Predictive Performance Modeling of Virtualized Storage Systems using Optimized Statistical Regression Techniques

Qais Noorshams, Dominik Bruhn, Samuel Kounev, Ralf Reussner
Chair for Software Design and Quality
Karlsruhe Institute of Technology
Karlsruhe, Germany
{noorshams, kounev, reussner}@kit.edu,
dominik.bruhn@student.kit.edu

ABSTRACT

Modern virtualized environments are key for reducing the operating costs of data centers. By enabling the sharing of physical resources, virtualization promises increased resource efficiency with decreased administration costs. With the increasing popularity of I/O-intensive applications, however, the virtualized storage used in such environments can quickly become a bottleneck and lead to performance and scalability issues. Performance modeling and evaluation techniques applied prior to system deployment help to avoid such issues. In current practice, however, virtualized storage and its performance-influencing factors are often neglected or treated as a black-box. In this paper, we present a measurement-based performance prediction approach for virtualized storage systems based on optimized statistical regression techniques. We first propose a general heuristic search algorithm to optimize the parameters of regression techniques. Then, we apply our optimization approach and create performance models using four regression techniques. Finally, we present an in-depth evaluation of our approach in a real-world representative environment based on IBM System z and IBM DS8700 server hardware. Using our optimized techniques, we effectively create performance models with less than 7% prediction error in the most typical scenario. Furthermore, our optimization approach reduces the prediction error by up to 74%.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques, Performance attributes

Keywords

I/O, Storage, Performance, Prediction, Virtualization

ICPE'13, March 21–24, 2013, Prague, Czech Republic.

1. INTRODUCTION

The high growth rate of modern IT systems and services demands a powerful yet flexible and cost-efficient data center landscape. Server virtualization is a key technology used in modern data centers to address this challenge. By enabling the shared resource usage by multiple virtual systems, virtualization promises increased resource efficiency and flexibility with decreased administration costs. Furthermore, modern cloud environments enable new pay-per-use cost models coupled with flexible on-demand resource provisioning.

Modern cloud applications increasingly have I/O-intensive workload profiles (cf. [2]), e.g., mail or file server applications are often deployed in virtualized environments. With the increasing popularity of such applications, however, the virtualized storage of shared environments can quickly become a bottleneck and lead to unforeseen performance and scalability issues. Performance modeling and evaluation techniques applied prior to system deployment help to avoid such issues and ensure that systems meet their quality-of-service requirements.

In current practice, however, virtualized storage and its performance-influencing factors are often neglected or treated as a black-box due to their complexity. Several modeling approaches considering I/O-intensive applications in virtualized environments have been proposed, e.g., [1, 17], however, without explicitly considering storage configuration aspects and their influences on the overall system performance. Moreover, the increasing complexity of modern virtualized storage systems poses challenges for classical performance modeling approaches.

In this paper, we present a measurement-based performance prediction approach for virtualized storage systems. We derive optimized performance models based on statistical regression techniques capturing the complex behaviour of the virtualized storage system. Furthermore, the models explicitly consider the influences of workload profiles and storage configuration parameters. More specifically, we first propose a general heuristic search algorithm to optimize the parameters of regression techniques. This algorithm is not limited to a certain domain and can be used as a general regression optimization method. Then, we apply our optimization approach and create performance models based on systematic measurements using four regression techniques. Finally, we evaluate the models in different scenarios to assess their prediction accuracy and the improvement achieved by

our optimization approach. The scenarios comprise interpolation and extrapolation scenarios as well as scenarios when the workload is distributed on multiple virtual machines. We evaluate our approach in a real-world environment based on IBM System z and IBM DS8700 server hardware. Using our optimized techniques, we effectively create performance models with less than 7% prediction error in the most typical interpolation scenario. Furthermore, our optimization approach reduces the prediction error by up to 74% with statistical significance.

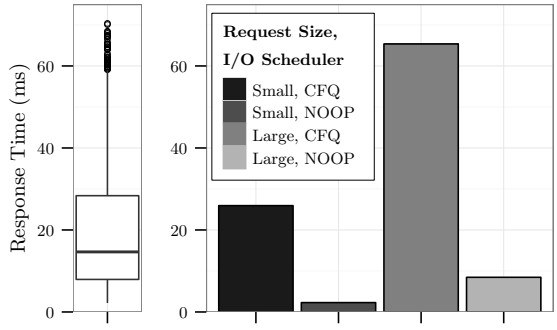
In summary, the contribution of this paper is multifold: i) We present a measurement-based performance modeling approach for virtualized storage systems based on four statistical regression techniques. ii) We propose a general heuristic optimization method to maximize the predictive power of statistical regression techniques. To the best of our knowledge, this is the first automated regression optimization method applied for performance modeling of virtualized storage systems. iii) We validate our approach in a real-world environment based on the state-of-the-art server technology of the IBM System z and IBM DS8700. iv) We comprehensively evaluate our approach in multiple aspects including interpolation and extrapolation scenarios, scenarios with multiple virtual machines, and the improvement achieved by our optimization approach.

The remainder of this paper is organized as follows: Section 2 further motivates our approach. Section 3 discusses the regression techniques we considered for performance modeling. In Section 4, we introduce our system environment and methodology. Section 5 presents our heuristic optimization algorithm. In Section 6, we extensively evaluate our performance models and optimization approach. Finally, Section 7 presents related work, while Section 8 summarizes and concludes with the lessons learned.

2. MOTIVATION

The use of virtualization technology has significant implications on the landscape of modern data centers. By consolidating multiple systems to increase resource efficiency, virtualization presents today's systems with increasing challenges to meet the non-functional requirements. Especially for virtualized storage systems, this trend is evident. At the cost of highly increased complexity, storage systems have evolved significantly from simple disks to sophisticated systems with multiple caching and virtualization layers. Furthermore, modern virtualized storage systems feature intricate scheduling and optimization strategies.

Without any doubt, there are a variety of performance influences of virtualized storage systems (cf. [23]), e.g., the architecture of the virtualization platform and the implications of workload consolidation. Moreover, the many logical layers between the application and the physical storage lead to complex effects and interdependencies of influencing factors, which are more and more difficult to quantify. The effect of a few simple influencing factors alone leads to a wide range of performance characteristics. Illustrated in Figure 1a for instance, the response time of an application spreads by more than the factor of 35 when varying five factors, e.g., the request size or the I/O scheduler. Moreover, Figure 1b exemplifies a shift in the influencing factors. The CFQ scheduler, which is the standard I/O scheduler since many years, impairs performance drastically for certain configurations. Consequently, the performance traits of an application deployed in such an environment are highly unclear and might be also subject to change.



(a) Distribution (b) Examples

Figure 1: Response Time Depending on Influencing Factors

This development is the motivation for our approach. The increasing complexity of modern virtualized storage systems poses challenges for classical performance modeling approaches to manually create, calibrate, and validate explicit performance models within a limited time period. The effects and interdependencies of the influencing factors are more and more complex, which needs to be accounted for and quantified. Furthermore, the effects are subject to change and need to be captured flexibly as the systems keep evolving. Therefore, we target at an automated measurement-based approach to virtualized storage performance modeling based on regression techniques. Our goal is to create practical performance models while lifting the burden for performance engineers to manually develop them. Building a regression-based performance model, however, is also not straightforward due to the many techniques and their parameters. To briefly introduce the techniques we considered, the following section analyzes multiple regression techniques and their parameters.

3. REGRESSION TECHNIQUES

Regression techniques are used to model the effect of *independent variables* on *dependent variables*, e.g., the effect of I/O request sizes on the request response time. In this section, we introduce and analyze the regression techniques we applied for modeling the performance of virtualized storage systems and discuss their parameters. Furthermore, we highlight the relationships between the considered techniques.

3.1 Linear Regression Models

One of the most popular techniques is based on *linear regression models (LRM)* [12] which assume a linear relationship between the dependent variable and the independent variables. Thus, the dependent variable is represented as a linear combination of the independent variables with an added constant intercept. More formally, for a vector of independent variables $\vec{x} := (x_1, \ldots, x_n)$ a model f with coefficients $\beta_0, \ldots, \beta_n$ is created of the form

$$f(\vec{x}) = \beta_0 + \sum_{i=1}^{n} \beta_i x_i.$$

The model f, however, does not necessarily have to be linear in the independent variables x_i. The independent variables can be derived, e.g., $x_2 = x_1^2$ or $x_3 = x_1 x_2$. For the sake of clarity, we explicitly exclude such definitions and specifically consider two forms: The *linear regression models* are explicitly linear in the independent variables as well. The *linear regression models with interaction* can include terms

expressing the interaction between variables, e.g., x_1x_2. This model is of maximum degree n in the independent variables x_i, however, the effects of the independent variables remain linear in the model. When creating a model based on a set of training data, the coefficients of the model are determined to minimize the error between the model and the training data.

