Mantis: Automatic Performance Prediction for Smartphone Applications

Yongin Kwon[1], Sangmin Lee[2], Hayoon Yi[1], Donghyun Kwon[1], Seungjun Yang[1], Byung-Gon Chun[3], Ling Huang[4], Petros Maniatis[4], Mayur Naik[5], Yunheung Paek[1]

[1]*Seoul National University,* [2]*University of Texas at Austin,* [3]*Microsoft,* [4]*Intel,* [5]*Georgia Tech*

Abstract

We present Mantis, a framework for predicting the performance of Android applications on given inputs automatically, accurately, and efficiently. A key insight underlying Mantis is that program execution runs often contain features that correlate with performance and are automatically computable efficiently. Mantis synergistically combines techniques from program analysis and machine learning. It constructs concise performance models by choosing from many program execution features only a handful that are most correlated with the program's execution time yet can be evaluated efficiently from the program's input. We apply program slicing to accurately estimate the evaluation cost of a feature and automatically generate executable code snippets for efficiently evaluating features. Our evaluation shows that Mantis predicts the execution time of six Android apps with estimation error in the range of 2.2-11.9% by executing predictor code costing at most 1.3% of their execution time on Galaxy Nexus.

1 Introduction

Predicting the performance of programs on smartphones has many applications ranging from notifying estimated completion time to users, to better scheduling and resource management, to computation offloading [13, 14, 18]. The importance of these applications—and of program performance prediction—will only grow as smartphone systems become increasingly complex and flexible.

Many techniques have been proposed for predicting program performance. A key aspect of such techniques is what *features*, which characterize the program's input and environment, are used to model the program's performance. Most existing performance prediction techniques can be classified into two broad categories with regard to this aspect: automatic but domain-specific [7, 16, 21] or general-purpose but requiring user guidance [10, 17].

For techniques in the first category, features are chosen once and for all by experts, limiting the applicability of these techniques to programs in a specific domain. For example, to predict the performance of SQL query plans, a feature chosen once and for all could be the count of database operators occurring in the plan [16]. Techniques in the second category are general-purpose but require users to specify what program-specific features to use for each given program in order to predict its performance on different inputs. For instance, to predict the performance of a sorting program, such a technique may require users to specify the feature that denotes the number of input elements to be sorted. For techniques in either category, it is not sufficient merely to specify the relevant features: one must also manually provide a way to compute the value of each such feature from a given input and environment, e.g., by parsing an input file to sort and counting the number of items therein.

In this paper, we present Mantis, a new framework to predict online the performance of general-purpose bytecode programs on given inputs automatically, accurately, and efficiently. By being simultaneously general-purpose and automatic, our framework gains the benefits of both categories of existing performance prediction techniques without suffering the drawbacks of either. Since it uses neither domain nor expert knowledge to obtain relevant features, our framework casts a wide net and extracts a broad set of features from the given program itself to select relevant features using machine learning as done in our prior work [25]. During an offline stage, we execute an instrumented version of the program on a set of training inputs to compute values for those features; we use the training data set to construct a prediction model for online evaluation as new inputs arrive. The instrumented program tracks various features including the decisions made by each conditional in the program (*branch counts*), the number of times each loop in the program iterates (*loop counts*), the number of times each method is called (*method call counts*), and the values that are assumed by each program variable (*variable values*).

It is tempting to exploit features that are evaluated at late stages of program execution as such features may be strongly correlated with execution time. A drawback of naïvely using such features for predicting program performance, however, is that it takes as long to evaluate them as to execute almost the entire program. Our efficiency goal requires our framework to not only find features that are strongly correlated with execution time, but to also evalu-

