PEMOGEN: Automatic Adaptive Performance Modeling during Program Runtime

Arnamoy Bhattacharyya
ETH Zurich
arnamoy.bhattacharyya@inf.ethz.ch

Torsten Hoefler
ETH Zurich
htor@inf.ethz.ch

ABSTRACT

Traditional means of gathering performance data are tracing, which is limited by the available storage, and profiling, which has limited accuracy. Performance modeling is often used to interpret the tracing data and generate performance predictions. We aim to complement the traditional data collection mechanisms with online performance modeling, a method that generates performance models while the application is running. This allows us to greatly reduce the storage overhead while still producing accurate predictions. We present PEMOGEN, our compilation and modeling framework that automatically instruments applications to generate performance models during program execution. We demonstrate the ability of PEMOGEN to both reduce storage cost and improve the prediction accuracy compared to traditional techniques such as least squares fitting. With our tool, we automatically detect 3,370 kernels from fifteen NAS and Mantevo applications and model their execution time with a median coefficient of variation ($\bar{R}^2$) of 0.81. These automatically generated performance models can be used to quickly assess the scaling and potential bottlenecks with regards to any input parameter and the number of processes of a parallel application.

1. INTRODUCTION

Performance analysis is the bread and butter of performance engineers. The optimization process can often be divided into performance data collection, interpretation to find bottlenecks, and code changes to overcome those bottlenecks. The first step, data collection, typically utilizes application tracing or profiling. The collected data is then analyzed with the help of performance tools. Performance engineers derive mental models at various complexities ranging from back-of-the-envelope to highly complex performance models to represent application performance. Those models then guide the tuning of the target application code, for example, algorithmic optimizations, architecture optimization, middleware and runtime optimization, or policy optimization.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2809-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2628071.2628100 .

Various offline machine-learning techniques such as Neural networks or Support Vector Machines (SVM) have been used to build predictor models that can be used to extrapolate or interpolate application performance and scaling based on trace data [26]. However, most of the machine learning predictor models used so far are complex functions consisting of many model parameters, making the models hard to interpret. The generation of machine learning models needs sufficiently much training data. The gathering of the huge amount of training data comes with two costs: (1) a series of training runs has to be performed that might not be feasible for long-running HPC application, and (2) the storage space of the profiling data can be potentially huge.

Only a small number of tools support performance modeling to help the user to guide decisions and predict bottlenecks outside the executed configurations [7]. Semi-empirical performance modeling [19] is used to generate interpretable performance models that can be presented to users so that they can estimate how the program's performance changes with the change in values of the input parameters. Optimizing compilers can also use the generated models to guide decisions. Empirical or semi-empirical performance models work in a similar fashion to the machine-learning techniques in the sense that the performance models of the kernels are built after collecting the training data offline [7] and therefore also suffer from storage cost.

In this work, we propose to automatically *learn* performance models during normal application execution to eliminate storage costs and reduce runtime overheads. We develop a tool that automatically instruments applications for modeling and a runtime library for performance modeling based on advances in online statistical learning methods. Our generated binaries are completely self-contained and fully equipped for automatic model generation. The data collection and generation can be controlled by the user and is stopped once the models reach a specified quality. An overview of our tool-chain is shown in Figure 1. We now briefly discuss LASSO, a statistical technique that can be used to generate interpretable models.

Least absolute shrinkage and selection operator (LASSO).

is a statistical technique to generate interpretable regression models by eliminating parameters that are not significant in the model construction. Recently, Garrigues et al. [14] proposed a homotopy algorithm to use LASSO in an

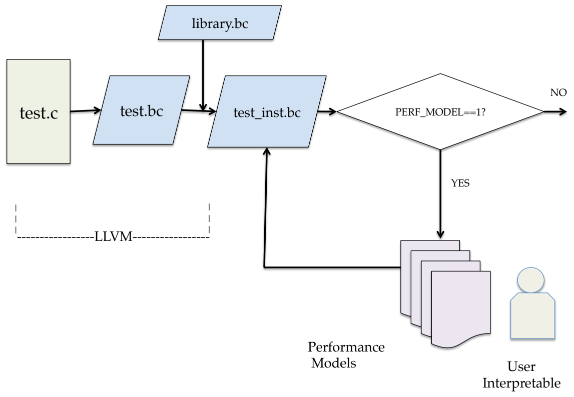


Figure 1: Work-flow of PEMOGEN. The code is compiled and instrumented using LLVM [25]. Based on the value of an environment variable PERF_MODEL, the instrumented code generates the performance models for kernels. The generated models can again be read by the code for model update and/or prediction.

online setting so that the model update can be performed as new data arrives one at a time. We show how to use their algorithm to build an application performance model automatically and transparently from the user. The user has to only supply the names of the input parameters that influence the execution time of a program and the desired threshold for goodness of fit of the generated model(s). A subset of the program input parameters become the parameters of the generated model. These two information are required because identifying the model parameters require knowledge from the domain expert and it's impossible to predict the accuracy of the performance model required by the user.

Research Contribution.

Our contributions to the state of the art are as follows:

- We propose a new static representation of the program called the Loop-Call Graph (LCG) that helps to automatically identify the *kernels* in the application.
- We propose a methodology for adaptive model generation so that the profiling and model update can be turned on and off automatically based on the goodness of fit (adjusted R^2) of the generated performance model on new data.
- We combine these two techniques into PEMOGEN, an automatic tool to generate interpretable performance models dynamically and transparently from the user.
- We present the results of model generation for the kernels in the NAS and Mantevo benchmarks.

2. METHODOLOGY

Automatically generating human-understandable performance models for codes involves several steps: (1) kernel identification, (2) identifying critical input parameters, (3) collect performance data, and (4) select and parameterize model hypotheses. Those steps are similar to but not identical to the manual steps proposed in earlier research [18,19]. We will now describe each of these steps in detail.

2.1 Automatic Identification of Kernels

The first step before building a performance model automatically is to determine the portions in the program for which the performance models have to be generated. These program portions are called *kernels*. The cumulative performance of all the *kernels* gives an estimate of the performance of the program for a set of program parameter values.

For the identification of *kernels*, we consider well defined program structures: natural *loops* and *functions*. We define a new static graph representation of the program called a *Loop-Call Graph (LCG)*. If a program P has a set of functions $F = (f_1, f_2, \ldots, f_{n_1})$ and a set of loops $L = (l_1, l_2, \ldots, l_{n_2})$, then the LCG is a directed graph that can be formally defined as:

Definition 1. *An LCG is a graph $G = \langle V, E \rangle$ where*

- $V = F \cup L$
- $E = (e_1, e_2, \ldots, e_{n_3})$ *where* $e_i = (v_1, v_2) \wedge v_1, v_2 \in V$

Where (v_1, v_2) means that control flows from v_1 to v_2. We use the LCG to aid the dynamic construction of *kernels* for different call sites and contexts. We define kernel as:

Definition 2. *A kernel k is a sub-graph $G' = \langle V', E' \rangle$ of the LCG G where*

- $V' \subseteq V$
- $E' \subseteq E \wedge ((v_1, v_2) \in E' \Rightarrow v_1, v_2 \in V')$

Each *kernel* is constructed from the dynamic call-chain starting from the entry point of the program. A kernel is composed of a subset of nodes of the LCG that are connected by a dynamic calling relation, denoted as $\rightarrow$, in the program.

To illustrate how we construct the LCG, consider the following pseudocode:

```
int my_func() {
        for(...) S1;
}
int main() {
        for(...) my_func();
        for(...) S2;
        my_func();
}
```

S1 and S2 are arbitrary sets of statements that do not contain additional loops. Figure 2 shows the LCG of this code.