Model Derivation. The most popular approach to create a model is the *method of least squares*. For a set of training data $\{(\vec{x}_1, y_1), \ldots, (\vec{x}_N, y_N)\}$ where $\forall i$: $\vec{x}_i := (x_{i1}, \ldots, x_{in})$, this method finds the coefficients $\vec{\beta} := (\beta_0, \ldots, \beta_n)$ such that the *residual sum of squares* defined as $RSS(\vec{\beta}) := \sum_{i=1}^{N} (y_i - f(\vec{x}_i))^2$ is minimized. To find the minimum, the derivative of RSS is set to zero and solved for $\vec{\beta}$. As RSS is a quadratic function, the minimum is guaranteed to always exist.

Parameters.
– None.

3.2 Multivariate Adaptive Regression Splines

Multivariate Adaptive Regressions Splines (MARS) [9] consist of piecewise linear functions (so-called *hinge functions*). Formally, these functions with so-called *knot* t are defined as $(x-t)_+ := \max\{0, x-t\}$ and $(t-x)_+ := \max\{0, t-x\}$. Let $\vec{x} = (x_1, \ldots, x_n)$ be the vector of independent variables as above, $H := \{(x_i - t)_+, (t - x_i)_+\}_{i=1,\ldots,n}$ and $t \in T$, where T is the set of observed values of each independent variable. Then, MARS constructs a model f of the form

$$f(\vec{x}) = \beta_0 + \sum_{i=1}^{n} \beta_i \, h_i(\vec{x})$$

with $h_i \in H$ and coefficients $\beta_0, \ldots, \beta_n$. Similar to the linear regression models, we explicitly distinguish between *MARS* and *MARS with interactions*. The latter also includes hinge functions that are a product of one or more functions in H.

Model Derivation. The parameters β_i are estimated by the method of least squares as for the linear regression model. The model is constructed iteratively in a forward step. Starting from a constant function, the algorithm adds the hinge functions in H that result in the maximum reduction of the residual squared error. If interactions are permitted, the algorithm can also choose a hinge function as a factor. However, each variable can only appear once in a term, i.e., higher-order powers of a variable are excluded. This step stops if a predefined number of maximum terms is reached or if the residual error falls below a predefined threshold. Finally, the model is pruned to avoid overfitting. The terms that produce the smallest increase in residual squared error are removed iteratively until a predefined number of terms is reached. Frequently, the pruning step also considers the model complexity combined with the residual squared error. This results in a trade-off between accuracy and complexity of the model.

Parameters.
– `nnodes`: Maximum number of terms in the forward step.
– `threshold`: Maximum acceptable residual error.
– `nprune`: Maximum number of terms after pruning.

3.3 Classification and Regression Trees

Classification and Regression Trees (CART) [5] are a group of algorithms that model data in a tree-based representation. CART models are binary trees with conditions in their non-leaf nodes and constant values in their leaf nodes. To determine the value of the dependant variable corresponding to a set of values of the independent variables, the evaluation starts at the root and the condition in this node is checked. If the condition is true, the left edge is followed, otherwise the right edge. This is repeated until a leaf is reached.

Model Derivation. Similar to MARS, the algorithm comprises a forward and a pruning step. In the forward step, an initial tree with a single node is created. The leafs in the tree are split iteratively at a certain splitting variable and a splitting point. The splitting point is determined such that the residual squared error is minimized. The algorithm splits a leaf until it contains less samples than a predefined value. The algorithm stops if no leaf can be split any further. In the pruning step, the tree is reduced using *cost-complexity pruning* (cf. [12]). Let m be the m-th leaf, R_m be the region specified by the conditions to the m-th leaf, and $l(T)$ be the number of leaves in the tree T. Furthermore, for the training data $\{(\vec{x}_i, y_i)\}_i$, let the number of observations in a region $n_m := |\{\vec{x}_i \,|\, \vec{x}_i \in R_m\}|$, the mean of the observations values $\hat{c}_m := n_m^{-1} \sum_{\vec{x}_i \in R_m} y_i$, and the mean squared difference between each observed value and the mean of the observations values $q_m(T) := n_m^{-1} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2$. Then, the cost complexity criterion with parameter α is defined as

$$c_\alpha(T) := \sum_{m=1}^{l(T)} n_m q_m(T) + \alpha \, l(T)$$

that is to be minimized. The parameter α is a trade-off between complexity and goodness-of-fit to the training data. It is determined according to the residual sum of squares with cross-validation. This splits the training data into two partitions consisting of model creation and model testing data. For a given α, the tree is greedily pruned using *weakest link pruning*. Iteratively, the node with the smallest per-node increase in $\sum_{m=1}^{l(T)} n_m q_m(T)$ is collapsed.

Parameters.
– `minsplit`: Minimum number of samples to split a leaf.
– `cp`: Maximum acceptable tree complexity after pruning.

3.4 M5 Trees

M5 trees [25] are – figuratively speaking – a combination of linear regression models and CART trees. Similar to CART trees, M5 models are binary decision trees with conditions in their non-leaf nodes. The leaf nodes, however, contain linear regression models instead of the constant values as in CART trees. For the prediction of a value, starting in the root node, the conditions are evaluated until a linear regression model in a leaf is reached. Then, the model is used as the predictor.

Model Derivation. Similar to MARS and CART, the tree is first built in a forward step and pruned in the next step. To build a M5 tree, the initial model consists of a one-node tree T. The tree is split iteratively into subtrees T_i until a predefined maximum number of splits is reached. M5 splits the tree at the condition that maximizes the expected reduction in error Δe with

$$\Delta e := \sigma(T) - \textstyle\sum_i \frac{|T_i|}{|T|} \, \sigma(T_i),$$

where σ is the standard deviation. For each leaf, a linear regression model is created. Finally, the M5 model f is greedily simplified to decrease the complexity c with

$$c(T) := n^{-1} \textstyle\sum_i |y_i - f(\vec{x}_i)| \cdot \frac{n+p}{n-p},$$

where $\{(\vec{x}_i, y_i)\}_i$ is the training data, n is the number of training data and p is the number of parameters in the model. The term $\frac{n+p}{n-p}$ allows an increase in error if the complexity is decreased. For each leaf node, parameters of the linear model are removed, whereas for every non-leaf node, the node is collapsed if this reduces $c(T)$.

Parameters.
– `nsplits`: Maximum number of splits in the forward step.

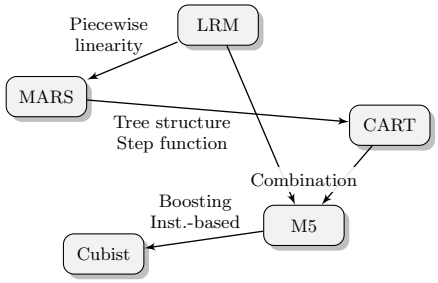


Figure 2: Relation between the Regression Techniques

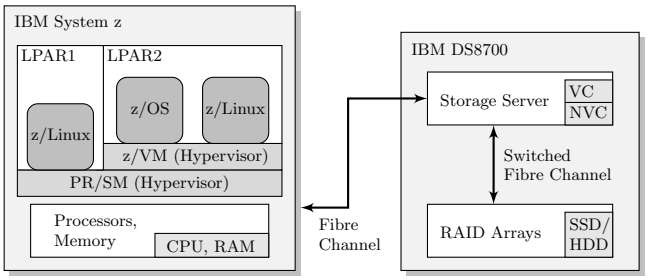


Figure 3: IBM System z and IBM DS8700

3.5 Cubist Forests

Cubist forests [26, 18] are an extension of M5 trees. Thus, Cubist forests are rule-based model trees with linear models in the leaves. Compared to M5, Cubist introduces two extensions. First, it follows a boosting-like approach, i.e., it creates a set of trees instead of a single tree. To obtain a single value, the tree predictions are aggregated using their arithmetic mean. Second, it combines model-based and instance-based learning (cf. [24]), i.e., it can adjust the prediction of unseen points by the values of their nearest training points.

Model Derivation. Initially, the maximum number of trees in Cubist is defined to construct a forest. The first tree is created using the M5 algorithm. The following trees are created to correct the predictions of the training points by the previous tree $f_t(\vec{x})$. Each value of a training point y_i is modified by $y'_i := 2\,y_i - f_t(\vec{x})$ to compensate for over- and under-estimations. Then, the tree creation is repeated. In contrast to, e.g., Random Forests [4] combining the prediction trees with the mode operator, Cubist aggregates the values predicted by each tree using arithmetic mean. Finally, the prediction of unseen points can be adjusted by the values of a possibly dynamically determined number of training points (so-called instance-based correction), cf. [24]. The prediction of a new point $\vec{x}$ is adjusted by the weighted mean of the nearest training points (so-called neighbors) with weight $w_n := 1/(m(\vec{x},\vec{n}) + 0.5)$ for every neighbor $\vec{n}$, where $m(\vec{x},\vec{n})$ is the Manhatten distance of $\vec{x}$ and $\vec{n}$. M5 is a special case of Cubist with one tree and no instance-based correction.

