Using Compiler Techniques to Improve Automatic Performance Modeling

Arnamoy Bhattacharyya, Grzegorz Kwasniewski, Torsten Hoefler
Department of Computer Science
ETH Zurich
Zurich, Switzerland
Email: arnamoyb, grzegorz.kwasniewski, htor@inf.ethz.ch

***Abstract*—Performance modeling can be utilized in a number of scenarios, starting from finding performance bugs to the scalability study of applications. Existing dynamic and static approaches for automating the generation of performance models have limitations for precision and overhead. In this work, we explore combination of a number of static and dynamic analyses for life-long performance modeling and investigate accuracy, reduction of the model search space, and performance improvements over previous approaches on a wide range of parallel benchmarks. We develop static and dynamic schemes such as kernel clustering, batched model updates and regulation of modeling frequency for reducing the cost of measurements, model generation, and updates. Our hybrid approach, on average can improve the accuracy of the performance models by 4.3%(maximum 10%) and can reduce the overhead by 25% (maximum 65%) as compared to previous approaches.**

***Keywords*-Performance Modeling, LASSO Regression, Static Analysis**

I. Introduction

Performance modeling is useful in a number of ways – ranging from finding scalability issues in scientific applications [1] to finding performance bugs [2]. Analytical performance modeling is difficult and requires detailed understanding of the application code [3]. Therefore an automated approach to generate performance models eliminates the necessity of having detailed domain knowledge and helps performance engineers to quickly determine the behaviour of an application.

The recent automation of various parts of the performance modeling process [4], [5], [1] led to a wider adoption of this methodology. A pure dynamic or a pure static approach to generate performance models is not sufficient as each of them suffers from the lack of precision of the generated models. Again, a dynamic model generation strategy suffers from overhead that may not be acceptable during a production run. In this work, we combine a number of existing and new approaches to tackle the life-long automatic model generation problem. Our goal is to generate performance models at high precision and low overhead so that there is minimum perturbation during the program execution.

Figure 1 depicts existing and new approaches and how we combine them together for the generation of performance models. We combine results from statically determining the iteration counts of program loops [6] with existing dynamic but limited performance modeling techniques [5]. Just as in model-checking, the static analysis over-approximates the state space of the program and may not always find accurate equations. However, we show how any information gained via static analysis can be used efficiently to improve the performance as well as the accuracy of existing dynamic techniques by reducing the model search space significantly. For example, in Figure 1, the automated loop iteration counting technique helps to find interaction terms between program inputs that will otherwise be very expensive to search in a dynamic setting. Static data flow analysis helps to map the *definition* of program inputs to the various *uses* inside loops and functions and therefore reduces the search space for automatic model generation for a given loop or function.

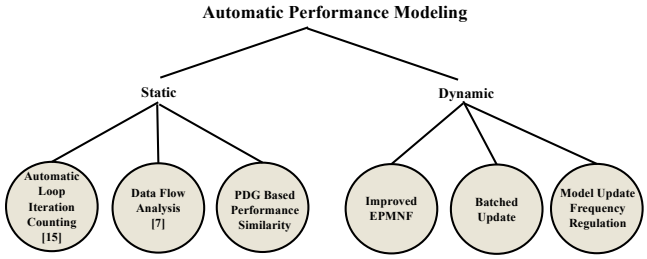


Figure 1: Combining Existing and New Approaches for Automated Performance Modeling. Our novel techniques are the citation-free nodes.

In addition to analyzing the effectiveness of combining the static and dynamic model generation, we build a tool that automatically generates self-modeling applications. As shown in Figure 1, to achieve lowest overhead, we develop several static and dynamic techniques, such as static Program Dependence Graph (PDG) [7] based *kernel* similarity analysis, batched model update and adaptive measurements. We also extend prior work on dynamic model generation (EPMNF) [5] by introducing interaction terms in models that greatly improve precision. Self-modeling at low overhead enables life-long performance modeling, where the

application automatically learns and observes performance of its parts during production (for various sets of input parameters used in practice). It can then either output the learned models or watch for outliers in the measurements and raise performance exceptions early.

A. Contribution

We make the following contributions in this paper:

- We explore the combination of static and dynamic analysis and investigate the accuracy and the reduction of the model search space for a wide range of benchmarks.
- We develop a static scheme to cluster kernels based on the similarity of their performance models. This reduces the overhead during life-long performance modeling.
- We design and analyze various dynamic strategies, such as batched model updates and regulation of measurement frequency for reducing the cost of measurements, model generation, and updates.

II. Limitation of a Pure Static or Dynamic Approach

Though performance modeling can be essential to understand performance limits, automation of such techniques only started recently. In these automatic approaches the performance M of a program is represented through the performance models m of n program parts, called *kernels*:

$$M = \{m_1, m_2, \ldots, m_n\} \tag{1}$$

We define the performance model m of each kernel as a linear regression function of a set of *predictors* $p = \{p_1, p_2, \ldots, p_p\}$.

$$m = \sum_{i=1}^{|p|} \alpha_i \cdot p_i + \beta \text{ where } p_i \in p \tag{2}$$

A predictor p_i is a function of one or more program input parameters ι. If there are r input parameters that influence the performance of a *kernel*, the predictor set is formed by applying a set of transformations $\tau_1, \tau_2, \ldots \tau_v$ on those input parameters.

$$p = \{\bigcup_{v} \bigcup_{r} \tau_v(\iota_r)\} \tag{3}$$

It can be worth noticing that an infinite number of transformations can be applied to each input parameter, thus giving rise to an infinite set of *predictors*. Researchers pursued two basic avenues to bind this infinite space of *predictors* so far: (1) static modeling by analyzing the source code [8] and (2) dynamic modeling during program runtime [9] . Still both methods have serious limitations which we briefly describe in the following.