This program has a set of functions $F = (a, b)$, and a set of loops $L = (a_1, a_2, b_1)$. Where 'a' and 'b' represent 'main' and 'my_func', respectively, and a_i denotes the ith loop in function a. A specific execution of our example program may lead to the following set of kernels at runtime: $K = \{(a), (a \rightarrow a_1), (a \rightarrow a_1 \rightarrow b), (a \rightarrow a_1 \rightarrow b \rightarrow b_1), (a \rightarrow b), (a \rightarrow b \rightarrow b_1), (a \rightarrow a_2)\}$.

2.2 Values of Model Parameters

After all kernels have been identified, the next step is to identify the input parameters that should be used to build a performance model for a specific kernel. The first task is to assemble a list of all input parameters that influence the

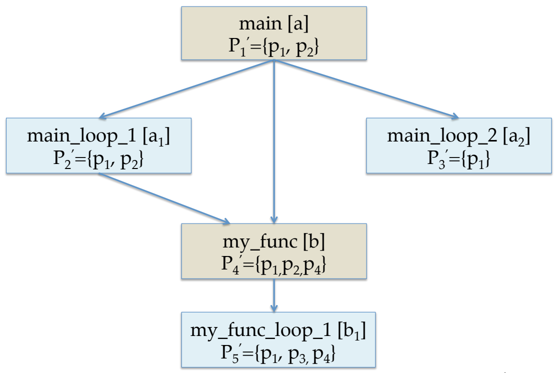


Figure 2: Example of a Loop Call Graph (LCG).

runtime of the application. We call such parameters *critical (input) parameters* [19].

Critical parameters should be scalar values such as sizes of dimensions or number of iterations. If the execution time of the program is determined by an input file or a vector, then it should be condensed into the smallest number of scalar critical parameters (e.g., if the input file is a sparse matrix, the critical parameter could be the number of non-zero elements in the matrix). A domain expert has to determine the complete set of parameters and supply them to PEMOGEN. We identify the set of parameters as $P = (p_1, p_2, \ldots, p_n)$.

Not all parameters may influence each kernel. Thus, PEMOGEN relates the specified parameters to the source code and uses static analysis to determine the parameters $P_i' \subseteq P$ that influence the runtime of each specific kernel k_i. PEMOGEN performs pointer analysis to construct use-def chains [3] from all statements in each function and loop in V and checks which critical parameters are used. It conservatively adds the parameter to the parameter set P_i' for may-dependencies. Finally, we use the following maps from each kernel to its parameter set during runtime to collect the profile $f_i : k_i \leftarrow P_i'$, $0 < i < |K|$.

The profiled dynamic value of the input parameters and the static mapping information between the kernels and their parameter sets are used in the construction of performance models of kernels.

2.3 Target Metrics

Our tool can generate kernel models for various metrics such as hardware performance counters, the number of messages communicated, the size of messages communicated, and the execution time. Without loss of generality, We choose to present the absolute execution time [16]. It is often the noisiest metric and thus the most challenging one for any modeling algorithm. The absolute execution time is the time elapsed for the execution of instructions that are parts of nested kernel. For example, in Figure 2, while calculating the absolute execution time of the kernel $a \rightarrow b$, we do not include the execution time of the kernel $a \rightarrow b \rightarrow b_1$ though they are nested.

2.4 Dynamic Construction of Performance Models

Shrinkage is a method in statistics where a penalty is applied to the coefficients of the regression predictor model and thus the coefficients values are shrunk to bring them close to zero. Shrinkage techniques are useful in cases where there is high correlation between the model parameters or in cases where some parameters cause the model to overfit. By selecting the parameter that best describe the model, these shrinkage methods often result in better prediction accuracy. Ridge regression technique [22] adds a L_2 penalty to the coefficients and thus shrunk the coefficients of models close to '0' but does not shrink any coefficient to '0' and therefore cannot eliminate any parameter. Therefore, although the ridge regression technique builds model with better prediction accuracy, the generated models are not different from the ordinary least square regression in terms of user interpretability.

By adding a L_1-penalty to the regression coefficients, the *Least absolute shrinkage and selection operator* (LASSO) technique [36] bridges the gap between user interpretability and prediction accuracy of the generated models. Due to the ability of LASSO in the generation of interpretable models, the LASSO method generated significant interest in statistics [9, 36], signal processing [10, 13] and machine learning [17, 31] communities.

We utilize the recently developed online version of the Least Absolute Shrinkage and Selection Operator (LASSO) method. We call this method oLASSO in the remainder of this paper.

First, we describe the regular (offline) LASSO briefly: LASSO assigns an L_1 penalty to the coefficients of input parameters (predictors) which leads to sparse solutions and thus achieves results that are easier to interpret. LASSO accepts n training examples or observations $(y_i, x_{ij}) \in \mathbb{R} \times \mathbb{R}^m$, $i = (1, 2, \ldots, n)$ and $j = (1, 2, \ldots, m)$. We assume we have m input parameters for the model. Here y is the performance metric we want to model and x is the set of input parameters. We wish to fit a linear model to predict the response y_i as a function of x_{ij} and a feature vector $\theta \in \mathbb{R}^m$, $y_i = x_i^T\theta + v_i$, where v_i represents the noise in the observation. The LASSO optimization problem is given by

$$\min_{\theta} \frac{1}{2} \sum_{i=1}^{n} (x_i^T\theta - y_i)^2 + \mu_n \|\theta\|_1 \tag{1}$$

where μ_n is a regularization parameter. The solution of Equation 1 is typically sparse, i.e. the solution θ has few entries that are non-zero, and therefore identifies which dimensions in x_i are useful to predict the response y_i. The original LASSO proposal [36] did not have the additional $\frac{1}{2}$ multiplied with the first term of Equation (1). It was later shown that the LASSO problem is equivalent to the Basis Pursuit Denoising (BPDN) optimization problem which needs the multiplier $\frac{1}{2}$. Also BPDN representation of LASSO makes it algorithmically easier to solve. LASSO is particularly useful in our case because LASSO is used in cases where the number of observations is less than the number of predictor variables and it generates a model that is more interpretable than Ordinary Least Square Regression [8] or Ridge Regression [22].

The offline LASSO algorithm works on a set of precomputed observations (training data). To use LASSO in an online fashion, we choose a homotopy algorithm proposed by Garrigues et al. [14]. Following this algorithm, the model is updated as dynamic training data $(y_i, x_{ij})_{i=1,\ldots,n}$ arrive one at a time. Let $\theta^{(n)}$ be the solution of the LASSO after observing n training examples and $\theta^{(n+1)}$ the solution after observing a new dynamic data point $(y_{n+1}, x_{n+1.j}) \in \mathbb{R} \times \mathbb{R}^m$.

Let $X \in \mathbb{R}^{n\times m}$ be a matrix whose i^{th} row is equal to x_i^T and $y = (y_1, \ldots, y_n)^T$, they introduce the following optimization problem that allows them to compute a homotopy from θ^n to θ^{n+1}:

$$\theta(t,\mu) = \underset{\theta}{\operatorname{argmin}} \frac{1}{2} \left\| \begin{pmatrix} X \\ tx_{n+1}^T \end{pmatrix} \theta - \begin{pmatrix} y \\ ty_{n+1} \end{pmatrix} \right\|_2^2 + \mu \|\theta\|_1 \quad (2)$$

The algorithm computes a path from $\theta^{(n)}$ to $\theta^{(n+1)}$ in two steps where $\theta^{(n)} = \theta(0, \mu_n)$ and $\theta^{(n+1)} = \theta(1, \mu_{n+1})$.