Parameters.

- `nsplits`: Maximum number of splits in the forward step.
- `ntrees` : Number of trees.
- `ninstances` : Size of instance-based correction.

3.6 Summary

Figure 2 shows an overview of the considered regression techniques illustrating the relationships among them. As described above, the MARS algorithm can be seen as an extension of LRM by allowing piecewise linear models. However, MARS regulates the number of linear terms. While MARS and CART seem relatively different, the forward step of MARS can be transformed into the one of CART by using a tree-based structure with step functions, cf. [12]. M5 in turn can be seen as a combination of LRM and CART. However, M5 differs from CART in the complexity criterion and the pruning procedure. Finally, Cubist extends M5 by introducing a boosting like scheme creating several trees that are aggregated by their mean. Furthermore, Cubist introduces an instance-based correction to include the training data in the prediction of unseen data.

4. METHODOLOGY

In our approach, we apply statistical regression techniques to create performance models based on systematic measurements. In this section, we present our experimental environment and setup as well as our measurement methodology and performance modeling approach.

4.1 System Under Study

A typical virtualized environment in a data center consists of servers providing computational resources connected to a set of storage systems. Such storage systems typically differ significantly from traditional hard disks and native storage systems due to the complexity of modern storage virtualization platforms.

In this paper, we consider a representative virtualized environment based on the IBM mainframe *System z* and the storage system *DS8700*. They are state-of-the-art high-performance virtualized systems with redundant or hot swappable resources for high availability. The System z combined with the DS8700 represent a typical virtualized environment that can be used as a building block of cloud computing infrastructures. It supports on-demand elasticity of pooled resources with a pay-per-use accounting system (cf. [22]). The System z provides processors and memory, whereas the DS8700 provides storage space. The structure of this environment is illustrated in Figure 3.

The *Processor Resource/System Manager (PR/SM)* is a hypervisor managing logical partitions (*LPARs*) of the machine and enabling CPU and storage virtualization. For memory virtualization and administration purposes, IBM introduces another hypervisor, *z/VM*. The System z supports the classical mainframe operating system *z/OS* and special Linux ports for System z commonly denoted as *z/Linux*. The System z is connected to the DS8700 via fibre channel. Storage requests are handled by a storage server having a volatile cache (VC) and a non-volatile cache (NVC). The storage server is connected via switched fibre channel with SSD or HDD RAID arrays. As explained in [8], the storage server applies several pre-fetching and destaging algorithms for optimal performance. When possible, read-requests are served from the volatile cache, otherwise they are served from the RAID arrays and stored in the volatile cache for future requests. Write-requests are written to the volatile as well as the non-volatile cache, but they are destaged to the RAID arrays asynchronously.

In such a virtualized storage environment, a wide variety of heterogeneous performance-influencing factors exists, cf. [23]. In many cases, the factors have a significantly different effect compared to traditional native storage systems. As Figure 1b illustrates, e.g., the standard Linux I/O scheduler CFQ performs significantly worse than the NOOP scheduler

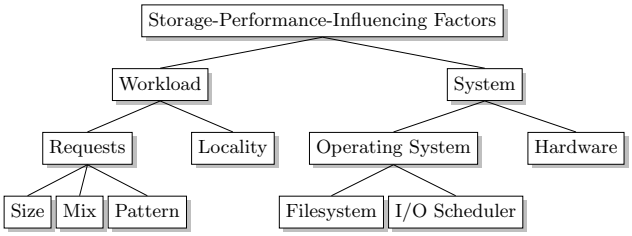


Figure 4: Performance Influences (derived from [23])

for certain configurations. Figure 4 gives an overview of the major storage-performance-influencing factors used as basis for our analysis. In general, we distinguish between workload and system factors. Workload factors comprise average request size, read/write mix and random or sequential access pattern. Furthermore, the locality of requests affects the storage cache effectiveness. System factors comprise operating system and hardware configurations. The major factors of the operating system are the file system and the I/O scheduler. Hardware factors can be analyzed on different abstraction levels, but are often system specific.

4.2 Experimental Setup

In our experimental environment, the DS8700 contains 2 GB NVC and 50 GB VC with a RAID5 array containing seven HDDs. Measurements are obtained in a z/Linux virtual machine (VM) with 2 IFLs (cores) and 4 GB of memory. Using the `O_DIRECT` POSIX flag, we isolate the effects of operating system caching in order to focus our measurements on the storage performance. However, we explicitly take into account the cache of the storage system by varying the overall size of data accessed in our workloads. As a basis for our experimental analysis, we used the open source *Flexible File System Benchmark*[1] (FFSB) due to its fine-grained configuration possibilities. FFSB runs at the application layer and measures the end-to-end response time covering all system layers from the application all the way down to the physical storage. The system and workload parameters for the measurements match the factors described in Section 4.1 and illustrated in Figure 4. The locality of the workload can be deduced from the overall size of the set of files accessed in the workload since FFSB requests are randomly distributed among the files. The considered parameter values are summarized in Table 1. The parameter space is fully explored leading to a total of 1120 measurement configurations.

4.3 Experimental Methodology

For a given configuration of a benchmark run, the measurements run in three phases: First, a set of 16 MB files is created. Second, the target number of workload threads are started and they start reading from and writing into the initial file set. After an initial warm up phase, the measurement phase starts and the benchmark begins obtaining read and write performance data. For each workload thread, the read and write operations consist of 256 sub-requests of the specified size directed to a randomly chosen file from the file set. If the access pattern is sequential, the sub-requests access subsequent blocks within the file. Each thread issues a request as soon as the previous one is completed.

For each of the 1120 configurations, we configured a one minute warm up phase and a five minute measurement phase.

[1] `http://github.com/FFSB-prime` (extending `http://ffsb.sf.net`)

Table 1: Experimental Setup Configuration

System Parameters	
File system	ext4
I/O scheduler	CFQ, NOOP
Workload Parameters	
Threads	100
File set size	1 GB, 25 GB, 50 GB, 75 GB, 100 GB
Request size	4 KB, 8 KB, 12 KB, 16 KB, 20 KB, 24 KB, 28 KB, 32 KB
Access pattern	random, sequential
Read percentage	0%, 25%, 30%, 50%, 70%, 75%, 100%

The measurement phase consists of five intervals of one minute length each. In one minute, the benchmark obtains between approximately 90 000 and 2 800 000 measurement samples depending on the configuration and approximately 575 000 measurement samples on average. Furthermore, we analyzed the measurement intervals to ensure stable and meaningful results. Illustrated in Figure 5, we evaluated the relative standard error (RSE) of the response time means for each configuration over the five intervals. For read requests, the 90th percentile of the RSE is 8.45% and the mean RSE is 3.35%. For write requests, the 90th percentile of the RSE is 5.35% and the mean RSE is 2.10%. Thus, we conclude that the measurement samples are sufficiently large and the measurements are sufficiently stable. This is important since our goal was to keep the measurement length limited.

Since a manual measurement and evaluation approach is highly tedious and error prone, we automated the process as part of a new tool we developed called *SPA (Storage Performance Analyzer)*[2]. Illustrated in Figure 6, our tool basically consists of a *benchmark harness* that coordinates and controls the execution of the FFSB benchmark and a tailored *analysis library* used to process and evaluate the collected measurements. The benchmark harness component runs on a controller machine managing the measurement process. Using SSH connections, the benchmark controller first configures the benchmark, then it executes the target workload, and it finally collects the results into an SQLite database. Furthermore, the benchmark controller guarantees a synchronized execution of experiments on multiple targets, i.e., on multiple VMs that can be deployed on the same system. The evaluation is automated using tailored analysis functions implemented using the open source statistics tool *R* [27]. The analysis library comprises the analysis, optimization and regression functions we created and applied for regression optimization and performance model creation. For more information on the technical realization, we refer the reader to [6].

