see Section 4.4) includes the identification of representative problems and of different alternatives.

Problems.

We used the five problems listed in Table 2 for running this experiment. All of them have one independent parameter and contain only integer constants as coefficients. We intentionally used simple structures which we assumed likely to be found in early generations. This would mean, that the GP algorithm only has to find the correct coefficient, since the structure is already found. The training sets for each Problem k of Table 2 consist of 150 samples with the input parameters $\mathbf{X} = (0, 1, \ldots, 149)^T$. The i-th row of the 150×2-matrix $\mathbf{T_k}$ had the form $(x_i, f_k(x_i))$. The fact that the training data is distributed equidistantly over an interval (here [0,149]) is a realistic assumption, since systems are often measured in such a systematic way. We also consider the simple linear or quadratic structure of the problems as representative and common for dependencies in performance analysis.

Alternatives.

We compared four different alternatives against each other: no mutation (N), uniform mutation (U), local mutation (L) and Gaussian mutation (G). Figure 3 depicts all mutation types. The gray line illustrates the probability for a value after the mutation and the bold dashed line represents the constant value before the mutation. The uniform mutation generates a new random number with equal probabilities in the entire interval, independently of the current

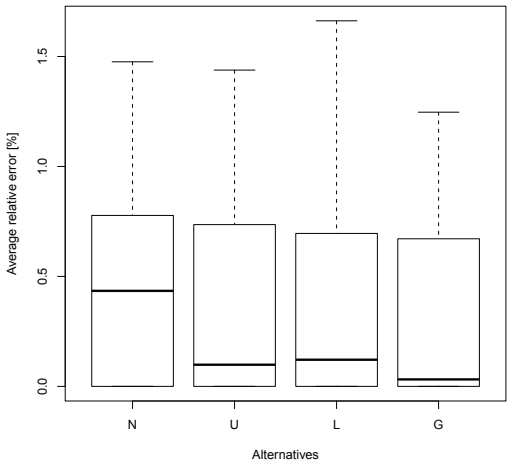

Figure 4: Boxplot with the model accuracies of Problem f_3

constant value. The local mutation increases or decreases the current value (each with a probability of $p = 0.5$) using a fixed delta. The Gaussian mutation uses a Gaussian distribution as probability density function with the current constant value as mean value. For all alternatives, we used an interval for valid constants of $[-100, 100]$ and a probability for the constant mutations of $p_{cm} = 0.75$.

The number of alternatives (4) and problems (5) leads to a total number of 20 experiments. To get stabilized results, we repeated each experiment for 100 times (see Section 4.4) and thus issued 2000 independent evolutions for this experiment.

4.5.2 Analyzing the Results

In this section, we present our analysis of the results for this set of experiments and finally answer Question *Q1*. We used two metrics for the analysis: The averaged relative error (see Section 4.2) on the training data to express the quality of each solution and the generation at which a perfect fit was found as an indicator for the convergence speed.

For problems f_0, f_1 and f_2 nearly every run of all four alternatives generated the exact model leading to an averaged relative error of 0. The experiment runs for f_3 and f_4 lead to more variation in the model accuracies. Figure 4 shows a boxplot with the averaged relative errors for problem f_3 for all 90 runs (due to the outlier removal mentioned in Section 4.4). We omitted all values above the upper whisker in all boxplots to improve the illustration. The uniform (U) and local (L) mutation perform quiet similar, whereas no mutation (N) seems to perform slightly worse by having a median of 0.43% compared to 0.10% for uniform and 0.12% for local mutation. The Gaussian (G) mutation tends to perform best by having the lowest variance and the lowest median (0.03%).