1. Vary the regularization parameter from μ_n to μ_{n+1} with $t = 0$. This amounts to computing a piecewise linear regularization path between μ_n to μ_{n+1} [11, 29, 32].

2. Vary the parameter t from 0 to 1 with $\mu_n = \mu_{n+1}$.

The solution of the LASSO problem can be computed once the active set (the indices in θ that have non-zero coefficients) and signs of the coefficients are known [14]. Therefore the intuition behind the two-step algorithm is to discover the active set and signs of the coefficients for computing $\theta^{(n+1)}$ at $t = 1$. For achieving this, the algorithm starts with the active set and signs available at $t = 0$, which is essentially calculated using LARS (step 1). Then the algorithm continues until a 'transition point' (value of t where the active set changes). At this point, the necessary updates to the active set and signs of the coefficients are met. These procedure iteratively continues until $t = 1$. Next the solution $\theta^{(n+1)}$ can be computed using the final active set and signs of the coefficients at $t = 1$.

2.5 Selection of the regularization parameter

The amount of regularization depends indeed on the variance of the noise present in the data which is not known a priori. It is therefore not obvious how to determine the amount of regularization. We use Garrigues' algorithm for choosing the best regularization parameter as new data arrives [14]. The algorithm uses $\mu_n = n\lambda_n$ such that λ_n is the weighting factor between the average mean squared error and the l_1-norm. The algorithm selects λ_n in a data-driven manner. The problem with n observations is given by:

$$\theta(\lambda) = \underset{\theta}{\operatorname{argmin}} \frac{1}{2n} \sum_{i=1}^{n} (x_i^T \theta - y_i)^2 + \lambda \|\theta\|^2 \quad (3)$$

As $\theta(\lambda)$ is piecewise linear, therefore its gradient can be computed unless λ is a transition point. If $err(\lambda) = (x_{n+1}^T \theta(\lambda) - y_{n+1})^2$ is the error on the new observation. they update to select λ_{n+1} is as follows:

$$\log \lambda_{n+1} = \log \lambda_n - \eta \frac{\partial err}{\partial \log \lambda}(\lambda_n) \quad (4)$$

$$\Rightarrow \lambda_{n+1} = \lambda_n \times \exp\left\{2n\eta x_{n+1,1}^T (X_1^T X_1)^{-1} v_1 (x_{n+1}^T \theta_1 - y_{n+1})\right\} \quad (5)$$

where the solution after n observations corresponding to the regularization parameter λ_n is given by and $v_1 = sign(\theta_1)$. Therefore the new observation is used as a test set, which allows the update of the regularization parameter before introducing the new observation by varying t from 0 to 1. The update is performed in the *log* domain to ensure that λ_n is always positive.

3. IMPLEMENTATION

PEMOGEN is implemented using LLVM [25] for performing the LCG construction and instrumentation. Figure 1 describes the execution flow of a program instrumented by PEMOGEN. The user can enable or disable the model generation by setting an environment variable. The model generation works transparently during the program runtime and the generated models for kernels are stored in a file that can be consulted by either the user or a performance monitoring and optimization system. In addition, following executions of the instrumented binary can also read the file to refine the model further.

3.1 Kernel Identification and Instrumentation

PEMOGEN uses the middle end of LLVM to analyze the intermediate representation (IR) of the program to build the LCG. Natural loops and functions are well-formed structures in the LLVM IR. After the identification of LCG nodes, PEMOGEN instruments the IR to prepare the IR for data collection of the intended metric. We implement the data collection functionality in a library that is automatically linked to the generated program executable.

Optionally, PEMOGEN can output the LCG in the dot graph language for visualization. If the program is compiled with debug information, the generated LCG contains code information such as source code file name and line number. This is particularly useful to map the kernels to the source code to identify specific latent performance problems.

3.2 Parameter Specification

The scalar model parameters have to be supplied by a domain expert. An LLVM pass instruments the source code so that the values of the model parameters are captured dynamically. The parameters can be of different types:

- **A member of a structure**: In this case, the parameter name has to be supplied along with the structure name.

- **A local variable in a function**: In this case, the parameter name has to be supplied along with the function name.

- **A global variable or a macro**: In this case, the parameter name is sufficient.

We use the following EBNF for the parameter file:

$$\begin{array}{rcl}
\langle\text{file}\rangle & \models & \langle\text{line}\rangle\ \langle\text{newline}\rangle \ \mid\ \langle\text{line}\rangle \\
\langle\text{line}\rangle & \models & \langle\text{word}\rangle\ \langle\text{space}\rangle\ \langle\text{flag}\rangle\ \langle\text{space}\rangle\ \langle\text{word}\rangle \\
\langle\text{flag}\rangle & \models & \langle 0\rangle \mid \langle 1\rangle \mid \langle 2\rangle \mid \langle 3\rangle \\
\langle\text{word}\rangle & \models & \langle\text{wchar}\rangle\ \langle\text{word}\rangle \ \mid\ \langle\text{wchar}\rangle \\
\langle\text{wchar}\rangle & \models & \langle\text{cchar}\rangle \ \mid\ \langle\text{ichar}\rangle \\
\langle\text{cchar}\rangle & \models & \text{A}\ldots\text{Z} \mid \text{a}\ldots\text{z} \mid 0\ldots 9 \\
\langle\text{ichar}\rangle & \models & \_
\end{array}$$

Each line contains three tokens: The first token is the parameter name as in the source code, the second token indicates the type of parameters ('0' for parameter inside structure, '1' for parameter inside function, '2' for global parameter, and '3' for parameter declared as macro), and the third token specifies the context (function name or struct name).

3.3 Mapping of Parameters to Kernels

We now discuss how we determine the set P_i' for each kernel k_i. We assume that the value of critical parameters does not change during program execution. For each specified critical parameter, PEMOGEN finds the statement in the IR that first accesses it. Then it performs a pointer analysis with all the access instructions in the body of each LCG node (a function or a natural loop) to find which kernels use the critical parameter. In the case where the pointer analysis reports a *may point-to* relation, PEMOGEN conservatively adds the name of the parameter in the parameter list of the kernel. PEMOGEN maps parameters to LCG nodes statically during compile time and then uses the static information to aggregate parameter lists for each kernel during runtime.

For parameters defined as macros, the tool conservatively adds them to the parameter list of all LCG nodes.

3.4 Constructing the oLASSO

PEMOGEN uses a reference implementation of the oLASSO algorithm [29] for constructing the online performance model. It selects the best predictor model from a pool of *model hypotheses*. We extended the Performance Model Normal Form (PMNF) as described by Calotoiu et al. [7] to use it with more than one parameter. Our Extended PMNF (EPMNF) for a parameter set P is given as:

$$f(P) = \sum_{i=1}^{|P|} \sum_{k=1}^{n} c_i . p_i^{j_{ik}} \log_2^{l_{ik}}(p_i) \tag{6}$$

This representation is, of course, not exhaustive, but proved practical in most scenarios since it is a consequence of how most computer algorithms are designed. A possible assignment of all c_{ik}, j_{ik} and l_{ik} in an EPMNF expression is called a *model hypothesis.*

The oLASSO method calculates the coefficients for each candidate model hypothesis. In this way, a number of coefficient vectors, one for each model hypothesis is generated. When a new measured value arrives, each model is tested using their goodness of fit on new values. If the goodness of fit of more than one model falls within an acceptable range (the acceptable fitting range, ϵ, has to be specified by the user), the model with fewer parameters is selected and the other models are discarded. If the selected model's fit goes above the acceptable range, we start building the whole set of model hypothesis again and continue updating them until we find a model whose fit falls within the specified ϵ.