4.4 Performance Modeling Methodology

We employ the regression techniques presented in Section 3. As independent variables, we used the system and the workload parameters listed in Table 1. As dependent variables, we used the system response time and throughput. For each regression technique, we first optimize the parameters to create effective models as explained in detail in the next section. We then create dedicated models for read and write requests as well as for response time and throughput predictions. Due to the structure of the independent variables, the LRM and MARS models *without* interaction perform *significantly worse* than all the other techniques. Therefore, we explicitly

[2] `http://sdqweb.ipd.kit.edu/wiki/SPA`

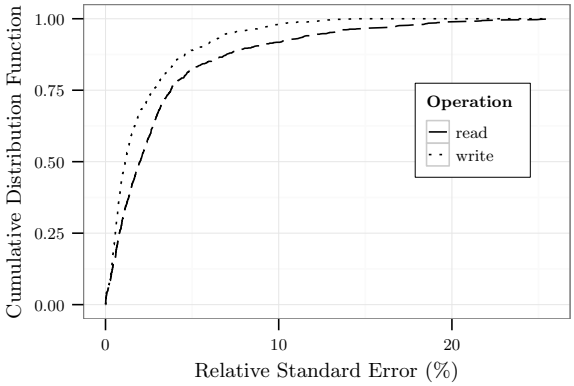


Figure 5: Distribution of the Relative Standard Error

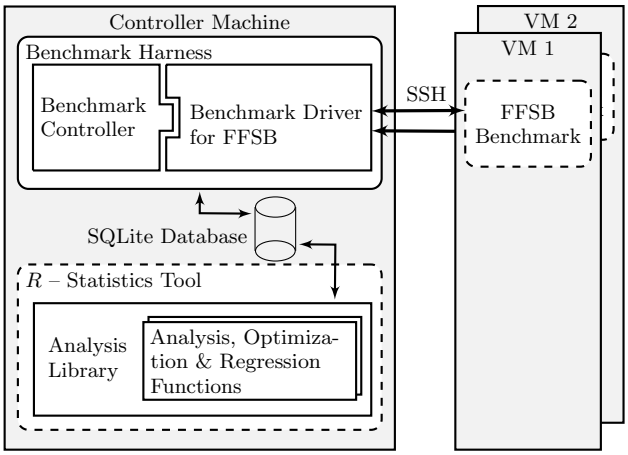


Figure 6: Overview of our Automated Methodology

exclude LRM and MARS models without interactions from consideration. Furthermore, Cubist stems from M5 both algorithmically and in terms of implementation and they are identical for Cubist's standard parameter values. Therefore, in the following, we do not explicitly distinguish between the two. Thus, having four models for each regression technique, we create a total of 16 performance models evaluated in detail in Section 6.

5. REGRESSION OPTIMIZATION

Regression techniques often have parameters, which influence the effectiveness of the technique in a certain application scenario. Usually, the parameters are left to their standard values or chosen based on an educated guess. However, both approaches are insufficient to systematically create powerful models.

To create effective performance models, we search for optimal parameters for each of our considered techniques[3], cf. Table 2. A full factorial search would take *weeks* due to the high computational costs of creating millions of regression models. Therefore, we propose a heuristic iterative search algorithm shown in Algorithm 1 and illustrated in Figure 7. In a nutshell, we split the parameter space into multiple subspaces and iteratively refine the search in the most promising regions.

[3]Note that there can be more input parameters than the ones considered here depending on the implementation of the technique. In this paper, we use the implementation in the statistics tool *R* [27].

Algorithm 1 Iterative Parameter Optimization

`Configuration:`
$k \leftarrow$ Number of splits
$n \leftarrow$ Number of explorations
$\mathcal{S}$ *// Stopping criterion*
`Definitions:`
$\vec{p} := (p_1, \ldots, p_l), L := \{1, \ldots, l\}$ *// l parameters*
$p_i \in [a_i, b_i], \forall i \in L$ *// Ranges of parameters*
$c(\vec{p}) :=$ Mean RMSE in cross-validation of parameters
`Init:`
$E \leftarrow \{(\vec{a} := (a_1, \ldots, a_l), c(\vec{a}))\}$
// Evaluate first border parameters
$M \leftarrow E$ *// M: Set of best parameters*
`Algorithm:`
while $\mathcal{S}$ does not apply **do**
  `j-th Iteration:`
  for all $(\vec{\nu}^j_{(h)}, \cdot) \in M$ **do** *// h-th pivot*
    for all $i \in L$ **do**
      $A_i \leftarrow \{p_i \mid (p_i, \cdot) \in E \wedge p_i < \nu^j_{(h)i}\}$
      if $A_i \neq \emptyset$ **then** $a^*_i \leftarrow \max A_i$
      else $a^*_i \leftarrow a_i$
      $B_i \leftarrow \{p_i \mid (p_i, \cdot) \in E \wedge p_i > \nu^j_{(h)i}\}$
      if $B_i \neq \emptyset$ **then** $b^*_i \leftarrow \min B_i$
      else $b^*_i \leftarrow b_i$
      $s^*_i \leftarrow \frac{b^*_i - a^*_i}{k+1}, \forall i \in L$ *// Step width*
      $S^*_i \leftarrow \{a^*_i, a^*_i + s^*_i, \ldots, a^*_i + k\, s^*_i, b^*_i\}$
      for all $s \in S^*_i$ **do**
        if s invalid **then** round s to next valid value
      end for
    end for
    $E^j_{(h)} \leftarrow \{(\vec{x}, c(\vec{x})) \mid \vec{x} \in S^*_1 \times \ldots \times S^*_l\}$
    // Evaluate parameters if not evaluated yet
  end for
  $E \leftarrow E \cup \bigcup_h E^j_{(h)}$ *// Save all evaluated parameters*
  $M \leftarrow$ Find n best tuples in E (w.r.t. c)
end while

Algorithm. The algorithm's configuration parameters are the number of splits (i.e., splitting points) in each iteration and the number of subspace explorations in the next iteration. The number of splits configures the sampling frequency of the search space and should be higher if multiple high but narrow optima can be assumed. The number of explorations configures how many local optima are analyzed in their adjacent area and should be higher if many local optima can be assumed. The algorithm can be configured with an arbitrary stopping criterion, e.g., when the improvement between iterations falls below a given threshold or when a specified maximum number of iterations is reached. For each of the given parameters $\vec{p} := (p_1, ..., p_l)$, a reasonable range $p_i \in [a_i, b_i]$ needs to be defined to limit the search space. The lower bound usually exists for most parameters and the upper bound can be derived based on the used training data. For a given set of parameter values, we evaluate the model using the mean *root mean square error* (RMSE) of a 10-fold cross-validation, denoted as $c(\vec{p})$. This means that the measurement data is separated into 10 groups and each group is used once as a prediction set while the rest of the data is used for model building. This evaluation approach is key to prevent overfitting of the prediction models.

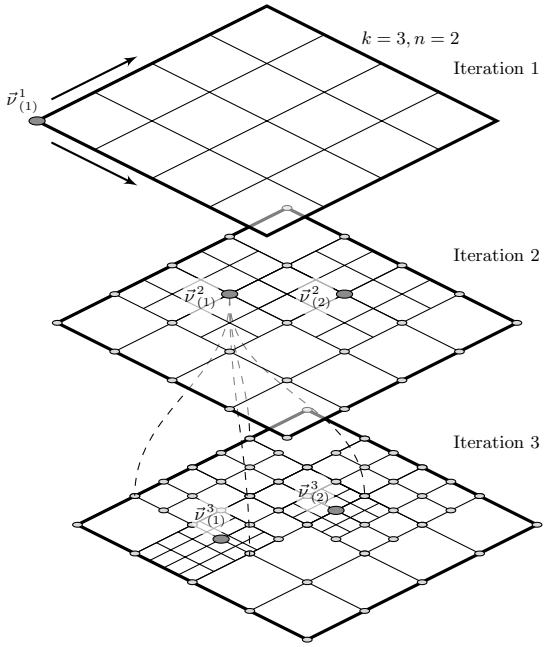


Figure 7: First Iterations of the Parameter Optimization

The algorithm begins with the evaluation of one corner point in the parameter space, cf. Figure 7. As initially this is the only evaluated point, it is the only point saved in a set M for exploration in the next iteration.

In each iteration, we explore every point $\vec{x}$ in the set M containing the points corresponding to the best parameter value assignments. The unevaluated space around $\vec{x}$ is split into $(k+1)^l$ subspaces. Initially, this is the whole parameter space. Each corner point of the subspaces is then evaluated and the best n found so far are saved in M for the next iteration. To save computation time, we maintain a set E of already evaluated parameter values. The algorithm repeats each iteration until the stopping criterion is fulfilled.