To identify the best alternative, we merged the models of the 90 runs for each of the five problems. This lead to 450 models per alternative. We applied the Kruskal-Wallis Rank Sum test to identify if the differences among all four distributions are significant. The null hypothesis states that no differences among all tested groups exist. The test returned

	Accuracy	**Generation**
Test	p-value	p-value
All (Kruskal-Wallis)	0.0007*	0.0012*
L-G (Wilcoxon)	0.6862	0.6111
N-G (Wilcoxon)	0.0022*	0.0022*
U-G (Wilcoxon)	0.0015*	0.0022*
N-L (Wilcoxon)	0.0086*	0.0109*
U-L (Wilcoxon)	0.0054*	0.0119*
U-N (Wilcoxon)	0.9841	0.9936

Table 3: Results of applied significance tests

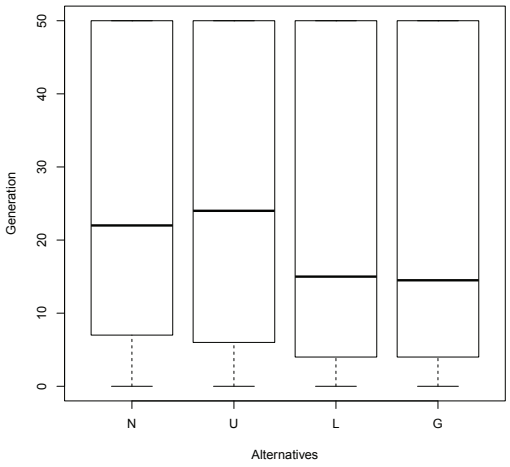

Figure 5: Boxplot with merged generations of Problems 0 to 4

a p-value of 0.0007 (see Table 3) implying significant differences between the groups. To identify between which groups the differences exist, we pair wisely applied the Wilcoxon test. The results are denoted in the column "Accuracy" of Table 3. The asterisk besides the value implies significant differences based on a significance level of $\alpha = 0.05$. This means, that local (L) and Gaussian (G) mutation perform significantly better than uniform (U) and no (N) mutation. Within both groups (L/G and U/N), no significant differences are testified.

The absolute differences in accuracy were quite small among all four alternatives. The medians of the merged accuracies for all alternatives are 0% and the largest difference of the third quartils is between N (0.183%) and G (0.000%). We further investigated the convergence speed, i.e., the generation at which the exact model was found. Figure 5 shows the boxplot containing the generations at which the exact models were found for each of the alternatives.

Local (L) and Gaussian (G) mutation perform similar and their medians are 15 (L) and 14.5 (G). The medians of uniform (U) and no (N) mutation are 24 (U) and 22 (N). The p-value of 0.0012 received by the Kruskal-Wallis test indicates that the null hypothesis can be rejected, meaning that significant differences among the alternatives exist. Thus, we performed a subsequent Wilcoxon test pair wisely for each alternative. The results of this test are depicted in the "Generation" column of Table 3. Again, an asterisk besides

the p-value indicates an significant difference between the distributions on a significance level of $\alpha = 0.05$. The test reveals that Gaussian (G) and local (L) mutation has a significantly positive effect on the convergence speed compared to no mutation (N) and uniform mutation (U).

This analysis allowed us to answer the initial question, which mutation type performs best for integer constants. The statistical tests left us the choice between local (L) and Gaussian (G) mutation and we decided to use the Gaussian mutation (G) due to the better tendencies in accuracy (see Figure 4) and convergence speed (see Figure 5). We applied the same procedure to the remaining questions of all goals. In the next section, we briefly present the results for all goals.

4.6 Results of the Optimization and Final GP Settings

We conclude this section about the adaption of the GP algorithm with a summary of all results for each of the goals presented in Section 4.3.