3.5 Reducing the Overhead of Model Generation

There can be significant overhead for updating the models for *loops* and *functions* that are called a number of times from a call site. Because in that case, the instrumented code has to call the model update function every time a new data point is generated. This can potentially be a problem for inner loops that may have thousands of iterations.

To have a balance between the collection of data for a metric and model update, we call the model generating function on batches of data points.

3.6 Model Confidence

The model update is enabled or disabled based on the fitting of the generated model on new data and the user-supplied acceptable fitting range ϵ.

We also sample the test data for measuring the fitting. As mentioned in the previous step, the model is updated for a batch of x observations at once instead of calling the model update function each time a new observation is received. The next x data points are used as *test* data and the adjusted R-square (ARS) of the predictions by the model is calculated on the test data:

$$\mathrm{R}^2 \equiv 1 - \frac{\sum_{i=1}^{x} (y_i - f_i)^2}{\sum_{i=1}^{x} (y_i - \bar{y})^2} \tag{7}$$

$$\mathrm{ARS} = \mathrm{R}^2 - (1 - \mathrm{R}^2) \frac{m}{x - m - 1} \tag{8}$$

Where x and m are the test data batch size and number of parameters respectively. If for a program, we can't collect a batch of data points, we keep the metric data temporarily until we have a batch. Still this method has much less storage overhead than other online methods mainly the storage overhead from HPC applications come from *kernels* that run a large number of times, either in parallel or in a loop. But in our technique, that is a favourable case because we do not have to store temporary data for a large number of batches.

We use the state machine shown in Figure 3 to evaluate the confidence of a model hypothesis. We assign counters to each model hypothesis. When the model is built for the first time, the confidence value (counter) is initiated to '0'. With each successful prediction (if ARS $< \epsilon$, where ϵ is supplied by user) the confidence of the model in incremented by one. If the model's fit is above the user defined range, the models confidence in decremented by one. The three states of the

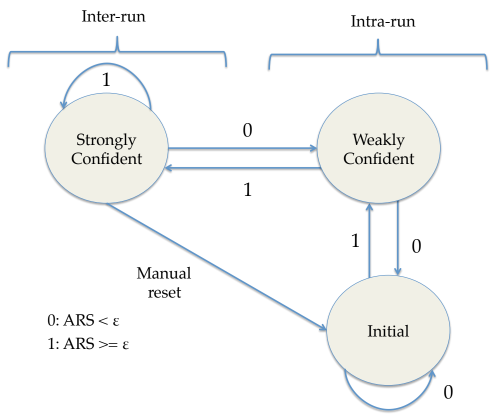


Figure 3: State Machine to Determine the Confidence of the Predictor Model.

model are:

- **Initial:** Each model is initiated with this state, with the counter value for model confidence being '0'. Models can also be reset to go back in initial confident state manually when a strongly confident model fails to fit well within a desired fitting range. The initial state of the model is updated each time a new test data batch arrives within an application run or across application runs (for kernels whose training batch does not reach value x within an application run).

Table 1: Applications from NAS [30] and Mantevo Benchmarks [1] and their input parameters.

Code	Description	Parameters (some abbreviated from original long names)
BT	Block tri-diagonal solver	`grid_points(1), grid_points(2), grid_points(3), niter, dt`
CG	Conjugate gradient, irregular memory access	`na, nonzer, niter, shift`
DC	Create data cube views	`input_tuples, attrnum, memoryLimit, dim, mnum`
EP	Embarrassingly parallel random number generator	`m, mk, mm, nn, nk, nq, epsilon, a, s`
FT	discrete 3D fast Fourier transform	`nx, ny, nz, maxdim, niter_default, ntotal, nxp, nyp, ntotalp`
IS	Bucket sorting	`{total_keys,max_key,num_buckets}_log_2, max_iters, test_array_size`
LU	Lower-upper Gauss-Seidel solver	`ipr, inorm, nitmax, dt, omega, tolnwt(1, 2, 3, 4, 5), nx, ny, nz`
MG	Multi-grid on a sequence of meshes	`lt, nit, nx, ny, nz`
SP	Scalar penta-diagonal solver	`nx, ny, nz, niter, dt_default`
UA	Unstructured adaptive mesh	`lelt, lmor, refine_max, fre, niter, nmxh, alpha`
Cloverleaf	Compressible Euler solver on a Cartesian grid	`states(energy, density, svel, yvel), end_time, end_step, x_cells, y_cells, {x,y}min, {x,y}max, {initial, max}_timestep, timestep_rise`
CoMD	Typical computations in molecular dynamics	`n{x,y,z}, {x,y,z}proc, nsteps, time_step, initial_{temp,delta}`
FE	Proxy application for unstructured finite element codes	`nx, ny, nz`
MiniGhost	Difference stencil in three dimensional domain	`nx, ny, nz, stencil, num_vars, perc_sum, num_sp, error_tol, rep_diff`
HPCCG	Approximation of unstructured implicit finite element	`nx, ny, nz`

- **Weakly Confident:** A initial state of a model hypothesis can become weakly confident when the model hypothesis' counter reaches the value '5'. We have found this value gives a good balance between model hypothesis updation overhead and keeping the ARS of the model within the user defined range for the set of benchmarks used. This state is also used within an application run to determine the confidence within a run.

- **Strongly confident:** In this state, the model updation as well as data collection for the *kernel* is turned off. Using this state gives us a change to optimize the data collection and model update overhead. We show the effect of this optimization in Section 4.6. The strongly confident state is an inter-run measure of confidence and therefore we keep a separate counter for the strongly confident state information. After the execution of the program with a certain set of parameter values, the strongly confidence counter of all the model hypotheses in weakly confident state is incremented by one. Once the strong confidence counter reaches the value '10', the model hypothesis goes to strongly confident state, where both the data collection and update is turned off. Of course using this method, the *kernels* whose behaviour largely varies according to different input parameter values can't be modeled. But while determining the performance of a program, the performance model of these *kernels* can be assumed to have any value for estimation.

There is another well known metric for determining model confidence, which is called root mean squared error (RMSE). But we did not use RMSE because by using ARS, we had an interpretable fit of the model on new data and could determine how good the built model is. As the kernels that have largely fluctuating behaviour for different parameter values are not targeted by our technique, the ARS on new data gives an estimate of the generated model's confidence in predicting the performance of non-fluctuating kernels.

4. EXPERIMENTAL EVALUATION

We applied PEMOGEN to the NAS parallel benchmarks [30] and the Mantevo benchmarks [1]. Table 1 provides an overview of all test programs and their critical parameters. The instrumented versions of these two benchmarks were run on a Cray XC30 (Piz Daint) supercomputer. Each node has eight Intel SandyBridge CPU cores with 32 GiB memories and one NVIDIA Tesla K20X GPU. All nodes are connected with Cray's Aries network in a Dragonfly topology.

We used the *dragonegg* plugin of LLVM-3.3 to compile Fortran codes. The benchmarks are instrumented using the LLVM compiler and then the binary was generated from the bitcode using the LLVM assembler and the GCC-4.8.2 linker with '`-O3`' optimization.