Example. In Figure 7, the first three iterations of the algorithm with $k = 3$ and $n = 2$ are illustrated. Initially, one corner point is evaluated. This pivot $\vec{\nu}^1_{(1)}$ is explored in the first iteration. Since there are no other evaluated points, the whole parameter space is cut with $k = 3$ splits for each parameter indicated by the grid lines (at the Iteration 1 level). Each of the intersection points depicted as small gray points (at the Iteration 2 level) is evaluated. After the best $n = 2$ evaluated points are found, they become the next pivots $\vec{\nu}^2_{(1)}$ and $\vec{\nu}^2_{(2)}$. Now, when $\vec{\nu}^2_{(1)}$ is explored, the space around it limited by the neighbouring evaluated points is explored, cf. dashed lines between Iteration 2 and Iteration 3. The same is done for the second pivot $\vec{\nu}^2_{(2)}$. This process is repeated in the next iterations and the best parameters are used for model building. Note that the space does not necessarily have to be split equidistantly for all parameters, cf. $\vec{\nu}^3_{(1)}$. That is why the complexity evaluation is independent from the exact values the algorithm explores.

Complexity. In our implementation of the algorithm, we used indexed data structures to realize the logic. Thus, the regression model building by means of the cross-validation evaluation represents by far the most computationally expensive part of the algorithm. Therefore, we evaluate the complexity of the cross-validation evaluation. In the worst case, in every iteration we evaluate every new set of parameter values except the initial corner one in the first iteration and the four corner ones in every other iteration. Thus, the number of evaluations in our search in the worst case is limited by

$$\mathcal{O}((k+2)^l - 1 + (\max j - 1) \cdot n \, ((k+2)^l - 4)) = \mathcal{O}(\max j \cdot n \, k^l),$$

where $\max j$ is the maximum number of iterations. By configuring $\max j$, n and k, the overhead of the algorithm can be adjusted and kept within bounds. As discussed in the next section, our evaluation showed that even moderately low values of these parameters lead to substantial improvements in the model accuracy.

Table 2: Parameters of the Regression Techniques

	Parameter	Standard	Valid/Used Range	Best $\vec{p}$ (RT_r, RT_w, TP_r, TP_w)
MARS[4]	`nnodes`	21 (der.)[5]	$[3, \infty)/[3, 800]$	255, 353, 131, 523
	`threshold`	0.001	$[0, 1)/[0, 0.999]$	$5.01e^{-6}$, $3.07e^{-7}$, $5.05e^{-5}$, $2.66e^{-5}$
	`nprune`	`nnodes`	$[2, \infty)/[2, 800]$	98, 134, 99, 141
CART	`minsplit`	20	$[2, \infty)/[2, 400]$	3, 2, 2, 2
	`cp`	0.01	$[0, 1)/[0, 0.999]$	$3.07e^{-7}$, $4.09e^{-7}$, 0, 0
Cubist	`nsplits`	100	$[2, \infty)/[2, 400]$	173, 217, 173, 217
	`ntrees`	1	$[1, 100]$	77, 41, 11, 26
	`ninstances`	0	$[0, 9]$	2, 4, 2, 1

Optimality. Our main goal is to achieve such model accuracy improvements efficiently. However, if we do not limit the number of iterations and allow to search long enough, our approach is able to find locally optimal solutions. Thus, for convex problems our approach also guarantees globally optimal solutions. This is since, let $k > 1$ and $n > 0$, $\forall i \in L$ one has if $j \to \infty$, then $s^*_i \to 0$ for continuous parameters and $s^*_i \to 1$ for discrete parameters. In other words, as we iteratively explore the current best solutions of the iteration, at least one solution will be fully explored.

Optimization Results. Based on the parameter optimization algorithm, for each regression technique we create four different models as described in Section 4: A read response time model (RT_r), a write response time model (RT_w), a read throughput model (TP_r), and a write throughput model (TP_w). We configured the number of splits k to four and the number of explorations n to five. We stop after 10 iterations. Table 2 summarizes the parameters of the regression techniques and lists their standard values as well as their valid ranges. For unbounded parameters, we chose a generous upper limit. For each model, the recommended parameter value assignments for our performance models determined by the optimization algorithm are shown. In the next section, we evaluate the accuracy of the resulting optimized regression models and demonstrate the benefits of our modeling approach.

6. EVALUATION

In this section, we analyze and evaluate our approach in three steps. First, we evaluate the performance models w.r.t. their goodness-of-fit as well as their predictive power for interpolation and extrapolation scenarios. Then, we analyze scenarios where the workload is distributed on multiple VMs. Finally, we evaluate the improvements in model accuracy achieved through our heuristic optimization. In each of the

[4]The standard MARS algorithm does *not* include interactions. However, we only consider MARS *with* interactions, cf. Section 4.4.

[5]Derived value $(\min(200, \max(20, 2 \cdot \#IndependentVariables)) + 1)$.

Table 3: Goodness-of-Fit of Performance Models

	Coefficient of determination R^2				
Model	RT_r	RT_w	TP_r	TP_w	$\frac{1}{4}\sum$
LRM	0.9720	0.9703	0.9699	0.9546	0.9667
MARS	0.9955	0.9991	0.9951	0.9973	0.9968
CART	0.9983	1	1	1	0.9996
Cubist	0.9990	0.9994	0.9995	1	0.9995

three steps, we use the optimized models generated by our SPA tool as a basis for the evaluation.

6.1 Model Accuracy

Goodness-of-Fit. For each of the optimized performance models, we evaluate how well it reflects the observed system behavior. To evaluate this goodness-of-fit, we analyze the coefficient of determination R^2 defined as

$$R^2 := 1 - \frac{SSE}{SST} = 1 - \frac{\sum_i (y_i - f(\vec{x_i}))^2}{\sum_i (y_i - \frac{1}{N}\sum_j y_j)^2} \leq 1,$$

where SSE and SST are the residual and the total sum of squares, respectively, $\{(\vec{x_i}, y_i)\}_{i=1,\ldots,N}$ represent the measurement data and f is the respective regression model. R^2 values close to 1 indicate a good fit. The results of our evaluation for the different models are shown in Table 3. Overall, most of the models fit very well to the measurements. Especially MARS performs particularly well even with a comparably low number of terms (cf. `nprune` parameter values). CART, however, has an expectedly good fit due to the low `minsplit` and `cp` parameters allowing for a highly splitted tree. Furthermore, as expected Cubist also fits very well due to the large number of trees and especially the instance-based correction. Finally, LRM exhibits a fairly well fit, however, exhibiting the lowest fit of all the models.

Predictive Power for Interpolation Scenarios. We evaluate 100 configuration scenarios with parameter values randomly chosen within the ranges shown in Table 4. For each scenario, we compare the model predictions with measurements on the real system.

Figure 9 shows the mean relative error for the various models. Overall, the models perform very well and especially MARS and Cubist exhibit excellent performance prediction accuracy with less than 7% and 8% error, respectively. The CART trees are highly splitted, yet with approximately 10% error their accuracy is acceptable. Finally, the LRM models exhibit the highest error with approximately 13%.

Figure 8 depicts the empirical cumulative distribution functions of the prediction errors. In all cases, the majority of the error is less than 10%. For MARS and Cubist, the error is less than 10% for more than 75% of the samples. Most notably, the 90th percentile of the error of both MARS and Cubist is less than 25%. Moreover, even the 99th percentile of the error is less than 50% for almost all models.

Predictive Power for Extrapolation Scenarios. We now consider configuration scenarios with parameter values outside of the ranges used to build the models. More specifically, we extrapolate the request size, the file set size, and the read percentage in two steps in each direction. We evaluate 256 scenarios by considering all combinations of the parameter values shown in Table 5. We explicitly distinguish between small (i.e., 1 KB and 2 KB) and large request sizes (i.e., 36 KB and 40 KB).

As shown in Figure 10, the predictive power of the models for extrapolation scenarios differs significantly for smaller and larger request sizes especially for throughput predictions. Scenarios with smaller request sizes are predicted fairly well for response time. The error for the best performing model Cubist is less than 15%. Throughput predictions, however, exhibit much higher errors. This is due to the large performance overhead of serving many small requests simultaneously. The file system introduces significant overhead since the standard extent sizes (in which data is stored) are 4 KB. Furthermore, the optimization strategies in the various layers (especially the I/O scheduler) are very eager to optimize the requests. The latter is difficult since multiple threads issue requests simultaneously. This effect is multiplied by the penalty introduced by frequent backend RAID array accesses if the file set size is larger than the NVC and the VC, respectively. On the other hand, scenarios with larger request sizes are predicted very well. This is notable since the models have to cope with unexplored regions in the parameter space. Especially, Cubist predicts response times with less than 9% error and throughput with approximately 16% error. Also, MARS performs impressively well with approximately 11% error for response time and approximately 21% error for throughput.