A. Static Performance Modeling

Several static analysis techniques exist for analyzing the performance of programs. Since scientific programs spend a major portion of the execution time in loops, most of the static analysis methods count the number of loop iterations to bound the asymptotic performance of loops. Polyhedral methods can handle loop nests whose iteration space (all possible values of the iteration variables) form affine sets [10]. Hoefler and Kwasniewski [6] show an analysis that handles a larger class of loops whose iterators are manipulated with affine functions. For example, consider the following loop[1]:

Listing 1: An Affine Loop

```
for (j = 1; j < niter; j = j*2)
 for (k = j; k < niter; k = k++)
  A[k,j] = (A[k-1,j-1] + A[k-1,j]) / 2
```

Using the polyhedral approach, the loop is modeled as the function[1] $\frac{niter(niter-1)}{2}$, which is an over-approximation. In the HK method, a program is represented as a set of nested loops - similar to the polyhedral model. It was shown in previous works that a subset of this class covers many important codes in parallel computing [8]. The HK method works with the following terms derived automatically from the outer loop nest of Listing 1:

1) Loop Induction variable: `j`
2) Initial assignment: `j=1`
3) Loop guard: `j<niter`
4) Loop update: `j=j*2`

If all loops are affine, using the above information, the HK method forms a set of equations that can be symbolically solved to get the performance model of the loop, which is a linear combination of a set of *predictors*.

For example, the exact and more precise model m of the loop above as given by the HK method is

$$m = niter \cdot \lfloor \log(niter) \rfloor - 2^{\lfloor \log(niter)+1 \rfloor} + niter + 1 \tag{4}$$

However, the HK method cannot handle non-affine loops. The performance model of such loops are marked as undefined ($undef$). Therefore, the $undef$ terms in the models make this method limited because the method cannot give precise analysis for non-affine loops. The following non-affine conditions may occur in the program:

- Multiple exits from a loop body
- Loop update is not an affine function
- Loop guard or initial assignment contain references to non-constant values (function calls, variables declared and changed outside of the loop tree, untraceable calls)

For example let us consider the following loop from the CG benchmark of the NAS benchmark suite [11]:

[1]Update (24.09.15): The listing was adapted to match the original HK method which uses strict inequalities.

Listing 2: A Non-affine Loop

```
do j=1,lastrow-firstrow+1 sum = 0.d0
 do k=rowstr(j),rowstr(j+1)-1
  sum = sum + a(k)*p(colidx(k))
 enddo
 w(j) = sum
enddo
```

The values of `rowstr(j)` and `rowstr(j+1)` cannot be determined statically and the inner loop of Listing 2 becomes non-affine. The HK method can trace back the value of `lastrow-firstrow` to the input parameter combination `na/nprows` therefore the model of the outer loop becomes:

$$m = \frac{\texttt{na}}{\texttt{nprows}} \cdot \textit{undef} \tag{5}$$

Therefore, though the HK method is useful to discover interaction between *predictors* to generate better quality models, the presence of *undef* terms in the models necessitates an extension of this method. We describe in Section III how we can still use the precise results from HK method to help the dynamic model generation as well as use runtime profiling information to resolve most *undef* terms obtained from the HK method.

B. Dynamic Performance Modeling

Recently dynamic approaches in performance modeling have been proposed [1], [5]. In these approaches, the *predictors* are formed by applying powers and logarithm transformations on program inputs. The search space of *predictors* is constructed from program input parameters using the following form which is called Extended Performance Model Normal Form (EPMNF):

$$p = \{\iota_i^k \log^l \iota_i^k,\ k, l \in \mathbb{R},\ \iota_i \in I\} \tag{6}$$

Here I represents the set of program input parameters. By assigning different values to k and l, the predictor set is constructed from the input parameters. An example model from EPMNF for program input parameters ι_1 and ι_2 would be $c_1 \cdot \iota_1^2 + c_2 \cdot \iota_2 \log \iota_2$, where c_1 and c_2 are constants. The formulation of EPMNF uses an educated guess about the performance of the applications studied. The method neither uses any static analysis to understand the behavior of the kernels nor does it consider the interaction of predictors. For example predictors such as $\iota_1 \cdot \iota_2$ (for two different input parameters ι_1 and ι_2) will not be considered using the EPMNF method. If we extend EPMNF to include predictors formed using all possible combinations of the program inputs, the search space of predictors will explode combinatorially as $\binom{ikl}{w}$, where w *predictors* are taken at a time for combination.

We show in Section III how we can extend EPMNF to consider combination of *predictors* to generate higher precision performance models, still at a relatively low overhead.

III. The new Hybrid Approach

To overcome the weakness of the previously described static and dynamic approaches, in this section we describe our new approach for generating performance models with higher accuracy. We also describe techniques for reducing the overhead of the model generation because our aim is to generate life-long performance models in a dynamic settings.

A. Modeling Granularity and Kernels

The performance of an application can be viewed as a single model for the whole source code, which is the sum or product of all its parts, but this approach has two drawbacks:

- **Accuracy:** The used methods could become less precise for complex models, for example, a whole program analysis may detect a loop that executes a logarithmic number of times and calls a matrix multiplication, however, it may not detect the $n^3 \log n$ model of the composition easily due to complex flow patterns.
- **Resolution:** Users are often interested in the finest possible resolution in order to spot potential performance problems, for example, it is less helpful to report that a program execution of a complex program takes cubic time than pointing to the set of most expensive functions in the program.

Without loss of generality, we consider functions and loops in a program as *kernels*. We use the Loop Call Graph (LCG) [5] representation of the program to identify kernels based on loop nesting and loops within functions.

B. Performance Model Parameters

The performance of each *kernel* depends on the values of a set of program input parameters. We require the programmer to specify a set of scalar parameters that influences the program runtime. The programmer specifies the names of the source code variables that influence the runtime and the compiler traces back the variables in the source from IR using debug information. Currently, our implementation supports the following three types of program inputs:

- **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.

In the next section we describe how static and dynamic analyses can help each other for the generation of performance models. Our performance model generation technique works by selecting, from an *initial* search space of predictors, the predictors that have the most significance to describe the performance of a *kernel*. This is done in two steps:

1) Construct the *initial* search space of predictors using: (i) the result from a precise static analysis (when there are no *undef* terms in the reported model) or (ii) using our improved definition of EPMNF in cases where the result from static analysis has *undef* terms.
2) Using runtime profiling information, run an online variant of LASSO regression [5] to select the most significant *predictors* and generate the performance model.