Goal 1 addresses the constant mutation and our experiments reveal that the use of integer and float constants having the intervals [-100,100] and [0,1] is sufficient. The Gaussian mutation is applied to both constant types with a probability of $p_{cm} = 0.75$. A final experiment shows that this configuration is sufficient to generate constants with the desired accuracy. Goal 2 investigates the influence of different fitness functions and the experiments indicate that the use of relative errors leads to a faster convergence than using absolute residuals. Goal 3 investigates the ability of the GP algorithm to solve multi-dimensional problems. The experiment results show that simple problems such as linear combinations can be approximated accurately even if the dimensionality is high (up to 10 dimensions). Goal 4 addresses preprocessing of the training data. Corresponding experiments reveal that averaging measurement data with same input configurations (results of repeated measurements) leads to better results and to faster execution times. An appropriate size of the training set highly depends on the complexity of the underlying problem. It is crucial that the training set contains enough data to represent the problem otherwise the algorithm cannot build reasonable prediction models. Finding an appropriate function set is addressed in Goal 5. Based on our experiments, we decided to use two different function sets during independent runs of the algorithm: The "Basic" function set contains the arithmetic functions $(+, -, *, /)$. The "Extended" function set additionally contains the *exp*, *power*, *log* and *hinge* function. Driven by the work of Gustafson et al. [9], we investigated in Goal 6 the proposed algorithmic improvement of the parent selection process. However, the experiment reveals no significant improvement compared to tournament selection with $k = 7$. Goal 7 addresses the choice of an appropriate population size and a reasonable number of generations. Based on the experiments, we decided to evolve 2048 individuals for 102 generations per evolution.

Goal 8 investigates the influence of domain knowledge, but corresponding experiments indicate that the use of domain knowledge neither significantly influences the accuracy of the results nor convergence speed. Goal 9 copes with the problem of overfitting. The experiments persuades us to use a combination of two techniques: The first is a cross-validation which selects the best-of-run individual among all best-of-generation individuals based on a separate test set. The second technique uses only a subset of the available training data. The subset is changed randomly for each generation (see [18]). Experiments show that using these two techniques reduces the overfitting problem sufficiently for our purposes.

5. EVALUATION

In this section, we compare the accuracy of the models inferred by our GP algorithm with models derived by other analysis techniques. We start with applying the algorithms to a simple and synthetic function which allows the visualization of the results. The second and main part of the evaluation describes the application of the algorithm to real measurements of a MySQL database.

We automated the evaluation using the SoPeCo (see Section 3). As competitive analysis techniques, we used *inverse distance weighting* (IDW) and *multivariate adaptive regression splines* (MARS). As explained in Section 2, we used MARS instead of neural networks, since ANNs and other machine learning techniques would require similar adjustments as the GP algorithm which was beyond the scope of this work.

All analysis techniques are implemented as analysis adapter for the SoPeCo. In the following, we present all three techniques and explain the configuration parameters of the adapters besides the configuration of dependent and independent parameters.

IDW is a multivariate interpolation technique. Interpolation techniques estimate the value of unknown points as a combination of existing points. IDW uses a weight function for this combination, where the weight for each existing point is inverse to the distance between that point and the estimated point. IDW uses all surrounding points for the interpolation, meaning that all known points influence the estimation, but closer points with higher effects (see [22]). The IDW adapter has two additional configuration parameters: A strategy for calculating the distance between two points and an exponent for the distance metric to express their influence. We used the Euclidean norm as distance and a weight exponent of 2.

MARS is a non-parametric regression technique that creates models by combining piecewise linear basis functions. The algorithm creates the prediction model in two steps: First, it iteratively builds a model by combining the existing model with new basis functions. The second step uses backward deletion to prune the model. This reduces the model complexity and avoids an overfitting of the model (see [11]). The MARS analysis adapter does not implement the algorithm itself, but delegates the call to a corresponding module of the statistic tool R [1].

The GP analysis adapter implements the GP algorithm as presented in this work. The implementation is based on the ECJ framework [16]. The user must configure the function sets ("Basic" and/or "Extended') as described in Section 4.6. We used both function sets for this evaluation scenario. Since the GP algorithm is non-deterministic, the adapter implements an internal loop which repeats the evolution for a configurable number of times for each function set. After