Table 4: Interpolation Setup Configuration

Parameter	Range
I/O scheduler	{CFQ, NOOP}
File set size	[1 GB, 100 GB] rounded to multiples of 16 MB
Request size	[4 KB, 32 KB] rounded to multiples of 512 bytes
Access pattern	{random, sequential}
Read percentage	[25%, 100%] for read requests [0%, 75%] for write requests

Table 5: Extrapolation Setup Configuration

Parameter	Values
I/O scheduler	{CFQ, NOOP}
File set size	{512 MB, 768 MB, 110 GB, 120 GB}
Request size	{1 KB, 2 KB, 36 KB, 40 KB}
Access pattern	{random, sequential}
Read percentage	{15%, 20%, 80%, 85%}

Summary. Overall, the performance models exhibited good accuracy and predictive power. To highlight the key aspects, the models managed to effectively capture the performance influences with very high goodness-of-fit characteristics. In interpolation scenarios, the models exhibited excellent performance prediction accuracy for arbitrarily chosen configuration scenarios. Here, MARS and Cubist performed best with approximately 7%-8% prediction error. Also, in extrapolation scenarios, the models showed promising results. Except for throughput predictions for smaller request sizes, the best performing model Cubist had a prediction error of only 9%-16%.

6.2 Workload Distribution

We now evaluate scenarios where the workload is distributed on multiple co-located VMs as done for example in the context of server consolidation. We distribute the workload evenly across two and three VMs with shared cores, i.e., both the number of threads and the file set size are scaled inversely by the number of VMs. We explicitly distinguish between the different I/O scheduler implementations as well as between random and sequential workloads. Thus, having 100 randomly chosen configurations based on the parameter ranges in Table 4, we conduct 25 measurements for each

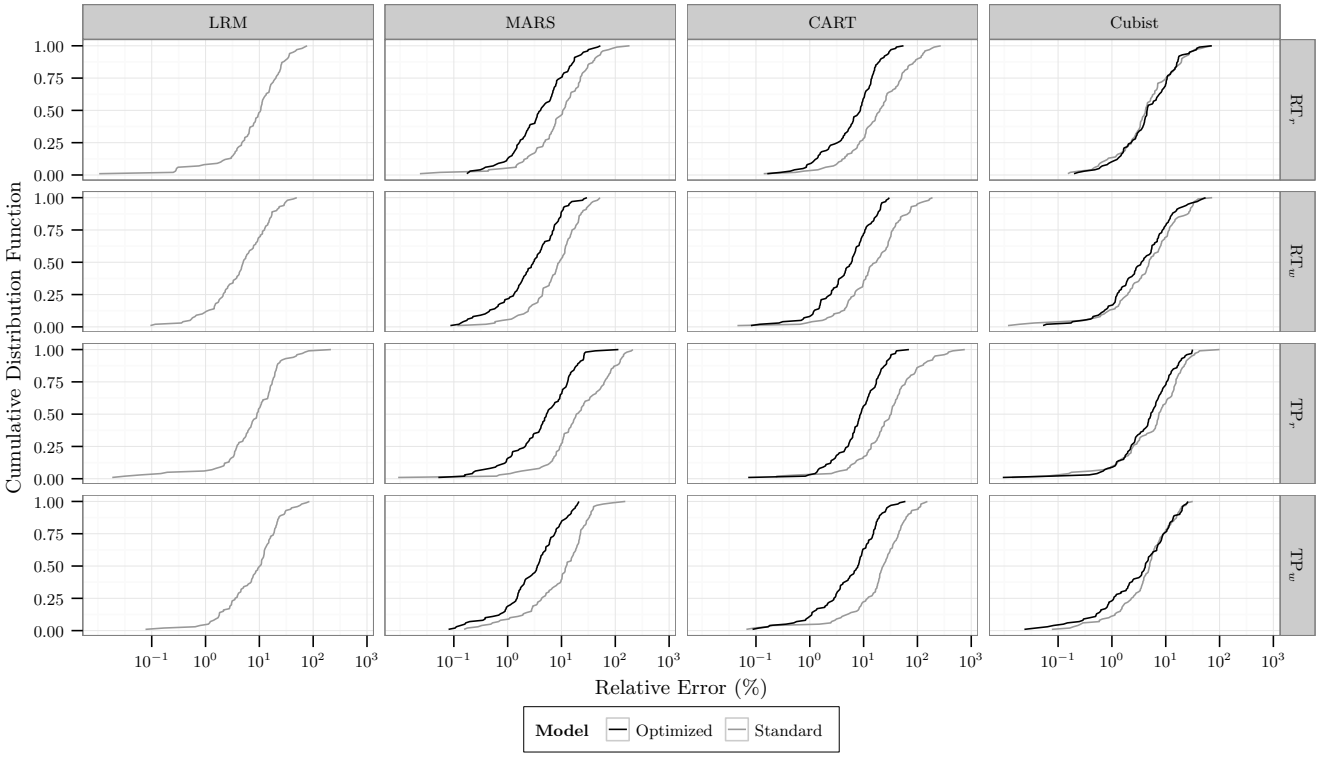
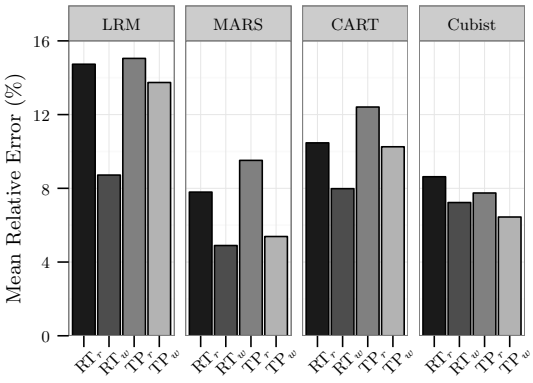
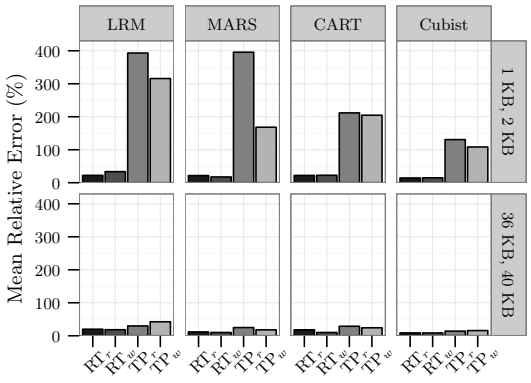


Figure 8: Cumulative Distribution Function of the Relative Error of the Performance Models (Note the Logarithmic x-Axis)



Model	Mean Relative Error (%) RT$_r$	RT$_w$	TP$_r$	TP$_w$	$\frac{1}{4}\sum$
LRM	14.73	8.72	15.05	13.74	13.06
MARS	7.80	4.90	9.52	5.39	6.90
CART	10.47	7.98	12.41	10.26	10.28
Cubist	8.63	7.23	7.75	6.44	7.51

Figure 9: Prediction Accuracy of Interpolation

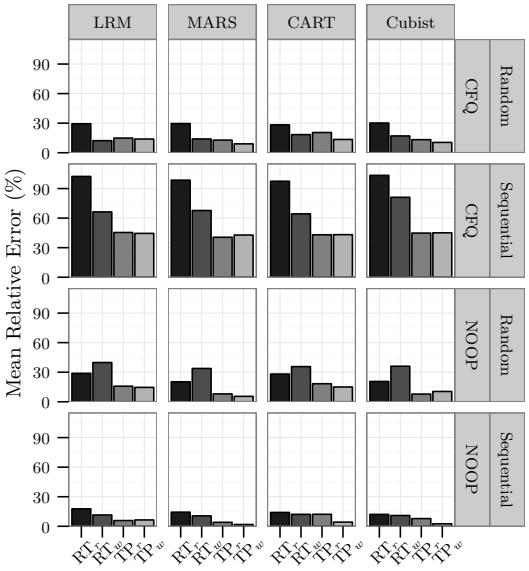


Model	Mean Relative Error (%) RT$_r$	RT$_w$	TP$_r$	TP$_w$	$\frac{1}{4}\sum$
Request Size: 1 KB, 2 KB					
LRM	22.36	33.65	393.19	315.81	191.25
MARS	21.79	17.54	395.66	168.26	150.81
CART	22.18	22.59	211.69	204.60	115.27
Cubist	14.47	14.96	130.76	108.38	67.14
Request Size: 36 KB, 40 KB					
LRM	19.90	18.34	29.77	42.47	27.62
MARS	11.66	10.00	24.86	17.98	16.13
CART	18.10	9.96	28.85	24.25	20.29
Cubist	8.70	8.52	14.12	15.74	11.77

Figure 10: Prediction Accuracy of Extrapolation

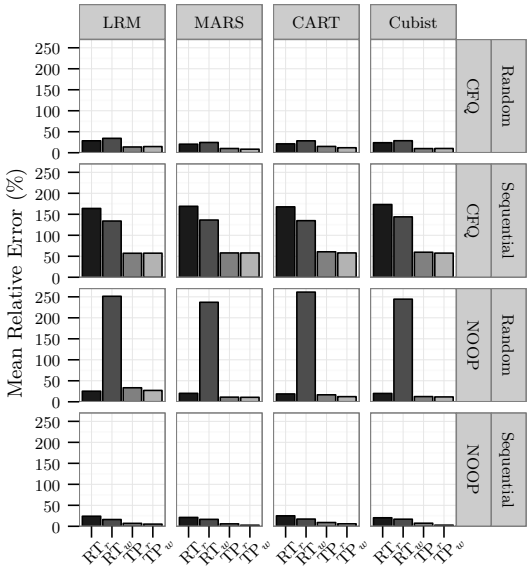
CFQ & NOOP scheduler and random & sequential workload combination. These measurements are used to evaluate the prediction error for each model. Note that the models are still extracted from measurements obtained in one VM to show the effectiveness of our approach.