C. Accuracy improvement and Search Space Reduction

Our new approach discovers new *predictors* that are combinations of two or more of them. We follow two approaches to discover the interaction terms:

1) Result From Static Analysis: Once the models are generated statically by analyzing the intermediate representation (IR) of the compiler, the models are checked for preciseness. If there are no *undef* terms in the static model, the model is parsed as a linear function of predictors and that set of predictors becomes the *initial* search space of predictors for that particular kernel.

2) Use Runtime Information: For static models with *undef* terms, we first build the *initial* predictor set of that kernel using static pointer analysis (to find which input parameters are used in which kernel) and Equation 6.

The pointer analysis is performed between the operands of the set of memory access (load/store) instructions inside a kernel and the set of first store instructions of the program input parameters. If there is a *must* points-to relation between an operand in a memory access instruction inside a kernel and the store to a program input parameter p_1, predictors are formed from p_1 using EPMNF. We also conservatively add an input parameter that has a *may* points-to relation with any memory access inside the kernel. If the pointer analysis confirms that for an input parameter p_2, no memory access operand inside a kernel points to p_2, we do not use p_2 for that kernel.

The above method handles variables that are related by a points-to relation to a program input parameter. But there can be *derived* variables that are related with program input parameter(s) by arithmetic operation(s). The next code-extract from the NAS Parallel Benchmark `IS` shows how variables are created using program input parameters and therefore can be traced back to program inputs.

Listing 3: Example: Variable Propagation

```
for( i=0; i<NUM_KEYS; i++ )
  bucket_size[key_array[i] >> shift]++;
```

In the source code, `NUM_KEYS` is derived from the program input parameters `TOTAL_KEYS_LOG_2` and `NUM_PROCS`. We traverse the data flow graph backwards in the IR and use the debug information included in the IR to find the relation of variables to the program input parameters. The traced back input parameters are included in the static model (during static analysis using HK method) and passed to EPMNF (for dynamic analysis).

We not only construct the *initial* search space of *predictors* formed using the EPMNF definition [5] in Equation 6, but also consider interactions among them. Thus we redefine Equation 6 for a calculating the predictor set in the hybrid approach as:

$$p_{hybrid} = \{\iota_i^k \log^l \iota_i^k \cup C_w(\iota_i^k \log^l \iota_i^k)\},\ \{\iota_i, \iota_2, \ldots, \iota_r\} \in I \tag{7}$$

Here C_w represents the interaction terms constructed from predictors formed using the EPMNF definition, taken w at a time. If from EPMNF we have κ *predictors* initially, we construct new *predictors* by taking all different combinations of *predictors* in groups of size w where $2 \leq w \leq \kappa$. For example, if EPMNF gives us three *predictors* $\iota_1^2, \log \iota_2, \frac{1}{\iota_3}$, we will construct the following new *predictors* in groups of size 2 and 3: $\iota_1^2 \cdot \log \iota_2, \frac{\log \iota_2}{\iota_3}, \frac{\iota_1^2}{\iota_3}, \frac{\iota_1^2 \cdot \log \iota_2}{\iota_3}$. We have seen from our experiments that the predictive quality of the models using the values $w = 2, 3, 4$ greatly improves precision of models and a value of $w > 4$ marginally improves the model quality (less than 0.01%). Though previous research showed that $w = 2, 3$ combination of *predictors* generate good quality models [12], but according to our experience, values of $w = 2, 3, 4$ improve quality of the models than taking only values $w = 2, 3$.

After the *initial* search space of *predictors* is formed using the result from static analysis and our improved definition of EPMNF, performance models are generated during runtime using an online version of the LASSO regression algorithm [5]. LASSO removes insignificant *predictors* and generates easily interpretable model.

D. Overhead Reduction

Model generation during a program run suffers from overhead arising from different actions, e.g., (i) gathering profiling data, (ii) measuring model accuracy to take a decision to whether to update the model and then (iii) updating the model. The overhead should be taken into account if we want to enable life-long performance modeling. Overhead also depends on the cardinality of *initial* search space of *predictors* that are passed to LASSO [13]. This cardinality is already tackled while using the HK method and restricting the values of w as described in the previous section.

In this section we describe various strategies to further reduce overhead.

1) Batched model update: Updating an existing model every time new measurement arrives can cause large overheads in loops that are executed many times during a program run. For example the following code from the NAS benchmark `BT`, fills a buffer that is to be sent to eastern neighbors:

Listing 4: Example large loop from BT

```
if (cell_coord(1,c) .ne. ncells) then
 do k = 0, cell_size(3,c)-1
  do j = 0, cell_size(2,c)-1
    do i = cell_size(1,c)-2,cell_size(1,c)-1
      do m = 1, 5
         out_buffer(ss(0)+p0) = u(m,i,j,k,c)
         p0 = p0 + 1
       end do
     end do
   end do
 end do
```

For a `B` class BT with 16 processes, the second loop in the nest is executed 6 million times, therefore giving rise to 6 million different executions of the third loop in the nest. If the model update function is called each time the third loop finishes execution (each time a new measurement arrives for modeling), the overhead resulted only from the third loop is 1.7%. There are more similarly large loops in BT and calling the model generation function on each new measurement causes a combined overhead as high as 23%.

To have a balance between the amount of collected profile data and model update frequency, we call the model generating function on batches of data points instead of calling the function every time a new measurement arrives. Still this method has much less storage overhead because mainly the storage overhead in applications come from *kernels* that run a large number of times.

We further regulate the model update frequency based on the prediction of the already generated models on unseen data. We describe this strategy in detail in Section III-D3.

2) Exploiting Static Model Similarities: Often scientific applications have "similar looking" loops. The performance models for these loops have the same functional form, but may differ in the *predictors* in them. For example, the two loop-nests in Listings 5 and 6 occur in the `accelerate_kernel_c_` function[2] from the Mantevo benchmark [14]. Both of them assign values to the same number of locations of the two different arrays `xvel1` and `yvel1`. While calculating the value to be assigned, both loops access the same number of locations from different arrays and apply the same operations on them. Static analysis can be used to identify such *similar* loops and cluster them. Then our tool generates model for only one loop per cluster, thus reducing overhead. We call this similarity *static* similarity.