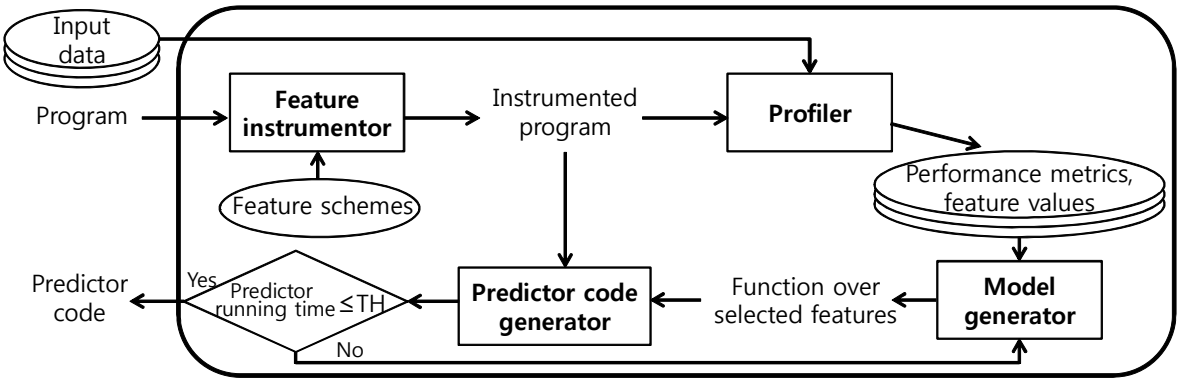

Figure 1: The Mantis offline stage.

ate those features significantly faster than running the program to completion.

To exploit such late-evaluated features, we use a program analysis technique called *program slicing* [44, 46]. Given a feature, slicing computes the set of all statements in the program that may affect the value of the feature. Precise slicing could prune large portions of the program that are irrelevant to the evaluation of features. Our slices are stand-alone executable programs; thus, executing them on program inputs provides both the evaluation cost and the value of the corresponding feature. Our application of slicing is novel; in the past, slicing has primarily been applied to program debugging and understanding.

We have implemented Mantis for Android applications and applied it to six CPU-intensive applications (Encryptor, Path Routing, Spam Filter, Chess Engine, Ringtone Maker, and Face Detection) on three smartphone hardware platforms (Galaxy Nexus, Galaxy S2, and Galaxy S3). We demonstrate experimentally that, with Galaxy Nexus, Mantis can predict the execution time of these programs with estimation error in the range of 2.2-11.9%, by executing slices that cost at most 1.3% of the total execution time of these programs. The results for Galaxy S2 and Galaxy S3 are similar. We also show that the predictors are accurate thanks to Android's scheduling policy even when the ambient CPU load on the smartphones increases.

We summarize the key contributions of our work:

- We propose a novel framework that automatically generates performance predictors using program-execution features with program slicing and machine learning.
- We have implemented our framework for Android-smartphone applications and show empirically that it can predict the execution time of various applications accurately and efficiently.

The rest of the paper is organized as follows. We present the architecture of our framework in Section 2. Sections 3 and 4 describe our feature instrumentation and performance-model generation, respectively. Section 5 describes predictor code generation using program slicing. In Section 6 we present our system implementation and evaluation results. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Architecture

In Mantis, we take a new white-box approach to automatically generate system performance predictors. Unlike traditional approaches, we extract information from the execution of the program, which is likely to contain key features for performance prediction. This approach poses the following two key challenges:

- What are good program features for performance prediction? Among many features, which ones are relevant to performance metrics? How do we model performance with relevant features?
- How do we compute features cheaply? How do we automatically generate code to compute feature values for prediction?

Mantis addresses the above challenges by synergistically combining techniques from program analysis and machine learning.

Mantis has an offline stage and an online stage. The offline stage, depicted in Figure 1, consists of four components: a feature instrumentor, a profiler, a performance-model generator, and a predictor code generator.

The feature instrumentor (Section 3), takes as input the program whose performance is to be predicted, and a set of *feature instrumentation schemes*. A scheme specifies a broad class of program features that are potentially correlated with the program's execution time. Examples of schemes include a feature for counting the number of times each conditional in the program evaluates to true, a

feature for the average of all values taken by each integer-typed variable in the program, etc. The feature instrumentor instruments the program to collect the values of features ($f_1, ..., f_M$) as per the schemes.