For two and three VMs, respectively, Figure 11 and Figure 12 show the mean relative error for all models structured according to the chosen I/O scheduler and workload pattern. Since CFQ performs significant optimizations that depend on the current load, the model extracted from measurements in one VM does not fit well with the measurements conducted in multiple VMs for sequential workload. For random workload, however, this effect is less significant. Response time is predicted fairly well and especially throughput is predicted very well. The NOOP scheduler performs only optimizations based on request splitting and merging. Thus, the model extracted from measurements in one VM fits much better for multiple VMs in most cases. Only for write response times,



Model	Mean Relative Error (%) RT_r	RT_w	TP_r	TP_w	$\frac{1}{4}\sum$
Scheduler: CFQ, Random Requests					
LRM	29.42	12.24	14.86	13.88	17.60
MARS	29.62	13.96	12.80	9.01	16.35
CART	28.27	18.39	20.51	13.44	20.15
Cubist	30.17	17.03	13.24	10.47	17.73
Scheduler: CFQ, Sequential Requests					
LRM	102.32	66.32	45.42	44.50	64.64
MARS	98.55	67.73	40.58	42.84	62.43
CART	97.45	64.34	43.08	43.20	62.02
Cubist	103.40	81.14	44.81	45.10	68.61
Scheduler: NOOP, Random Requests					
LRM	28.85	39.83	15.79	14.58	24.76
MARS	20.31	33.82	7.97	5.48	16.90
CART	28.11	35.70	18.30	15.00	24.28
Cubist	20.61	36.07	7.78	10.46	18.73
Scheduler: NOOP, Sequential Requests					
LRM	17.71	11.47	5.82	6.43	10.36
MARS	14.35	10.57	4.00	1.82	7.69
CART	14.05	12.13	12.19	4.22	10.65
Cubist	12.08	10.95	7.81	2.49	8.33

Figure 11: Prediction Accuracy of the One-VM-Models when the Workload is Distributed on Two VMs



Model	Mean Relative Error (%) RT_r	RT_w	TP_r	TP_w	$\frac{1}{4}\sum$
Scheduler: CFQ, Random Requests					
LRM	28.43	34.32	13.74	14.66	22.79
MARS	20.37	24.44	10.08	8.40	15.82
CART	21.12	28.34	14.97	11.88	19.08
Cubist	23.88	28.60	9.89	10.13	18.12
Scheduler: CFQ, Sequential Requests					
LRM	163.93	133.93	57.13	57.37	103.09
MARS	169.06	136.32	57.91	57.85	105.29
CART	167.72	134.81	60.63	57.98	105.29
Cubist	173.29	143.90	59.63	57.51	108.58
Scheduler: NOOP, Random Requests					
LRM	25.21	251.33	33.34	27.04	84.23
MARS	20.14	237.03	11.02	10.64	69.71
CART	18.65	261.30	16.55	12.19	77.17
Cubist	19.85	244.44	12.32	11.58	72.05
Scheduler: NOOP, Sequential Requests					
LRM	24.01	16.13	7.04	5.17	13.09
MARS	21.16	16.49	5.83	2.59	11.52
CART	24.95	17.23	8.99	5.87	14.26
Cubist	20.30	16.89	7.35	2.70	11.81

Figure 12: Prediction Accuracy of the One-VM-Models when the Workload is Distributed on Three VMs

the model exhibits a high error particularly for larger file set sizes exceeding the storage cache. We conclude that the frequent cache misses and write request optimizations in the storage system lead to higher prediction errors. Nonetheless, in most cases the performance model still provides a valid approximation of the system performance.

Summary. Distributing the workload among multiple VMs leads to complex optimization and virtualization effects. While our models were not tuned for these effects, they still provided promising prediction results in most scenarios. In some scenarios, the error was evident and expected as, e.g., the CFQ scheduler is known for very active and eager request optimization. Still, the mean error remains less than 10%-20% for the MARS and Cubist models in most cases. While further in-depth analysis of the virtualization and scheduler influences is out of scope of this paper, the results are very encouraging.

6.3 Regression Optimization

To evaluate the improvements in model accuracy achieved through our regression optimization algorithm, we compare the accuracy of the models when using the optimized regression parameters vs. the standard parameters, respectively. We evaluate the performance prediction error for each model with 100 random configurations based on the ranges shown in Table 4, cf. Section 6.1.

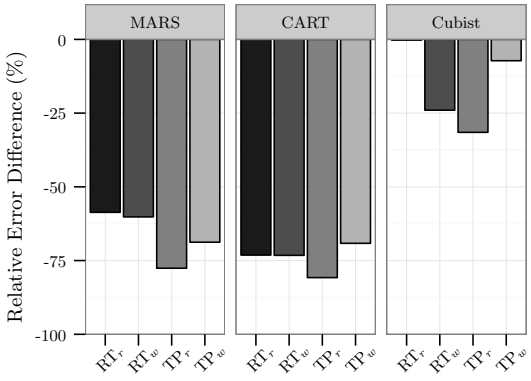
In Figure 13, we show the reduction in error for each model. Overall, the results show significant improvements of the model accuracy, i.e., reduced modeling error. Especially MARS and CART benefit from the parameter optimization exhibiting an error reduction of 66.30% and 74.08%, respectively. The improvement of Cubist is less, however, an error reduction of approximately 16% is still observed. Interestingly, the highest error reduction was achieved for read throughput predictions across all models. The MARS

models showed most improvements for throughput metrics, while the CART models were almost evenly improved across all metrics. Surprisingly, only a small error reduction was achieved for Cubist's read response time predictions, whereas significant improvements were seen for write response time and read throughput predictions.

In Figure 8, we evaluate the error distribution of the different techniques in the form of empirical cumulative distribution functions. The results show that our optimization especially eliminated large prediction errors. Furthermore, the results confirm that the optimization especially improved CART and MARS with a decisive reduction in large errors. For Cubist, the improvement was less than for the other techniques, however, the error reduction is still evident. This is due to the fact that the technique performs competitively well even with standard parameters and the optimization is primarily able to decrease the number of extreme errors. Still, the optimization process was key for achieving the high quality results of the MARS and the Cubist models presented in Section 6.1 and Section 6.2. These two models exhibited the highest accuracy in almost all of the considered evaluation scenarios

Finally, we evaluate the statistical significance of the optimization results. In a *paired t-test*, we evaluate the mean of each difference of the prediction error of the respective regression technique between the optimized and the standard parameter values. Thus, for each regression technique the null hypothesis H_0 is that the true mean of differences of the prediction error is equal to zero. For the t-test, the p-value of both MARS and CART is less than $2.2e^{-16}$ and the p-value of Cubist is 0.0003394. Thus, H_0 is rejected confirming that the optimization is statistically significant for every considered regression technique.

Summary. Overall, the parameter optimization was crucial for creating effective performance models. The mean error reductions of approximately 16%, 66%, and 74% for Cubist, MARS, and CART, respectively, lead to performance models with high accuracy and predictive power. Especially the improvements for MARS lead to accurate models that, together with the Cubist models, exhibited the best prediction accuracy in our scenarios. Furthermore, the improvement achieved by the parameter optimization was statistically significant for every considered regression technique.



Model	Relative Error Difference (%) RT_r	RT_w	TP_r	TP_w	$\frac{1}{4}\sum$
MARS	−58.64	−60.20	−77.59	−68.78	−66.30
CART	−73.14	−73.24	−80.78	−69.15	−74.08
Cubist	−0.25	−24.01	−31.52	−7.23	−15.75

Figure 13: Error Reduction by Parameter Optimization

7. RELATED WORK

Many general modeling techniques for storage systems exist, e.g., [7, 11, 20, 21, 28], but they are only shortly mentioned here as our work is focused on virtualized environments.