We use Program Dependence Graph (PDF) [7] for the detection of static similar loops. A program dependence graph is a graph G where the set of vertices V are either statements or predicates and the set of edges E are either control flow dependences or data flow dependences between the nodes. For detecting *static* similar loops, we use a similar technique as used in clone detection [15], with two differences. First, we do not consider functions because the static similarity among loops occurs much more often than static similarity among functions in scientific codes. Therefore, we restrict our analysis to loops. Secondly, while comparing two nodes in the PDGs of two loops to find a *syntactic equivalence* (defined in the next paragraph), we ignore variable names in the statement or predicate. Therefore we can cluster loops whose performance models have the same functional form but only differ in the *predictors*.

Our analysis starts by choosing a pair of loops in the program. Our algorithm first finds two nodes, one each from the respective PDGs of the two loops, that have the same syntactic structure (they have the same instructions at the same location of their respective basic blocks and the number of arguments and the *type* of arguments for the instructions also are the same). Once one such pair of nodes is found, we traverse the PDGs of both loops to check the syntactic equivalence of the successor nodes. Our analysis searches for equivalence in both control flow and data flow successors and therefore can find loops that are semantically similar (loops that perform in a similar way). The algorithm succeeds when each node in a PDG of a loop has a matching node in the PDG of the other loop. The algorithm aborts as soon as the successor nodes for the respective PDGs are not syntactically equivalent. The algorithm thus clusters all the loops in the program into static similar clusters.

This similarity information leads to the reduction of number of loops needed to be analyzed dynamically, as only one 'base' loop per similarity cluster needs be profiled. Though during run time the performance of static similar loops may vary due to caching activity. To tackle that problem, we introduce a fall-back mechanism based on the prediction confidence of the models in the next section.

3) Frequency of Model Update: To reduce overhead of runtime model generation, we regulate the frequency of model update based on the prediction of the model on unseen data. This regulation not only allows the reduction of model update frequency for good quality models but also allows us a fall-back mechanism for the kernels that are found to be static similar but at run time, for some reason (e.g. caching activity) behaves *dissimilar*ly. We use an exponential-backoff strategy similar to the one used in congestion control of a network for models with a good prediction performance on unseen data. We call a model whose prediction on unseen data batch falls under a certain confidence interval (95%) of previous predictions, a *strong* model. The metric we use for measuring this prediction confidence is described in the next section. Once a model reaches the *strong* state, both the frequency of profiling and update of the model is regulated. We keep a 'hit' counter every time the prediction on new data (batch) falls under 95% confidence interval of previous predictions. With a hit, the profiling and model update for a loop is delayed for the next q number of iterations where q is given by:

$$q = b \cdot rand(0, 2^{h_c}) \tag{8}$$

[2]Update (24.09.15): The function is taken from the CloverLeaf miniapp.

Listing 5: Loop 1 (static similarity)

```
for (k=y_min;k<=y_max+1;k++) {
 for (j=x_min;j<=x_max+1;j++) {
  xvel1[FTNREF2D(j,k,x_max+5,x_min-2,y_min-2)]
  =xvel0[FTNREF2D(j,k,x_max+5,x_min-2,y_min-2)]
  -stepbymass[FTNREF2D(j,k,x_max+5,x_min-2,y_min-2)]
  *(xarea[FTNREF2D(j,k,x_max+5,x_min-2,y_min-2)]
  *(pressure[FTNREF2D(j,k,x_max+4,x_min-2,y_min-2)]
  -pressure[FTNREF2D(j-1,k,x_max+4,x_min-2,y_min-2)])
  +xarea[FTNREF2D(j,k-1,x_max+5,x_min-2,y_min-2)]
  *(pressure[FTNREF2D(j,k-1,x_max+4,x_min-2,y_min-2)]
  -pressure[FTNREF2D(j-1,k-1,x_max+4,x_min-2,y_min-2)]
  ));
 }
}
```

Listing 6: Loop 2 (static similarity)

```
for (k=y_min;k<=y_max+1;k++) {
 for (j=x_min;j<=x_max+1;j++) {
  yvel1[FTNREF2D(j,k,x_max+5,x_min-2,y_min-2)]
  =yvel0[FTNREF2D(j,k,x_max+5,x_min-2,y_min-2)]
  -stepbymass[FTNREF2D(j,k,x_max+5,x_min-2,y_min-2)]
  *(yarea[FTNREF2D(j,k,x_max+4,x_min-2,y_min-2)]
  *(pressure[FTNREF2D(j,k,x_max+4,x_min-2,y_min-2)]
  -pressure[FTNREF2D(j,k-1,x_max+4,x_min-2,y_min-2)])
  +yarea[FTNREF2D(j-1,k,x_max+4,x_min-2,y_min-2)]
  *(pressure[FTNREF2D(j-1,k,x_max+4,x_min-2,y_min-2)]
  -pressure[FTNREF2D(j-1,k-1,x_max+4,x_min-2,y_min-2)]
  ));
 }
}
```

Here, b is the batch size and h_c is the current hit counter value. For a kernel that is a function, the delay is simply the random number between 0 and 2^{h_c}. We chose to generate a random number because that gives us a chance to detect sudden fluctuations in the behavior of a kernel. If the back-off counter is monotonically increasing, the probability of detecting this sudden fluctuation becomes low. When a miss occurs, the back-off counter value is set to 0. If the number of iterations of a loop does not reach the delay size q at the end of the execution, the model for the loop is updated.

E. Measuring Model Accuracy

Though our modeling methodology can be generalized for a number of performance metrics, we model execution time because then the models can be used to find both performance bugs and scalability issues. In a dynamic setting, a typical run creates a dynamic profile *prof* for v kernels in an application. A dynamic profile of a *kernel* consists of a series of *measurements* mr_i's during that run:

$$prof = \{mr_1, mr_2, \ldots, mr_v\} \tag{9}$$

The measurement mr_i of a kernel i is a tuple consisting of the measured execution time (in regression terms, *response*) for one execution of the kernel and the dynamic values of its predictor set:

$$mr_i =< t_i, p_i > \tag{10}$$

For loops, the measurements are done for calculating the fitting of the model. The model is updated for a batch of b measurements at once instead of calling the model update function each time a new measurement is received. The next b measurements are used as *test* data to evaluate the model accuracy. We use adjusted R-square (ARS) of the predictions to determine the accuracy. But as we are calculating ARS out-of-sample, we call this metric Predicted Adjusted R-Square (PARS). The formula for PARS is the same as ARS and is given below [16]:

$$\mathrm{R}^2 = 1 - \frac{\sum_{i=1}^{b} \left(t_i - pred(p_i)\right)^2}{\sum_{i=1}^{b} \left(t_i - \bar{t}\right)^2} \tag{11}$$

$$\mathrm{PARS} = \mathrm{R}^2 - (1 - \mathrm{R}^2)\frac{m}{b - m - 1} \tag{12}$$