For constructing the set of candidate model hypotheses from the EPMNF as described in Section 3.4, we used the following values of $j_{ik} = \left\{-1, 0, \frac{1}{3}, \frac{1}{2}, 2, \frac{3}{2}, 3\right\}$ and $l_{ik} = \{0, 1, 2\}$. Using other values for i_k and j_k only increased the model generation overhead but did not increase the prediction accuracy in our benchmarks significantly. For example, if a kernel has two parameters p_1 and p_2 in its parameter set, we would start with the following set of hypothesis functions: $(p_1 + p_2)$, $(\sqrt{p_1} + \sqrt{p_2})$, $\left(\sqrt{p_1^3} + \sqrt{p_2^3}\right)$, $\left(\sqrt{p_2^3} \log^2 p_1 + \sqrt{p_2^3} \log^2 p_2\right), \cdots$, $(\log^2 p_1 + \log^2 p_2)$. Each model hypothesis is a linear combination of the input parameters and we did not consider interaction terms (e.g., p_1/p_2, $p_1 p_3$). This set of candidate pools is not exhaustive for representation of kernels. Again if we had to consider the linear combination of all possible permutations of model hypotheses, the candidate model space for each kernel would become $21m$ where m is the number of parameters. This might cause a huge overhead in model generation in an online setting. Therefore we added one candidate kernel that is formed by the linear combination of the above 21 hypothesis. This gave us to find some interesting terms in the generated models and also took care of the overhead

4.1 Kernel Detection

Table 2 summarizes various application characteristics. It shows the number of statically identified functions and loops and the number of dynamically detected kernels. The context-sensitive kernel detection may lead to a large number of kernels if loops or functions are called from many different contexts. However, our results show that for the benchmarks considered, the maximum number of kernels discovered is 664 for the application UA from the NAS benchmarks, indicating a lesser number of different call sites for functions that contain a large number of loops. Theoretically, the number of kernels could vary depending on the input. We did not observe any such variation in our experiments.

Table 2: Number of functions ("Func"), loops, kernels ("Ker"), and parameters ("Par") for all benchmarks. Column "Zero" shows how many parameters were shrunk to zero (on average) by oLASSO and "Trace" shows the required storage in GiB for offline model generation.

Code	Func	Loops	Ker	Par	Zero	Trace
BT	28	181	211	6	1	25
CG	16	47	30	5	0	3.4
DC	79	104	178	6	1	12.45
EP	10	9	12	10	3	.75
FT	21	42	39	10	2	14.5
IS	5	16	12	6	3	1.2
LU	27	172	165	14	3	15.6
MG	24	77	98	6	2	14.5
SP	29	250	229	6	2	23.3
UA	75	473	664	8	2	28
Cloverleaf	88	634	645	16	2	25.4
CoMD	135	118	210	11	1	11
FE	546	130	610	6	4	20
MiniGhost	33	114	137	10	5	11
HPCCG	114	28	130	4	0	8

4.2 Experimental Design

We perform two experiments for each code: First, we collect the data during the execution to perform ordinary least squares (OLS) and the standard (offline) LASSO, called fLASSO in the following. In the second run, we utilize PEMOGEN to compute the parameters using oLASSO during the application run. For OLS and fLASSO, we used the `lm` [8] and `glmnet` [12] functions implemented in GNU R, respectively.

For obtaining the results of Sections 4.3, 4.4, 4.5, and 4.6, the benchmarks were run with the ten different sets of input parameter values for each of the three regression techniques to generate the performance models. Then, for calculating ARS, we ran the benchmarks with five new sets of parameter values, different from the values used in the training run. All parameters were either chosen from the different classes of NAS benchmarks or by varying the parameters in the input file of Mantevo benchmarks.

We also report the time overhead of running the three regression methods. Results show that the optimization using the state confidence machine as described is Section 3.6 reduced the profiling overhead of oLASSO as compared to the two other methods by halting the execution of kernels whose ARS became greater than user supplied 'ϵ' (we chose the value to be .85).

4.3 Goodness of fit of Performance Models

The LASSO method not only simplifies the generated performance model by shrinking some parameter coefficients to zero, but also it achieves better prediction accuracy than ordinary least square regression (OLS). The lower accuracy of OLS could be due to overfitting if there are correlations between the model parameters or some parameters do not influence the runtime. Especially the second is likely because we include parameters conservatively (cf. Section 3.2).

Figure 4 shows the ARS for all benchmarks and all three methods. We see that oLASSO has always the same or lower ARS than OLS, meaning the models generated by online LASSO have better prediction accuracy. The ARS pattern for both OLS and online LASSO agrees for the kernels in the application. The most significant difference between OLS and online LASSO can be seen in the MiniGhost and EP benchmarks indicating the existence of correlated parameters and and their elimination using LASSO. The ARS of oLASSO is generally very close to that of fLASSO with oLASSO being slightly worse in the general case.

4.4 Parameter Selection

To give an idea about the ability of oLASSO in generating interpretable models, we report the average number of model parameters whose coefficients were shrunk to 0. Table 2 shows the total number of parameters of the applications and the average number of zero-coefficient parameters (column "Zero") in the NAS and Mantevo benchmarks.

There can be various reasons why a parameter's coefficient is shrunk to zero. Either the parameter has high correlation with another parameter or the parameter is not *important* in the generation of the performance model. In the later case OLS will try to still use the parameter and suffer from overfitting. These parameters appear in the parameter set of a kernel because of our conservative approach of adding parameters to the parameter list when either the pointer analysis is unsure of a points-to relation (*may* points-to case) or there is no scope of pointer analysis (macro definitions). Comparing the results in Table 2 and Figure 4, we can see that the average number of parameters whose coefficients are shrunk to zero has a contribution towards the better prediction accuracy of online LASSO in case of benchmarks like EP, IS, FT, and MiniGhost.

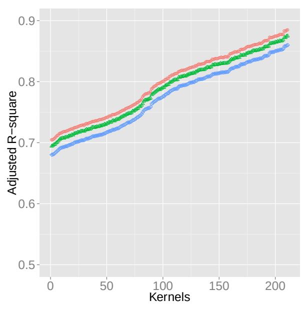
4.5 Storage Cost Elimination

A major advantage of an online model generation technique comes from its ability to eliminate the temporary storage cost that is needed to store the profile information for a number of training runs. As the model generation process continues in an incremental fashion as data arrive one at a time, there is no need to keep the data for the previous training runs and then feeding the data to a model generation technique as in the case of regular regression.

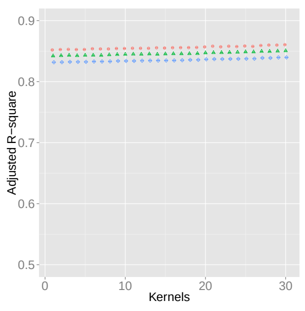
The online model generation also eliminates the cost of running the application a number of times to train the generated model with the profile data. The more training runs with different values of the model parameters, the merrier. But it is impossible to explore the whole space of possible values of the model parameters for the generation of a model with a 100% accuracy. Therefore in most cases the training runs are stopped after a desired prediction accuracy of the generated model has been reached. In other words, if the model's ARS is within an acceptable range, the training runs stopped and the application continues to execute without modeling. We chose an ARS range of .85 as an acceptable value for fitting.

In this section we report the required storage cost and the time necessary for training runs for the offline regression techniques (fLASSO and OLS) for the NAS and the Mantevo benchmarks. The storage cost is reported for the ten different sets of parameter values as were used in the previous experiments.