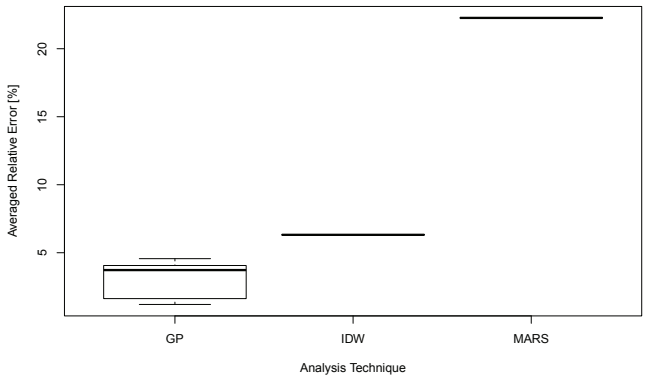

Figure 6: Model accuracies of the one-dimensional synthetic function

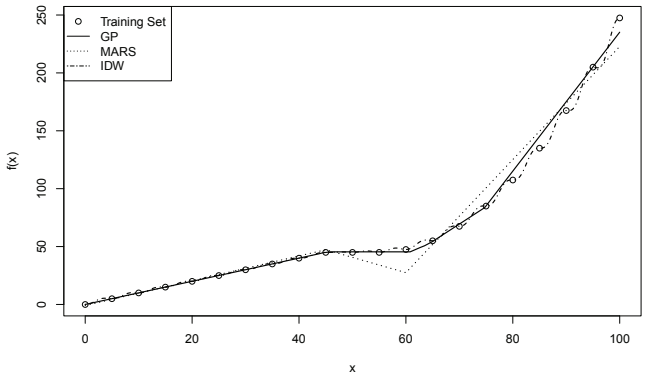

Figure 7: One-dimensional synthetic function illustrating the results of IDW, MARS and GP

all iterations, the adapter returns the best model based on the accuracy on internal test sets. We configured 5 iterations and thus one call of the adapter leads to 10 evolutions: 5 with the "Basic" function set and 5 with the "Extended" function set. Despite this mechanism, the results of the adapter are still non-deterministic. Thus, we called the GP analysis adapter 10 times for this evaluation.

In the following, we present the results of our twofold evaluation. The first part is based on a synthetic function and the second part uses realistic measurement results of a MySQL database system.

5.1 Synthetic Function

To illustrate the results of GP, IDW and MARS, we applied the different techniques to a synthetic function with only one independent parameter (one-dimensional). The target function is piece wisely defined as:

$$f(x) = \begin{cases} x, & \text{if } x \leq 45, \\ 45, & \text{if } 45 < x \leq 55, \\ 0.1 * (x - 55)^2 + 45, & \text{if } x > 55. \end{cases} \quad (4)$$

The function is linear in the first part, constant in the middle part and quadratic in the last part. The training set for all three techniques contains 21 equally-distributed points with a distance of $\Delta x = 5$.

We used the averaged relative error (see Equation 3) as a metric to compare the three techniques. The averaged relative error is based on a validation set containing every value for x in the interval $[0, 100]$ with a step width of $\Delta x = 0.1$ and the corresponding $f(x)$. This leads to a total validation set size of 1001.

Figure 6 depicts the averaged relative errors of the results derived by the three different analysis techniques. We used a boxplot to illustrate the results from all 10 runs of the GP adapter. The results ranged from 1.20% up to 4.66%, with a median value of 3.72%. IDW lead to a model with 6.32% averaged relative error and MARS to a model with 22.66%. The worst model returned by the GP adapter is still 1.66% better than the result of IDW and 18% better than the MARS model.

Figure 7 shows the resulting functions of the IDW and MARS analysis and one result of the 10 GP runs. The circles illustrate the training set containing 21 points. The dotted line represents the MARS result, which is composed of three linear functions. The first linear function perfectly fits the target function, whereas the second part approximates the constant part and the third the polynomial part. The dashed line shows the IDW result and the principle of IDW. Since IDW is an interpolation technique, every point of the training is intersected. The solid line depicts the third-best result of the GP analysis with an averaged relative error of 1.62%. This result is composed of 5 linear functions: One represents the linear part and one the constant part. The other three functions approximates the polynomial part of the target function.

5.2 Case Study: MySQL Database

In this part of the evaluation, we used measurements of a MySQL 5.5 [17] database to derive a software performance curve which describes the response times for different query types. We start the section with explaining the experiment setup for retrieving the measurements. Next, we explain two sampling methods (equidistant and random) which were used to retrieve points for the training set. Finally, we present and discuss the results of all three analysis techniques.