Where b and m are the test data batch size and number of *predictors* respectively. $pred(p_i)$ represents the predicted value of the response based on the generated model on the *predictors*. $\bar{t}$ is the average of measured execution times in a batch. We use PARS because widely used metrics like R^2 can be misleading in an online setting. R^2 gives the fit within training data, but not on unseen test data. As we are using the online method for the lifetime of an application, the number of *training* runs is unlimited in our case and therefore R^2 is not a good metric for us. If the batch size is not met at the end of a program, the model update function is called on the remaining data points. The effectiveness of this batch update in reducing overhead is described in Section IV-C where we tune the batch size b based on a trade-off between memory requirement and model update overhead.

PARS is not enough to determine whether the assumption about the linearity of the model holds. A high value of PARS can be deceiving because it does not consider the variation of fitted values from the mean of observations. We use a second test, the Lack of Fit (LOF) F-test [17] to determine if there is a significance lack-of-fit of the linear fitted model on new data.

The F-value for the test is calculated using the formula:

$$\mathrm{F} = \frac{\sum_{i=1}^{h} \left(\bar{t} - pred(p_i)\right)^2 / (c - 2)}{\sum_{i=1}^{h} \left(t_i - \bar{t}\right)^2 / (h - c)} \tag{13}$$

Here h is the number of data points and c is the LOF degrees of freedom and is determined by the number of distinct values of the *predictors*. F-statistics needs multiple observations for a fixed set of predictor values so that the term $h - c$ does not become 0. Therefore we need to keep the previously seen input parameter configurations. For measuring F-statistics on a batch of b observations, $h = b + w$, where w is the number of seen input parameter values. The storage space and retrieval of seen input configurations are optimized using hash tables. The p-value for an F-test is then calculated using the incomplete beta function (I) [18] of the F-value and the

two degrees of freedom according to the formula [18]:

$$p_{F-test} = 1 - I_{\frac{(c-2)\mathrm{F}}{(c-2)\mathrm{F}+(h-c)}}((c-2)/2, (h-c)/2) \quad (14)$$

A p-value less than 0.05 can be interpreted as a significant lack of fit.

IV. Experiments

A. Accuracy Improvement

We run all our experiments on a Intel core-i7 3.4 GHz quad-core machine where each core is 2-way multi-threaded. We extend an existing online modeling tool PEMOGEN [5] with our new techniques for accuracy improvement and overhead reduction. We generate performance models for a number of scientific applications from the NAS [11] and MILC [19] benchmark suites. We use the DragonEgg plugin of LLVM-3.3 to compile Fortran codes. The benchmarks are instrumented using the LLVM compiler and then the binary is generated from the bitcode using the LLVM assembler and the GCC-4.7 linker with '-O3' optimization. MPICH version 3.1.2 is used for benchmarks that use MPI.

We vary the input parameters according to the different classes of the NAS benchmarks (different problem sizes) and according to the input provided with the MILC benchmarks including various grid configurations and values such as preferred accuracy, seed for random number generator etc. For the generation of models, we use 20 *training* runs including replicated measurements with a fixed set of input parameter values. We also run 20 *test* runs with a mixture of seen (in the *training* runs) and unseen input parameter values (with replications) to measure the accuracy of the generated models. The replicated runs with a fixed set of input parameter values are necessary to calculate the LOF of the models (see Section III-E). In the experiments we compare the accuracy improvement and overhead reduction of our hybrid approach over the previous dynamic approach [5].

Figure 2 reports the distribution of percentage improvement of PARS value (accuracy) for the generated models that have insignificant LOF from the new hybrid approach as compared to the previous dynamic method. As can be seen in the figure, the hybrid strategy significantly improves the PARS for the generated models compared to the previous dynamic approach.

Figure 2 also shows the distribution of PARS and p-values of LOF for the generated models for one benchmark from each of the two benchmark suites. We do not include figures for all the benchmarks due to space contraints but we observed from our experiments that for benchmarks such as `BT`, `LU`, `SP` the models for a number of kernels suffer from LOF for the previous online technique due to presence of interaction terms of *predictors*. Using static analysis, those interaction terms can be discovered and therefore the p-value for the models of a number of kernels become insignificant.

For the MILC benchmark `gp_quark_prop`, the performance of the previous dynamic technique is poor because a large portion of the kernels suffer from significant LOF. This indicates that models formed using the old EPMNF definition are not suitable for these benchmarks and the statically discovered *predictors* that include interaction of *predictors* should be considered for better fitting models.

It is also interesting to see from Figure 2 that in `BT` and `kid_su3_rmd`, a number of kernels have a high PARS value (> 0.80) but a significant LOF. However, the PARS of the models using the hybrid approach are higher than the maximum achievable using the previous dynamic technique.

Even after using the hybrid approach, there are a few kernels that suffer from significant LOF (p-value < 0.05). This indicates that these kernels are not accurately modeled even using the hybrid approach. They are mostly complex functions of input parameters that are not easy to model automatically.

B. Overhead Reduction

The hybrid approach reduces the overhead by reducing the number of *predictors* that is passed to LASSO (see Section III). The second column of Table I shows the average number of parameters reduced by applying the hybrid strategy — the highest number of parameters reduced is for `FT` from NAS suite, accounting for the 3.6% overhead reduction by the hybrid approach over the previous dynamic approach.