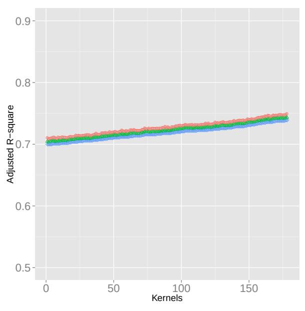
As the number of kernels grows in the program, the storage cost of the profiles also grows. For example, in Table 2,



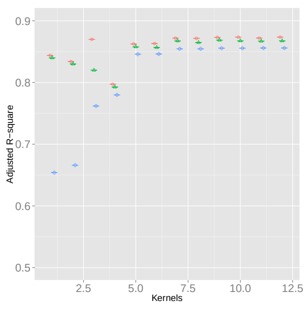
(a) BT



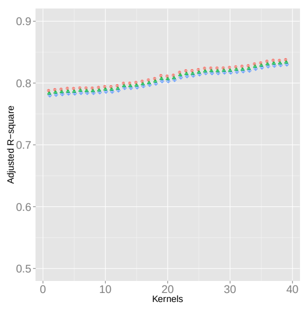
(b) CG



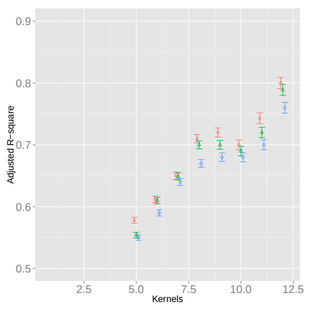
(c) DC



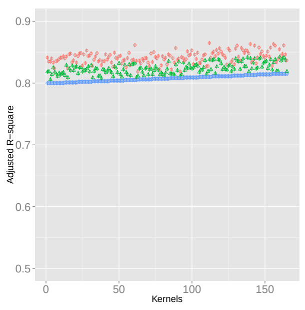
(d) EP



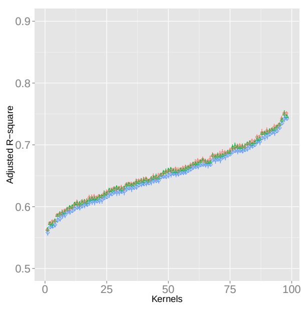
(e) FT



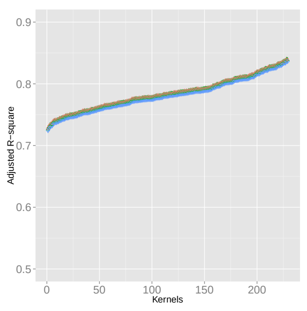
(f) IS



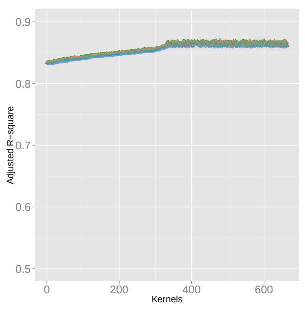
(g) LU



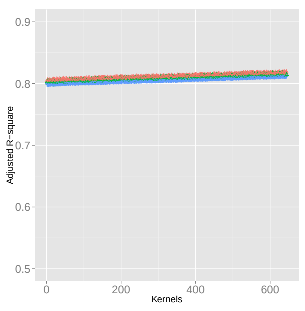
(h) MG



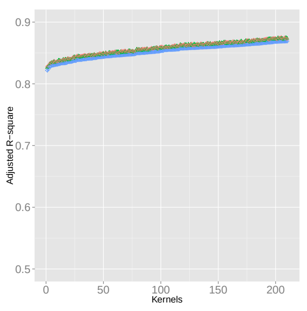
(i) SP



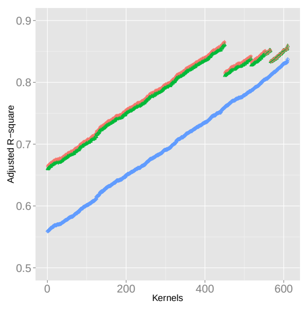
(j) UA



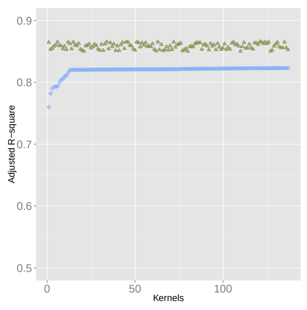
(k) Cloverleaf



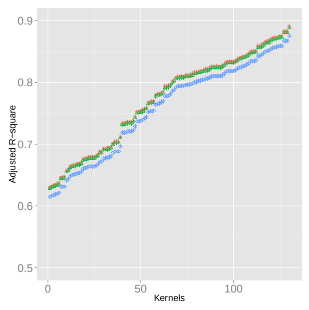
(l) CoMD



(m) FE



(n) MiniGhost



(o) HPCCG

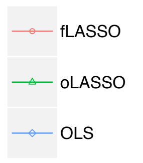


Figure 4: Comparison of predicting performance of oLASSO with OLS and Regular offline LASSO (fLASSO) in terms of their fit on new test data. The 95% confidence interval for the ARS is also included in the figure. The ARS values can range from '0' to '1' and higher is better. The *kernels* are sorted as per the ARS values of OLS along the x-axis.

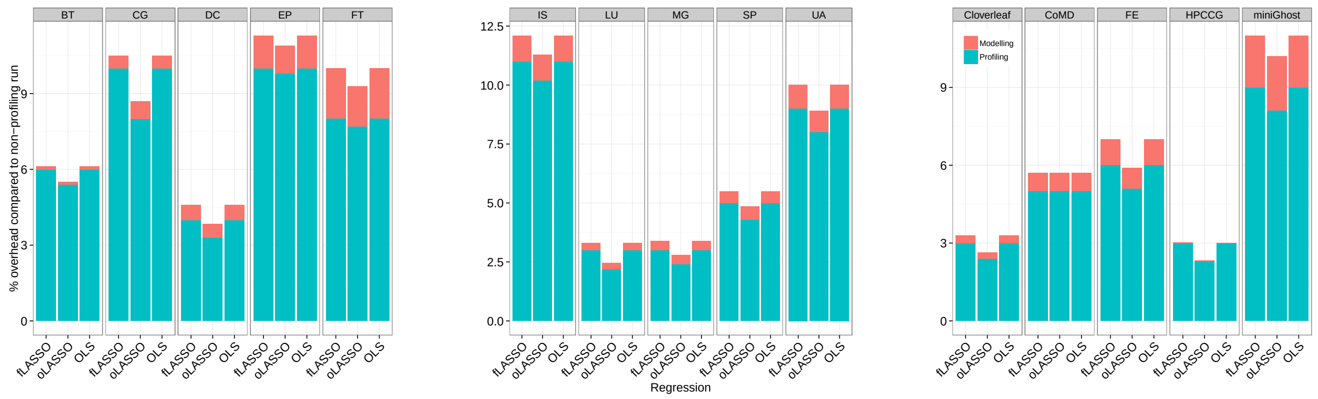


Figure 5: Time Overhead of three LASSO methods. The profiling and model generation time has been shown separately.

UA, *BT*, *Cloverleaf* benchmarks have a huge storage overhead due to the presence of the highest number of kernels. In general, all our offline experiments required between several hundreds of Megabytes and several Gigabytes storage. The oLASSO algorithm requires to store a maximum of $36 \times m$ floating point values and is therefore well suited for online model learning.

4.6 Overheads of Online and Offline Modeling

In this section, we describe the overhead of running the three different regression techniques for model generation. The overhead comes from both profiling and model generation (OLS, fLASSO, oLASSO) and model update (oLASSO). As seen in Figure 5, profiling constitutes a significant portion of the overhead in all the three techniques. The data collection overhead depends both on the number of kernels and the number of times they are executed in the program. The data collection overhead for oLASSO is less than that of the offline techniques due to the use of the confidence-state machine as described in Section 3.6. This optimization allows the overhead of oLASSO being smaller than fLASSO though the model generation overhead of fLASSO sometimes is less than the model generation and update overhead of oLASSO. The computational complexity of oLASSO is $O(d^2)$ where d is the number of active parameters (whose coefficients are non zero) at a step [29]. For the fLASSO, the computational complexity is $O(Nm)$ where N data points are present for m number of input parameters [12]. As the cardinality of the input parameter set is not very high in our case, the model generation overhead for fLASSO did not have a significant advantage over oLASSO.