The MySQL 5.5 database was deployed on a machine having an Intel Core 2 Duo Processor with 2 × 2GHz, 2GB RAM and a SATA hard disk with 5400RPM and 8MB buffer (Hitachi-Travelstar). The database was accessed via LAN from a remote machine. The Software Performance Cockpit was used to automate the measurements. To receive stable results, each query was repeated for 50 times and the arithmetic average was used as training data.

The target metric is the response time and the input parameters were restricted to six parameters expressing the query type, the queue length and the queue structure. The first parameter (`AccessType`) differentiates between reading access (`Read`) and writing access (`Update`). The table size is fixed to 100,000 rows in this evaluation scenario. The second parameter (`NumberOfRequestedLines`) defines how many rows are accessed and is set to 1 (12,000) to approximate small (large) requests. The remaining four parame-

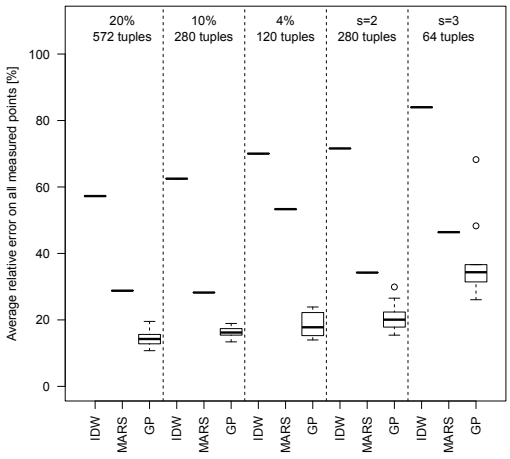

Figure 8: Model accuracies when using different analysis methods and training sets

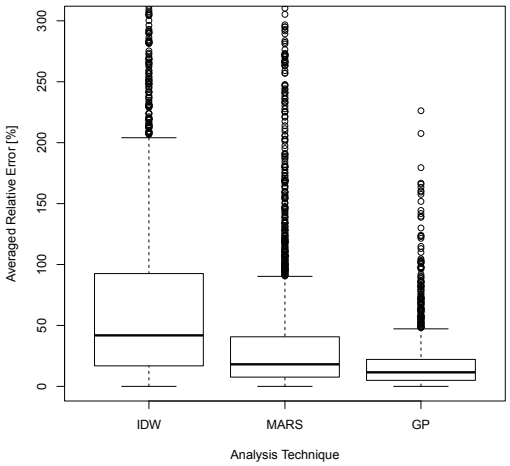

Figure 9: Relative errors of predictions for all points in the validation set using the 4% randomly-distributed training set

ters (`NumRead_1`, `NumRead_12000`, `NumUpdate_1` and `NumUpdate_12000`) describe the queue length and structure of the database. The value for each parameter indicates how many threads of the corresponding query type (Read/Update 1/12,000 lines) are currently in the queue. The domain for these parameters was defined by the interval [0,10]. The queue length can be derived by totaling all four parameters and was restricted to 10 in this scenario. The semantics of the parameters also imply that the thread counter for the current target query type must be larger than zero. For example, if we observe a small read job, $NumRead\_1$ must be larger than zero.

These parameter and domain definitions and the constraints, lead to 2,860 valid parameter combinations for this scenario. We measured all 2,860 points of the parameter space and used a subset as training set. The complete set was used as validation set to calculate the averaged relative error for all derived models. We used two different sampling types: random and equidistant. The former one randomly picks points among all valid points and adds them to the training set. These training sets were sampled once and then used as basis for all analysis techniques. The latter technique systematically creates all valid parameter combinations by varying all parameters using a minimum, maximum and a step width. This technique results in an equidistant distribution of points in the parameter space. We created three random training sets containing 20% (572 tuples), 10% (280 tuples) and 4% (120 tuples) of all 2,860 measurements and two equidistant training sets using a step width of $s = 2$ and $s = 3$ for all thread parameters, leading to 280 and 64 tuples.