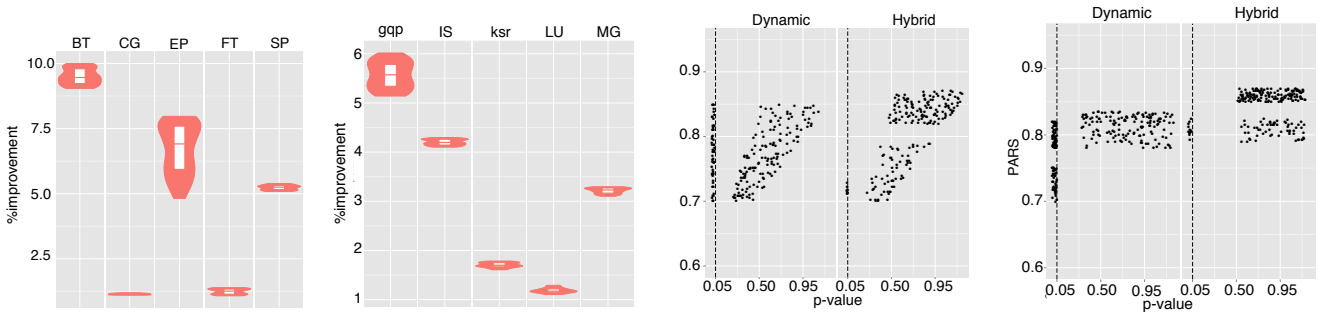
We realize that the overhead is still significant for life-long performance modeling. Now we give the efficacy of the previously described overhead reduction techniques.

C. Batched Model Update

We perform experiments with different batch sizes to find a suitable size for modeling in the benchmarks. Table II shows the different memory and timing overhead for different batch sizes. The maximum timing overhead is huge if the update function is called every time a new data point arrives. Also the improvement beyond a batch size of 2048 is insignificant due to the absence of enough iterations for the overhead to further decrease in the loops of the benchmarks. We choose a batch size of 512 for keeping the overhead at a maximum of 9.11% with respect to a run without profiling and model generation.

D. Exploiting Static Model Similarities

We perform experiments to find the number of *static* similarity clusters formed and how accurately we can predict the performance of other loops in a similarity cluster by using the hybrid model of the base loop in that cluster. For measuring the accuracy, we first generate hybrid models of the base loop from each similarity cluster for 20 *training* runs. Then for 20 *test* runs, we check the number of clusters in which the PARS of at least one loop (other than the base loop) falls under 95% confidence interval at least once in the 20 *test* runs. We also measure how much overhead is reduced



(a) (b) (c) BT (d) kid_su3_rmd

Figure 2: (a) and (b) PARS improvement of the new hybrid approach over the previous dynamic approach. (c) and (d) P-value from F-test vs. PARS plot for the previous dynamic and hybrid approach for benchmarks. ** gqp and ksr stand for gp_quark_prop and kid_su3_rmd respectively.

Table I: Statistics for various experiments with NAS and MILC Benchmarks.

Benchmark	Avg. Parameter Reduction (%) (Sec IV-B)	Static Similar Clusters (%) (Sec IV-D)	Successful Clusters (%) (Sec IV-D)	Frequency Regulated Miss (%) (Sec IV-E)
BT	25	50	48	4.2
CG	11	34	33	3.2
EP	22	22	19	0.5
FT	32	23	23	1
IS	8	44	42	1.3
LU	6	32	29	2
MG	6	25	23	0
SP	10	22	22	2.7
gp_quark_prop	12	6	5	8
kid_su3_rmd	18	2	2	4

Table II: Memory and Time Trade-off for Various Message Sizes

Batch Size	Max. Memory Overhead (In Bytes)	Max. Calling Overhead
2	16	75
8	64	60.12
32	256	42.33
128	1024	25.12
256	2K	18.14
512	4K	9.11
1024	8K	9.05
2048	16K	8.95
4096	32K	8.93
8192	64K	8.93

if we apply the clustering strategy for the 20 *training* runs. These two statistics show us how much profiling and training overhead saved during the 20 *training* runs is worthwhile.

Column 3 of Table I shows the number of clusters formed as a percentage of total number of kernels. As seen from the column, a number of kernels can be clustered together using this approach and also for a major part of these clusters, the PARS of the models of loops other than the base loop on *test* runs fall under 95% confidence interval.

It is interesting to see that the percentage of clustered kernels in the NAS benchmarks is significantly higher than the MILC benchmarks. The `BT` benchmark has the highest percentage of *static* similar loops. A significant number of these loops come from the `copy_faces` subroutine that copies the face values of a variable defined on a set of cells to the overlap locations of the adjacent sets of cells. Because a set of cells interface in each direction with exactly one other set, there are similar loops to fill six different buffers. Then there are six similar loops for unpacking the received data.

The benchmark `IS` has a lot of similar loops that perform simple array assignments based on the variable `NUM_KEYS`, which is essentially the number of keys for the bucket sort algorithm and is computed by dividing the total number of keys by the number of processors.

The $Hy + G$ columns in Figure 3 show the overhead reduced using the clustering strategy as compared to the non-clustering hybrid approach (the *Hybr* columns) for NAS and MILC benchmarks. The best reduction is achieved in `BT` benchmark because `BT` has a high number of similar loops and also the prediction of the performance for the other loops using the model of the base loop in the cluster are accurate. The overhead reduction of MILC benchmarks are moderate using this strategy.