5. CASE STUDIES

We now present two different case studies, one from NAS and one from the Mantevo benchmarks to illustrate our method in detail.

5.1 IS

IS is a NAS benchmark that implements the bucket sort algorithm. We randomly chose the main iteration loop of the NAS IS benchmark. IS has the following input parameters: `total_keys_log_2` (p_1), `max_key_log_2` (p_2), `num_buckets_log_2` (p_3), `max_iterations` (p_4), `test_array_size` (p_5), and `num_procs` (p_6). First for building the prediction model using oLASSO, we run the benchmarks 25 times with different random combinations of the sets parameter values as follows: p_1=(16, 20, 23, 25), p_2=(10, 16, 19, 21), p_3=(8, 10, 12), p_4=(5, 10, 15, 20), p_5=(15, 20, 22, 25, 28), p_6=(16, 20, 32, 64, 80). The runtime model generated from the 25 runs had the `max_key_log_2` parameter eliminated:

$$T = .004p_1 - 1.01p_3 + 0.04p_4 + 1.107p_5 \log p_5 + \frac{330.45}{p_6} \tag{9}$$

Using the model of Equation (9), we plotted the predictions and the actual execution times using a new set of parameter values: p_1=12, p_3=9, p_4=25, p_5=8 with changing p_6 and p_1=22, p_3=6, p_4=35, p_6=32 with changing p_5. We kept all the parameter values fixed but changed one parameter to observe the predicted and actual trends in the execution times with change in values of that one parameter. We repeat this experiment for the two paramters: number of processors and the `test_array_size`. From Figure 6, the oLASSO generated model was able to predict the performance trend with changing values of both number of processes and `test_array_size`.

5.2 HPCCG

HPCCG is a simple conjugate gradient benchmark that has 4 input parameters: `nx` (p_1), `ny` (p_2), `nz` (p_3), `num_procs` (p_4). For training the model, we ran the benchmark 25 times using random combinations from the following sets of input parameter values: p_1=(16, 32, 64, 100, 110, 120, 800, 1000), p_2=(16, 32, 64, 100, 110, 120, 800, 1000), p_3=(16, 32, 64, 100, 110, 120, 800, 1000), p_4=(8, 16, 32, 64, 128, 256). The generated model for the `ddot` kernel of the HPCCG benchmark in *weak scaling* mode was:

$$T = 0.79p_1 + 0.02p_3 + 1.11\sqrt{p_4}\log p_4 \tag{10}$$

The absence of parameter `ny` can be explained by the fact that function `ddot` works on the current row of the 3-D stencil. The points in the current row of the stencil are skipped in either `nx` or `ny` based on their values. The skipped points in the y-direction accounts for the removal of `ny` from the model. Also as the function operates on rows, the relation with `nz` is much less than that with `nx`.

We used the same method as IS to find out how well the model fits for new data using different sets of input parameter values as are used in training. The values used were p_1=80, p_2=40, p_3=55, with changing p_4 and p_2=14, p_3=125, p_4=32, with changing p_1. For plotting, we kept all the parameter values fixed but varied one parameter at a time. We chose the `nx` and `num_procs` parameters for evaluating the model predictions for the HPCCG benchmark. Figure 6 shows the results.

As can be seen from Figure 6, the model was able to successfully fit in the performance trend of the kernel.

6. RELATED WORK

Model-based Simulation is a simulation technique where not each hardware detail is investigated but abstract models are used to determine the runtime or resource consumption. The simulation time can be less than the execution time leading to a "simulation speedup" of several orders of magnitude [21, 37, 41] while still accurately capturing important details of the execution. This technique has been used to discover important performance effects [20], however, it often fails to provide the required insight to understand the root cause of such effects. It is a complex task to execute such simulations in practice such that simulation tools are often only used by computer scientists and not by application developers. Nevertheless, model-based simulation is a very accurate practical and accurate technique to predict large-scale performance.

The use of performance modeling manually has been explored before. Hoefler et al. aimed to popularize performance modeling by defining a simple six-step process to create application performance models [19]. Bauer, Gottlieb, and Hoefler show how to model performance variations using simple statistical tools [5]. They also describe how to measure the influence of certain system parameters such as the network topology.

There are approaches that focus on models generated for a very specific purpose but less on human-readable general-purpose models. For example, Ipek et al. propose multilayer artificial neural networks to learn application performance [24] and Lee et al. compare different schemes for automated machine-based performance learning and prediction [26]. Zhai, Chen, and Zheng extrapolate single-node performance to complex parallel machines [40]. Wu and Muller [39] extrapolate traces to larger process counts and can thus predict communication operations.

There is also ample research in building performance tools that can hint optimization opportunities. The authors of the Statistical Stall Breakdown [4] describe a mechanism that samples hardware counters and dynamically multiplexes hardware counters to compute a breakdown model for a PowerPC based microprocessor.

The work presented in [23] focuses on automating the process for parallel performance experimentation, analysis and problem diagnosis. Such mechanism is built on top of the PerfExplorer performance data mining system combined with the OpenUH [27] compiler infrastructure. The usage of PerfExplorer is useful for easy comparison of several experiments using the same application, and the selective instrumentation of TAU [34] helps avoid excessive overhead during the execution and gives the opportunity to provide optimization suggestions to the user.

The PerfExpert [6] tool employs the HPCToolkit [35] measurement system to execute a structured sequence of performance counter measurements to detect probable core, socket and node-level performance bottlenecks in important procedures and loops of an application.

The work described in [33] characterizes the memory behavior, including memory footprint, memory bandwidth and cache efficiency of several scientific applications. Based on the analysis of the executions of such applications they also estimate the impact of the memory system on the amount of the instruction stalls and on the real computation performance. Their results are shown per application execution, summing up all the information from the different tasks.

There are other performance tools that exploit processor hardware counters and that have integrated sampling capabilities into their analyses. Tools like TAU, Scalasca [38], HPC-Toolkit, use sampling in addition to instrumentation, their sampling capabilities are mainly focused on assigning time consumption to source code lines instead of providing finer details on the hardware counters.

Approaches exist to reduce the cost of storage and time of offline modeling. For example, the database of stored profiles are analyzed to discover parts of an application that represent the whole program execution and the performance of only that portion of the program is observed instead of the whole execution [34]. There are approaches to filter out profile data of program parts that take negligible time [15]. There are proposals of using statistical techniques (e.g. PCA, F-Ratio analysis) to reduce the dimensionality of huge collected profile data so that the analysis techniques can scale [2].

Gonzalez et al. presents a tool that automatically characterizes the different computation regions of the program [15]. They use density based clustering algorithms to performance counters to find similar code regions (that belong to the same counter). Their clustering is based on two hardware metric combination: Processor cycle combined with IPC and Completed Instructions, L1 and L2 Cache misses [15]..