Figure 8 depicts the model accuracies for different analysis methods and training sets. IDW returned the worst results with model accuracies between 57% and 84%. As expectd, the accuracy increases, when increasing the size of the training set. Surprisingly, it appears that randomly-sampled training sets lead to better model accuracies than equidistantly-sampled training sets (compare columns "10%" and "s=2" of Figure 8). The accuracies of the MARS results are between IDW and GP and vary between 28% and 53%. The MARS results are also proportional to the training set size and random sampling performs better than equidistant sampling ("10%" and "s=2"). It is noticeable that the "s=3"-training set with 64 tuples leads to a better model than the 4%-training set with 120 tuples. The boxplots for GP represent the results of 10 independent runs of this analysis method (not to be mixed up with the internal repeats to receive one result). The models created with GP outperform the IDW and MARS models in most cases (except 2 models with the "s=3"-training set).

The GP analysis appears to be more stable towards the training set size. The medians of the model accuracies for all 10 runs vary between 14% and 21% for all training sets (except the "s=3"-training set with a median of 34%). The errors among all 10 calls using the same training set vary up to 10% for the randomly-sampled training sets. Due to the almost constant model accuracies, independent of the training set size (compare columns "20%", "10%" and "4%" of Figure 8), we assume that the remaining model errors are caused by measurement errors and are representing the noise and error in the training data. Higher accuracies are not desirable in this scenario as they would result in over fitting.

We compare the models created using the 4% randomly-distributed training set in more detail: The MARS model has an averaged relative error of 52.3% and IDW of 70.0%. The averaged relative error for all 10 GP models are between 14.0% and 23.0%. Besides the averaged relative error of a software performance curve, other metrics such as the variance of the relative errors among all predictions are meaningful. Instead of using the variance, we used a boxplot to visualize the quality for single predictions. Figure 9 depicts the relative errors for predictions of all points in the validation set. The model marked with "GP" is the median model (based on the averaged relative error for all predictions) among all 10 models created for this evaluation. The dispersion of the GP model is the lowest and 75%

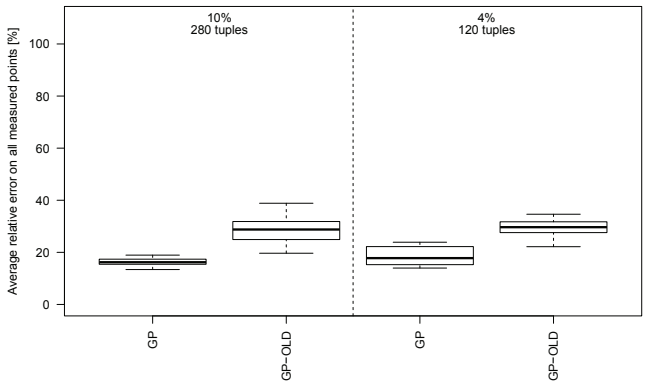

Figure 10: Differences of model accuracies between initial and optimized GP algorithm

of all predictions have a relative error smaller than 22.1%. The worst prediction has a relative error of 226.2%. For MARS (IDW), 75% of all predictions have relative errors below 40.7% (92.5%) and the error of the worst prediction is 1723% (706.7%).

5.3 Impact of the GP Optimization

In this section, we use the experiment setting described above to validate the influence of all parameter optimizations for the GP algorithm and all applied techniques to prevent overfitting. For this purpose, we built models using the GP algorithm without any techniques to prevent overfitting and the initial parameter settings (51 generations with a population size of 1024; no constant mutation; only arithmetic functions). Due to the non-determinism, we repeated the execution 10 times with the unmodified GP algorithm ("GP-Old") using the 10% and 4% training set.

Figure 10 depicts the results. The optimized GP algorithm, including the techniques to prevent overfitting, leads to more accurate models. The median runs have a difference of 12.6% (for the 10% training set) and 11.9% (for the 4% training set). This result shows, that the effort we put in optimizing the GP algorithm was justified. All optimizations, including the techniques to prevent overfitting, caused improvements of the derived software performance curves of about 12%.