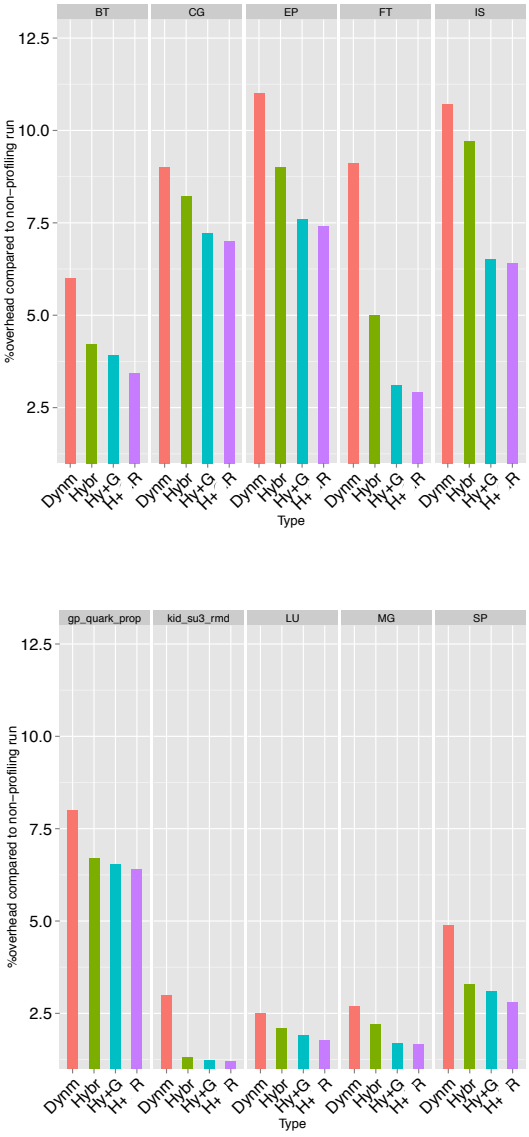


Figure 3: Overhead for the dynamic, hybrid approach and hybrid approach+various overhead reduction strategies. *Hy+G* shows the overhead after applying the strategy described in Section III-D2 and *H+R* shows overhead reduction after applying strategy described in Section III-D3 on top of the previous strategy.

E. Frequency of Model Update

As described before, model update frequency regulation not only reduces the overhead of model update for a good quality model but also allows a fall-back mechanism for poor quality models to improve. We use the running mean and running standard deviation of the PARS values measured dynamically [20] so that we can avoid the storage of previous PARS values of a particular model while applying the strategy.

Similar to the previous experiments, we generate models for 20 *training* runs of the benchmarks. Then we turn the frequency regulation strategy on and observed how the overhead is reduced in the *training* runs. We also report what percentage of *kernels* for which the regulation started, have at least one miss (where the new PARS goes above the confidence interval level) for the 20 *test* runs.

As seen from the $H + R$ columns in Figure 3, the frequency regulation strategy is able to further reduce the overhead when applied along with the similarity clustering strategies. `BT` and `SP` benefit most from this strategy for having a number of loops (that deals mainly with copying and packing/unpacking values to neighbors in various directions) with high iteration count that have generated models with good fits.

The last column of Table I shows the percentage of loops where there is at least one miss for the *test* runs. The percentage is not the lowest for `BT` and `SP` but still the high iteration count loops help reducing maximum overhead for these benchmarks. For MILC benchmarks, there are higher number of misses than NAS indicating possibility of either better possible models (*undefs* from static models and missing *predictors* with interaction terms from the improved EPMNF) or kernels whose performance can not be modeled as a linear function of input parameters.

F. Sample Models

1) LU: LU is a Lower-Upper Gauss-Seidel solver from the NAS parallel benchmark suit. The code fragment loops over three dimensional arrays `rsd` and `frct`, negates the values of the `frct` array and stores them to the same corresponding locations of the `rsd` array.

Listing 7: Example loop from LU

```
do k = 1, nz
  do j = 1, ny
     do i = 1, nx
        do m = 1, 5
           rsd(m,i,j,k) = - frct(m,i,j,k)
        end do
     end do
   end do
end do
```

The model generated by the hybrid approach for the above loop nest is:

$$f(P) = 1.41 \cdot nz \cdot ny \cdot nx + 4.75 \tag{15}$$

The previous online approach would start with a larger search space formed using EPMNF, causing more overhead and then produce a model with 15 *predictors* in it (not given due to space constraints). The PARS for the previous online model is 0.82 with a LOF p-value of 0.03, showing the necessity of the hybrid approach.

The hybrid model discovers through static analysis the interaction between nx, ny, nz that represent the number

of points in the 3D grid. Not only the overhead is reduced for a way smaller search space, but also the hybrid model has a PARS of 0.94 with a p-value of LOF 0.59, meaning the model fits well for the kernel.

2) GP_QUARK_PROP: The following loop from `hmom_action` function of the `d_action_rhmc.c` file of the `GP_QUARK_PROP` MILC benchmark calculates a sum over all sites on a particular node in the upward direction:

Listing 8: Example loop from MILC

```
sum=0.0;
FORALLSITES(i,s){
  for(dir=XUP;dir<=TUP;dir++)
   sum+=(double)ahmat_mag_sq(&(s->mom[dir]))
     -4.0;
}
```

The performance model for the loop generated from the hybrid approach is:

$$f(P) = nx \cdot ny \cdot nz \cdot nt \cdot (1.56 \cdot TUP - 0.49 \cdot XUP + 0.45) + 0.001 \quad (16)$$

The model from the pure online approach has 45 terms in it with a PARS of 0.75 and a p-value 0.01. But the hybrid model has a PARS of 0.88 with a p-value for F-test 0.40, indicating the model fits well for the kernel. The macro `TUP` is always greater than `XUP` and therefore the execution time does not become negative. The value of `i and s` are 0 and a variable `sites_on_node`, where `sites_on_node` is the multiplication of input parameters `nx`, `nx`, `ny` and `nt`. These information are gathered by source code backtracking (see Section III-C).

V. Related Work

The use of performance modeling manually has been explored before. There are approaches that focus on models generated for a very specific purpose but less on human-readable general-purpose models. For example, Ipek and de Supinski propose multi-layer artificial neural networks to learn application performance [9] and Lee and Brooks compare different schemes for automated machine-based performance learning and prediction [21]. Zhai, Chen, and Zheng extrapolate single-node performance to complex parallel machines [22]. Wu and Müller [23] extrapolate traces to larger process counts and can thus predict communication operations.

Hoefler and Gropp. aimed to popularize performance modeling by defining a simple six-step process to create application performance models [3]. Bauer, Gottlieb, and Hoefler show how to model performance variations using simple statistical tools [24].

Another objective of performance modeling is to predict application performance on a different target architecture. Carrington et al. propose a model-based prediction framework for applications on different computers [25], Marin and Mellor-Crummey demonstrate how application models can be derived semi-automatically to predict performance on different architectures [26], and Yang, Ma, and Müller model application performance on different architectures by running kernels on the target architecture [27]. Besides program inputs, *predictors* generated from hardware features can be used in our hybrid approach to model performance of applications across architectures. This poses an interesting direction of future work.

The authors of the Statistical Stall Breakdown [28] describe a mechanism that samples hardware counters and dynamically multiplexes hardware counters to compute a breakdown model for a PowerPC based microprocessor.