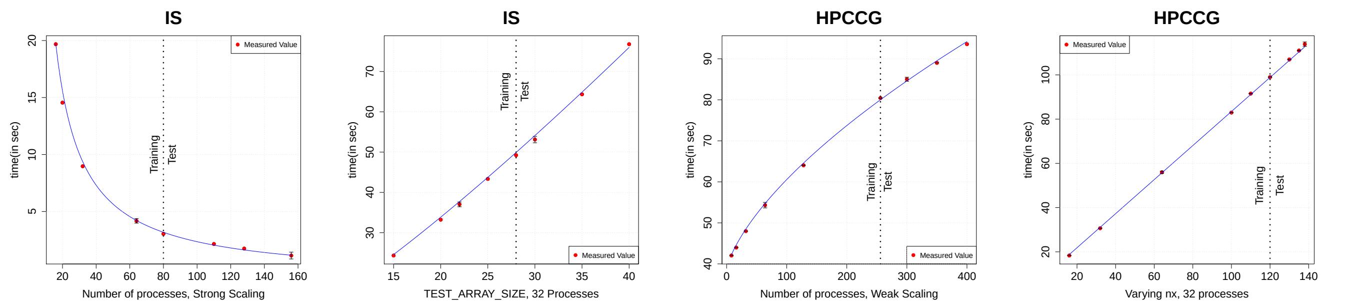
Servat et al. detects clusters based on IPC and number of instructions committed and then detects the change of performance counters like cache misses inside the clusters [28]. They modify the code manually for optimization based on the detected phase changes (slope of hardware counters w.r.t time) to get performance improvement.

But none of these above methods gave a solution to build the performance models in an online setting that would greatly reduce the storage cost. Also none of the above methods talked about the interpretability of the generated model. To our knowledge, our research is the first method of using oLASSO to generate interpretable performance models in an online fashion.

7. CONCLUSION

We demonstrated PEMOGEN, an automatic tool that can generate dynamic and adaptive, interpretable performance models using the Least Absolute Shrinkage and Selection Operator (LASSO) technique in an online fashion. PEMOGEN can be downloaded from `http://spcl.inf.ethz.ch/Research/Performance/PEMOGEN/`

We showed that the Coefficient of Variation (adjusted R^2) of online LASSO (oLASSO) is significantly better than ordinary least square regression (OLS) and very close to that of the offline LASSO (fLASSO). Based on our results, we argue



(a) Varying `num_procs` **(b)** Varying `TEST_ARRAY_SIZE` **(c)** Varying `num_procs` **(d)** Varying `nx`

Figure 6: The predicted and actual performance trends of IS and HPCCG kernels with varying the values of one parameter, while keeping the other parameter values fixed.

that using online LASSO is a better choice for the generation of performance models in HPC applications not only because the model generation and update can run transparently as the application runs and can automatically be turned on and off based on some confidence value of the model, but also the online technique greatly reduces the storage cost of training data. We also showed the effectiveness of using a confidence state machine in reducing the profiling overhead in online LASSO.

We expect that our technique to generate self-modeling applications will rapidly become state-of-the-art in high-performance programming.

8. REFERENCES

[1] Mantevo Project. `http://mantevo.org`.

[2] D. H. Ahn and J. S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *SC'02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–16, 2002.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[4] R. Azimi, M. Stumm, and R. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS'05:Proceedings of the 19th Annual International Conference on Supercomputing*, pages 101–110, 2005.

[5] G. Bauer, S. Gottlieb, and T. Hoefler. Performance modeling and comparative analysis of the MILC lattice QCD application su3 rmd. In *Proc. of CCGrid*, 2012.

[6] M. Burtscher, B.-D. Kim, J. M. J. Diamond, L. Koesterke, and J. Browne. PerfExpert: an easy-to-use performance diagnosis tool for HPC applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[7] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 45:1–45:12, 2013.

[8] J. M. Chambers. *Linear models, Chapter 4 of Statistical models in S.* Wadsworth & Brooks/Cole, 1992.

[9] Y. Dodge. *Statistical Data Analysis Based on the L1-Norm and Related Methods.* Birkhauser, 2002.

[10] D. Donoho. Compressed sensing. *Information Theory, IEEE Transactions on*, 52(4):1289–1306, April 2006.

[11] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *Annals of Statistics*, 32:407–499, 2004.

[12] J. Friedman, T. Hastie, and R. Tibshirani. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software*, 33(1):1–22, 2010.

[13] J.-J. Fuchs. On sparse representations in arbitrary redundant bases. *Information Theory, IEEE Transactions on*, 50(6):1341–1344, June 2004.

[14] P. Garrigues and L. El Ghaoui. An homotopy algorithm for the Lasso with online observations. In *Neural Information Processing Systems (NIPS)*, volume 21, 2008.

[15] J. Gonzalez, J. Gimenez, and J. Labarta. Automatic detection of parallel applications computation phases. In *IPDPS*, pages 1–11, 2009.

[16] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, 1982.

[17] I. Guyon and A. Elisseeff. An Introduction to Variable and Feature Selection. *J. Mach. Learn. Res.*, 3:1157–1182, Mar. 2003.

[18] T. Hoefler. Bridging Performance Analysis Tools and Analytic Performance Modeling for HPC. In *Proceedings of Workshop on Productivity and Performance (PROPER 2010).* Springer, Dec. 2010.

[19] T. Hoefler, W. Gropp, W. Kramer, and M. Snir. Performance Modeling for Systematic Performance Tuning. SC '11, pages 6:1–6:12, 2011.

[20] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010.

[21] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 597–604. ACM, Jun. 2010.

[22] A. E. Hoerl and R. W. Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 12(1):55–67, 1970.

[23] K. Huck, O. Hernandez, V. Bui, S. Chandrasekaran, B. Chapman, A. Malony, L. McInnes, and B. Norris. Capturing performance knowledge for automated analysis. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC'08*, pages 49:1–49:10, 2008.

[24] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Proc. of the 11th Intl. Euro-Par Conference*, pages 196–205, 2005.

[25] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[26] 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 *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPoPP'07)*, pages 249–258, 2007.

[27] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: an optimizing, portable OpenMP compiler. In *12th Workshop on Compilers for Parallel Computers*, 2006.

[28] G. Llort, J. Gonzalez, H. Servat, J. Gimenez, and J. Labarta. On-line detection of large-scale parallel application's structure. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, 2010.

[29] D. Malioutov, M. Cetin, and A. Willsky. Homotopy continuation for sparse signal representation. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*, volume 5, pages v/733–v/736 Vol. 5, March 2005.

[30] NASA. NAS Parallel Benchmarks. `https://www.nas.nasa.gov/publications/npb.html`.

[31] A. Y. Ng. Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 78–, 2004.

[32] M. R. Osborne. An effective method for computing regression quantiles. *IMA Journal of Numerical Analysis*, 12(2), 1992.

[33] M. Pavlovic, Y. Etsion, and A. Ramirez. Analysis of memory system requirements for scientific computing. In *IEEE International Symposium on Workload Characterization*, 2009.

[34] S. Shende and A. Malony. The TAU parallel performance system. *International Journal of High Performance Computer Applications*, 20:287–311, 2006.

[35] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, and M. Krentel. HPCToolkit: performance tools for scientific computing. *Journal of Physics: Conference Series*, 012088, 2008.

[36] R. Tibshirani. Regression Shrinkage and Selection Via the Lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

[37] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely. PSINS: An open source event tracer and execution simulator. In *DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2009*, pages 444–449, June 2009.

[38] F. Wolf, B. Wylie, E. Abraham, D. Becker, W. Frings, K. Furlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi. Usage of the SCALASCA for scalable performance analysis of large-scale parallel applications. In *Tools for High Performance Computing*, pages 157–167, 2008.

[39] X. Wu and F. Muller. Scalaextrap: Trace-based communication extrapolation for SPMD programs. *ACM Transactions on Programming Languages and Systems*, 34(1), 2012.

[40] J. Zhai, W. Chen, and W. Zheng. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. *SIGPLAN Notices*, 45(5):305–314, 2010.

[41] G. Zheng, G. Gupta, E. Bohm, I. Dooley, and L. Kale. Simulating Large Scale Parallel Applications Using Statistical Models for Sequential Execution Blocks. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 221–228, Dec 2010.