6. DISCUSSION

The evaluation shows, that the models derived by GP are more accurate than the models created by MARS and IDW. The results also reveal that the algorithm performs well with only a few number of measurement samples. We integrated techniques to reduce the undesired effect of overfitting. The symbolic regression approach does not only find coefficients but simultaneously finds a structure for the model. Hence, the approach does not take any assumptions about the dependencies between the input and output parameters such as linearity in linear regression. The algorithm is capable of finding complex dependencies and might perform automatic parameter selection by using only variables which highly contribute to the model quality in terms of the fitness value. The experiments performed in the third goal (G3) reveal that the algorithm also performs well for high-dimensional problems (tested up to ten dimensions).

The analysis method is, by nature, non-deterministic. Thus, it is necessary to repeat the analysis several times and finally select the best performance curve (based on a test set). To minimize this limitation, we integrated this loop including the selection step in our analysis adapter for the Software Performance Cockpit. The number of iterations can be configured by the user depending on the available time for the analysis step. However, the repeated execution of our algorithm leads to longer execution times compared to MARS and IDW. Another limitation of our approach is that the resulting formulas, representing the performance curve, are very complex and thus are not immediately comprehensible for the Performance Analyst. To overcome this lack of transparency, the Performance Analyst might simplify the resulting formula using tools such as MATLAB [25] in order to further analyze and interpret the formula manually.

7. CONCLUSION

In this paper, we examined the use of genetic programming as analysis method for deriving software performance curves. The concepts of evolutionary algorithms are very intuitive, but the adoption to a specific problem is not. We employed the GQM approach to systematically derive experiments for the optimization of the GP parameters. These optimizations include aspects such as constant mutations, population size and number of generations, function sets and techniques to prevent overfitting. In a final evaluation, we show that the optimized GP algorithm outperforms MARS and IDW in terms of model accuracy. A comparison of the optimized GP algorithm with the initial GP algorithm reveals an increase in the model accuracy of around 12%.

Our GP approach can be used by Performance Analysts to derive more accurate software performance curves. The complexity of the approach is hidden in an analysis adapter for the SoPeCo and thus the approach is easy-to-use just by configuration. We do not claim the GP approach fits for all analysis scenarios. Moreover, we see the approach as an additional analysis method which has its advantages in its flexibility. E.g., if a linear dependency between the parameters is known, MARS or a simple linear regression might be more appropriate. But especially if no parameter dependencies are known the GP approach might be an appropriate analysis method.

In our future work, we plan to apply this GP analysis also as a method to estimate parameter-dependent resource demands. Concerning the evaluation, we are currently working on a larger set of case studies comprising of standard benchmarks (e.g., SPECjvm2008) and SAP internal applications where we additionally compare multiple inference techniques such as CART and Kriging. Furthermore, more research is necessary to automatically identify the most influencing factors on the performance in order to reduce the number of measurements and to handle the curse of dimensionality. Concerning a further optimization of the GP algorithm, it is necessary to investigate the mutual dependencies between algorithm parameters (e.g., between mutation rate and number of generations). Additional improvements might be achieved by experimenting with other techniques to prevent overfitting and evaluate their efficiency using synthetic and realistic data.

8. REFERENCES

[1] The R Project for Statistical Computing. `http://www.r-project.org`, last visited: 05.09.2010, 2010.

[2] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *Software Engineering, IEEE Transactions on*, 30(5):295 – 310, may 2004.

[3] V. R. Basili, G. Caldiera, and H. D. Rombach. The Goal Question Metric Approach. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering - 2 Volume Set*, pages 528–532. John Wiley & Sons, 1994.

[4] D. Beasley, R. Martin, and D. Bull. An overview of genetic algorithms: Part 1. Fundamentals. *University computing*, 15:58–58, 1993.

[5] M. Courtois and M. Woodside. Using regression splines for software performance analysis. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 105–114, New York, NY, USA, 2000. ACM.