The work closely related to the approach presented in this paper can be classified into two groups. The first group is focused on modeling storage performance in virtualized environments. Here, Kraft et al. [17] present two approaches based on queueing theory to predict the I/O performance of consolidated virtual machines. Their first, trace-based approach simulates the consolidation of homogeneous workloads. The environment is modeled as a single queue with multiple servers having service times fitted to a Markovian Arrival Process (MAP). In their second approach, they predict storage performance in consolidation of heterogeneous workloads. They create linear estimators based on mean value analysis (MVA). Furthermore, they create a closed queueing network model, also with service times fitted to a MAP. Both methods use monitored measurements on the block layer that is lower than typical applications run. In [1], Ahmad et al. analyze the I/O performance in VMware's ESX Server virtualization. They compare virtual to native performance using benchmarks. They further create mathematical models for the virtualization overhead. The models are used for I/O throughput degradation predictions. To analyze performance interference in a virtualized environment, Koh et al. [16] manually run CPU bound and I/O bound benchmarks. While they develop mathematical models for prediction, they explicitly focus on the consolidation of different types of workloads, i.e., CPU and I/O bound. By applying different machine learning techniques, Kundu et al. [19] use artificial neural networks and support vector machines for dynamic capacity planning in virtualized environments. Further, Gulati et al. [10] present a study on storage workload characterization in virtualized environments, but perform no performance analysis.

The second group of related work deals with benchmarking and performance analysis of virtualized environments not specifically targeted at storage systems. Hauck et al. [13] propose a goal-oriented measurement approach to determine performance-relevant infrastructure properties. They examine OS scheduler properties and CPU virtualization overhead. Huber et al. [14] examine performance overhead in VMware ESX and Citrix XenServer virtualized environments. They create regression-based models for virtualized CPU and memory performance. In [3], Barham et al. introduce the Xen hypervisor comparing it to a native system as well as other virtualization plattforms. They use a variety of benchmarks for their analysis to quantify the overall Xen hypervisor overhead. Iyer et al. [15] analyze resource contention when sharing resources in virtualized environments. They focus on analyzing cache and core effects.

8. CONCLUSION

Summary. We presented a measurement-based performance prediction approach for virtualized storage systems. We created optimized performance models based on statistical regression techniques capturing the complex behaviour of the virtualized storage system. We proposed a general heuristic search algorithm to optimize the parameters of regression

techniques. This algorithm is not limited to a certain domain and can be used as a general regression optimization method. We applied our optimization approach and created performance models based on systematic measurements using four regression techniques: LRM, MARS, CART, and Cubist. We evaluated the models in different scenarios to assess their prediction accuracy and the improvement achieved by our optimization approach. The scenarios comprised interpolation and extrapolation scenarios as well as scenarios when the workload is distributed on multiple virtual machines. We evaluated the models in a real-world environment based on IBM System z and IBM DS8700 server hardware. In the most typical scenario, the interpolation ability of the models were excellent with our best models having less than 7% prediction error for MARS and less than 8% for Cubist. The extrapolation accuracy was promising. Except for throughput predictions of 1 KB and 2 KB requests, the prediction error of our best model Cubist was always less than 16%. Furthermore, using our models we were able to successfully approximate the performance if the workload is distributed on two and three virtual machines. The average prediction error of both MARS and Cubist was less than 20% in most scenarios. Finally, our optimization process showed to be crucial and reduced the prediction error by 16%, 66%, and 74% for Cubist, MARS, and CART, respectively, with statistical significance.

Lessons Learned. i) Our performance modeling approach was able to extract powerful prediction models. For interpolation and extrapolation scenarios, the models showed to have excellent prediction accuracy. ii) Even if the workload is distributed on several virtual machines, the performance models are able to approximate the expected performance in most cases. iii) Our optimization process showed to be key for the prediction accuracy of the models and achieved statistically significant improvements for every considered regression technique. iv) While LRM and CART showed to perform well, MARS and Cubist consistently showed the best fit and the best prediction accuracy.

Application Scenarios. Generally, our approach is targeted at the analysis and evaluation of virtualized storage systems. It is especially beneficial for complex systems and in cases that prohibit explicit fine-grained performance models due to, e.g., time constraints for the manual performance model creation, calibration, and validation. Our automated performance modeling approach can aid (e.g., the system developer) to measure the system once and extract a performance model that represents the environment. This model can be used in different scenarios (e.g., by customers) to assess the system characteristics and evaluate deployment decisions.

9. ACKNOWLEDGMENTS

This work was funded by the German Research Foundation (DFG) under grant No. RE 1674/5-1 and KO 3445/6-1. We especially thank the Informatics Innovation Center (IIC)[6] for providing the system environment of the IBM System z and the IBM DS8700.

10. REFERENCES

[1] I. Ahmad, J. Anderson, A. Holler, R. Kambo, and V. Makhija. An analysis of disk performance in VMware ESX server virtual machines. In *WWC-6*, 2003.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, 2003.

[4] L. Breiman. Random Forests. *Machine Learning*, 45, 2001.

[5] L. Breiman, J. Friedman, C. J. Stone, and R. Olshen. *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Chapman & Hall, 1984.

[6] D. Bruhn. Modeling and Experimental Analysis of Virtualized Storage Performance using IBM System z as Example. Master's thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2012.

[7] J. S. Bucy, J. Schindler, S. W. Schlosser, G. R. Ganger, and Contributors. *The DiskSim Simulation Environment - Version 4.0 Reference Manual*. Carnegie Mellon University, Pittsburgh, PA, 2008.

[8] B. Dufrasne, W. Bauer, B. Careaga, J. Myyrrylainen, A. Rainero, and P. Usong. IBM System Storage DS8700 Architecture and Implementation. `http://www.redbooks.ibm.com/abstracts/sg248786.html`, 2010.

[9] J. H. Friedman. Multivariate Adaptive Regression Splines. *Annals of Statistics*, 19(1):1–141, 1991.

[10] A. Gulati, C. Kumar, and I. Ahmad. Storage workload characterization and consolidation in virtualized environments. In *VPACT '09*.

[11] P. Harrison and S. Zertal. Queueing models of RAID systems with maxima of waiting times. *Performance Evaluation*, 64:664–689, 2007.

[12] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2nd edition, 2011.

[13] M. Hauck, M. Kuperberg, N. Huber, and R. Reussner. Ginpex: deriving performance-relevant infrastructure properties through goal-oriented experiments. In *QoSA-ISARCS '11*.

[14] N. Huber, M. von Quast, M. Hauck, and S. Kounev. Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments. In *CLOSER '11*.

[15] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell. VM3: Measuring, modeling and managing VM shared resources. *Computer Networks*, 53:2873–2887, 2009.

[16] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An Analysis of Performance Interference Effects in Virtual Environments. In *ISPASS '07*.

[17] S. Kraft, G. Casale, D. Krishnamurthy, D. Greer, and P. Kilpatrick. Performance Models of Storage Contention in Cloud Environments. *SoSyM*, 2012.

[18] M. Kuhn, S. Witson, C. Keefer, and N. Coulter. Cubist Models for Regression. `http://cran.r-project.org/web/packages/Cubist/vignettes/cubist.pdf`, 2012. Last accessed: Oct 2012.

[19] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta. Modeling Virtualized Applications using Machine Learning Techniques. In *VEE '12*.

[20] A. S. Lebrecht, N. J. Dingle, and W. J. Knottenbelt. Analytical and Simulation Modelling of Zoned RAID Systems. *The Computer Journal*, 54:691–707, 2011.

[21] E. K. Lee and R. H. Katz. An analytic performance model of disk arrays. *SIGMETRICS Perform. Eval. Rev.*, 21(1), 1993.

[22] P. Mell and T. Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[23] Q. Noorshams, S. Kounev, and R. Reussner. Experimental Evaluation of the Performance-Influencing Factors of Virtualized Storage Systems. In *EPEW '12*, volume 7587 of *LNCS*. Springer, 2012.

[24] J. R. Quinlan. Combining Instance-Based and Model-Based Learning. In *ICML '93*.

[25] J. R. Quinlan. Learning with Continuous Classes. In *AI '92*. World Scientific.

[26] RuleQuest Research Pty Ltd. Data Mining with Cubist. `http://rulequest.com/cubist-info.html`, 2012. Last accessed: Oct 2012.

[27] The R Project for Statistical Computing. `http://www.r-project.org/`, 2012. Last accessed: Oct 2012.

[28] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage Device Performance Prediction with CART Models. In *MASCOTS '04*.

[6] `http://www.iic.kit.edu/`