The work by Huck et al. [29] 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 [30] compiler infrastructure. The PerfExpert [31] tool employs the HPCToolkit [32] 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 of Pavlovic et al. [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 [34], HPCToolkit, 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.

Gonzalez, Gimenez and Labarta present a tool that automatically characterizes the different computation regions of the program [35]. Llort, Gonzalez and Servat detect clusters based on IPC and number of instructions committed and then detects the change of performance counters like cache misses inside the clusters [36].

Alam and Vetter propose code annotations, called "Modeling Assertions" [37] that combine empirical and analytical modeling techniques and help the developer to derive performance models for his code. Kerbyson and Alme propose a performance modeling approach [38] that is based on manually developed human expert knowledge about the application. Those modeling techniques rely on empirical execution of serial parts on the target architecture and are usually applied to stable codes which limits their usefulness

during software development.

The recent automatic online performance modeling strategy [5] had serious limitations as described in previous sections. Our hybrid strategy is the first technique that combines the knowledge obtained from static analysis and the power of dynamic analysis to produce more precise model. Our method can easily be extended for modeling other performance metrics such as cache misses,number of floating point instruction etc.

VI. Conclusion

We show how to combine static and dynamic analysis techniques in life-long performance modeling. Using static analysis on top of pure dynamic analysis enables us to discover *predictors* formed by the combination of *predictors*, which was not possible by a previous dynamic approach. Our improved dynamic analysis helps the deficiency of static analysis in resolving *undef* terms. The hybrid approach is able to generate models that are more precise and have a better prediction accuracy (up to 10% improvement). The hybrid models not only help to discover interaction among *predictors* but also reduce the cardinality of the *predictor* search space for the dynamic analysis greatly. We also show how batch update of models reduce the modeling overhead. We discover that there are a number of *static* similar loops that need not to be modeled individually. We also show, by regulating the frequency of model generation, a very small fraction of kernels in the benchmarks show variance in performance from the prediction of the generated models. We expect that this hybrid model generation strategy will greatly help performance engineers without enough domain knowledge to find possible performance bottlenecks for tuning self-modeling applications.

References

[1] 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*, ser. SC '13, 2013, pp. 45:1–45:12.

[2] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 77–88.

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

[4] N. R. Tallent and A. Hoisie, "Palm: Easing the burden of analytical performance modeling," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14, 2014, pp. 221–230.

[5] A. Bhattacharyya and T. Hoefler, "PEMOGEN: Automatic Adaptive Performance Modeling during Program Runtime," in *Proceedings of 23rd International Conference on Parallel Architecture and Compilation Techniques*, Aug. 2014.

[6] T. Hoefler and G. Kwasniewski, "Automatic Complexity Analysis of Explicitly Parallel Programs," in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'14)*. ACM, Jun. 2014.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[8] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The Polyhedral Model is More Widely Applicable Than You Think," in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, ser. CC'10/ETAPS'10, 2010, pp. 283–303.

[9] 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*, 2005, pp. 196–205.

[10] R. M. Karp, R. E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *J. ACM*, vol. 14, no. 3, pp. 563–590, Jul. 1967.

[11] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks–Summary and Preliminary Results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC)*, 1991, pp. 158–165.

[12] L. Huang, J. Jia, B. Yu, B. gon Chun, P. Maniatis, and M. Naik, "Predicting execution time of computer programs using sparse polynomial regression," in *Advances in Neural Information Processing Systems (NIPS) 23*, J. Lafferty, C. Williams, J. Shawe-taylor, R. Zemel, and A. Culotta, Eds., 2010, pp. 883–891.

[13] 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*, vol. 5, March 2005, pp. v/733–v/736 Vol. 5.

[14] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009. [3]

[15] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, ser. SAS '01, 2001, pp. 40–56.

[16] I. G. G. Kreft, I. Kreft, and J. de Leeuw, *Introducing Multilevel Modeling*, ser. ISM (London, England). SAGE Publications, 1998. [Online]. Available: http://books.google.com/books?id=tu2XjCN544YC

[3]Update (24.09.15): We cite the official technical report taken from the official Mantevo site.

[17] J. D. Hart, *Nonparametric Smoothing and Lack-of-Fit Tests*. Springer New York, 1997.

[18] J. R. Lackritz, "Ridge Regression: Biased Estimation for Nonorthogonal Problems," *The American Statistician*, vol. 38, no. 4, pp. 312–314, 1984.

[19] "MILC Code Version 7," http://www.physics.utah.edu/~detar/milc/milc_qcd.html.

[20] B. Welford, "Note on a Method for Calculating Corrected Sums of Squares and Products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[21] 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)*, 2007, pp. 249–258.

[22] J. Zhai, W. Chen, and W. Zheng, "Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node," *SIGPLAN Notices*, vol. 45, no. 5, pp. 305–314, 2010.

[23] X. Wu and F. Muller, "Scalaextrap: Trace-based communication extrapolation for SPMD programs," *ACM Transactions on Programming Languages and Systems*, vol. 34, no. 1, 2012.

[24] 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.

[25] L. Carrington, A. Snavely, and N. Wolter, "A Performance Prediction Framework for Scientific Applications," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 336–346, Feb. 2006.

[26] G. Marin and J. Mellor-Crummey, "Cross-architecture Performance Predictions for Scientific Applications Using Parameterized Models," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '04/Performance '04, 2004, pp. 2–13.

[27] L. T. Yang, X. Ma, and F. Mueller, "Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05, 2005, pp. 40–.

[28] 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*, 2005, pp. 101–110.

[29] 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*, 2008, pp. 49:1–49:10.

[30] 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.

[31] 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*, 2010, pp. 1–11.

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

[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] 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*, 2008, pp. 157–167.

[35] J. Gonzalez, J. Gimenez, and J. Labarta, "Automatic detection of parallel applications computation phases," in *IPDPS*, 2009, pp. 1–11.

[36] 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*, 2010, pp. 1–10.

[37] S. Alam and J. Vetter, "A framework to develop symbolic performance models of parallel applications," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[38] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings, "Predictive Performance and Scalability Modeling of a Large-scale Application," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, ser. SC '01, 2001, pp. 37–37.