[6] P. Dalgaard. *Introductory statistics with R*. Corr. print. edition, 2003.

[7] A. Eiben and J. Smith. *Introduction to evolutionary computing*. 2003.

[8] F. Ferrucci, C. Gravino, R. Oliveto, and F. Sarro. Genetic programming for effort estimation: An analysis of the impact of different fitness functions. *Search Based Software Engineering, International Symposium on*, 0:89–98, 2010.

[9] S. Gustafson, E. Burke, and N. Krasnogor. On improving genetic programming for symbolic regression. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 1, pages 912 –919 Vol.1, 2005.

[10] J. Happe, D. Westermann, K. Sachs, and L. Kapova. Statistical Inference of Software Performance Models for Parametric Performance Completions. In G. Heineman, J. Kofron, and F. Plasil, editors, *Research into Practice - Reality and Gaps (Proceedings of QoSA 2010)*, volume 6093 of *LNCS*, pages 20–35, 2010.

[11] T. J. Hastie, R. J. Tibshirani, and J. H. Friedman. *The elements of statistical learning : data mining, inference, and prediction*. Springer series in statistics. Springer-Verlag, Berlin, Germany, New York, NY, 2. ed. edition, 2009.

[12] J. R. Koza, editor. *Genetic programming*, volume [1, book]: On the programming of computers by means of natural selections. MIT Press, Cambridge, Mass., 3. print. edition, 1993.

[13] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Perform. Eval.*, 67:634–658, August 2010.

[14] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson. Estimating service resource consumption from response time measurements. In *VALUETOOLS '09: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2009. ICST.

[15] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 249–258, New York, NY, USA, 2007. ACM.

[16] S. Luke. ECJ. `http://cs.gmu.edu/~eclab/projects/ecj/`, last visited: 16.01.2011.

[17] Oracle. Mysql 5.5. last visited 04.01.2011.

[18] L. Panait and S. Luke. Methods for evolving robust programs. In *Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartII*, GECCO'03, pages 1740–1751, Berlin, Heidelberg, 2003. Springer-Verlag.

[19] D. C. Psichogios, R. D. De Veaux, and L. H. Ungar. Non-parametric system identification: A comparison of mars and neural networks. In *American Control Conference, 1992*, pages 1436 –1441, june 1992.

[20] C. Ryan and M. Keijzer. An analysis of diversity of constants of genetic programming. In *Proceedings of the 6th European conference on Genetic programming*, EuroGP'03, pages 404–413, Berlin, Heidelberg, 2003. Springer-Verlag.

[21] A. B. Sharma, R. Bhagwan, M. Choudhury, L. Golubchik, R. Govindan, and G. M. Voelker. Automatic request categorization in internet services. *SIGMETRICS Perform. Eval. Rev.*, 36(2):16–25, 2008.

[22] D. Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*, ACM '68, pages 517–524, New York, NY, USA, 1968. ACM.

[23] C. U. Smith. *Performance Engineering of Software Systems*. 1990.

[24] D. Thakkar, A. E. Hassan, G. Hamann, and P. Flora. A framework for measurement based performance modeling. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 55–66, New York, NY, USA, 2008. ACM.

[25] The MathWorks Inc. MATLAB, version 7.11, 2010.

[26] D. Westermann and J. Happe. Software Performance Cockpit. `http://www.software-performance-cockpit.org/`, September 2011.

[27] D. Westermann, J. Happe, M. Hauck, and C. Heupel. The performance cockpit approach: A framework for systematic performance evaluations. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*, page To Appear. IEEE Computer Society, 2010.

[28] D. Westermann, R. Krebs, and J. Happe. Efficient experiment selection in automated software performance evaluations. In *Proceedings of 8th European Performance Engineering Workshop (EPEW)*, page To appear., 2011.

[29] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society.

[30] T. Zheng, C. Woodside, and M. Litoiu. Performance model estimation and tracking using optimal filters. *Software Engineering, IEEE Transactions on*, 34(3):391 –406, may-june 2008.