Next, the profiler takes the instrumented program and a set of user-supplied program inputs ($I_1, ..., I_N$). It runs the instrumented program on each of these inputs and produces, for each input I_i, a vector of feature values ($v_{i1}, ..., v_{iM}$). It also runs the original program on the given inputs and measures the performance metric (e.g., execution time (t_i)) of the program on that input.

The performance-model generator (Section 4) performs sparse nonlinear regression on the feature values and execution times obtained by the profiler, and produces a function (λ) that approximates the program's execution time using a subset of features ($f_{i1}, ..., f_{iK}$). In practice, only a tiny fraction of all M available features is chosen ($K \ll M$) since most features exhibit little variability on different program inputs, are not correlated or only weakly correlated with execution time, or are equivalent in value to the chosen features and therefore redundant.

As a final step, the predictor code generator (Section 5) produces for each of the chosen features a code snippet from the instrumented program. Since our requirement is to efficiently predict the program's execution time on given inputs, we need a way to efficiently evaluate each of the chosen features ($f_{i1}, ..., f_{iK}$) from program inputs.

We apply program slicing to extract a small code snippet that computes the value of each chosen feature. A precise slicer would prune large portions of the original program that are irrelevant to evaluating a given feature and thereby provide an efficient way to evaluate the feature. In practice, however, our framework must be able to tolerate imprecision. Besides, independent of the slicer's precision, certain features will be inherently expensive to evaluate: e.g., features whose value is computed upon program termination, rather than derived from the program's input. We define a feature as *expensive to evaluate* if the execution time of its slice exceeds a threshold (TH) expressed as a fraction of program execution time. If any of the chosen features ($f_{i1}, ..., f_{iK}$) is expensive, then via the *feedback loop* in Figure 1 (at the bottom), our framework re-runs the model generator, this time without providing it with the rejected features. The process is repeated until the model generator produces a set of features, all of which are deemed inexpensive by the slicer. In summary, the output of the offline stage of our framework is a predictor, which consists of a function (λ) over the final chosen features that approximates the program's execution time, along with a feature evaluator for the chosen features.

The online stage is straightforward: it takes a program input from which the program's performance must be predicted and runs the predictor module, which executes the feature evaluator on that input to compute feature values, and uses those values to compute λ as the estimated execution time of the program on that input.

3 Feature Instrumentation

We now present details on the four instrumentation schemes we consider: *branch counts*, *loop counts*, *method-call counts*, and *variable values*. Our overall framework, however, generalizes to all schemes that can be implemented by the insertion of simple tracking-statements into binaries or source.

Branch Counts: This scheme generates, for each conditional occurring in the program, two features: one counting the number of times the branch evaluates to true in an execution, and the other counting the number of times it evaluates to false. Consider the following simple example:

```
if (b == true) {
  /* heavy computation */ }
```

The execution time of this example would be strongly correlated with each of the two features generated by this scheme for condition (`b == true`). In this case, the two features are mutually-redundant and our performance-model generator could use either feature for the same cost. But the following example illustrates the need for having both features:

```
for (int i = 0; i < n; i++) {
  if (a[i] == 2) {
    /* light computation */ }
  else {
    /* heavy computation */ }
}
```

Picking the wrong branch of a conditional to count could result in a weakly correlated feature, penalizing prediction accuracy. The false-branch count is highly correlated with execution time, but the true-branch count is not.

Loop Counts: This scheme generates, for each loop occurring in the program, a feature counting the number of times it iterates in an execution. Clearly, each such feature is potentially correlated with execution time.

Method Call Counts: This scheme generates a feature counting the number of calls to each procedure. In case of recursive calls of methods, this feature is likely to correlate with execution time.

Variable Values: This scheme generates, for each statement that writes to a variable of primitive type in the program, two features tracking the sum and average of all values written to the variable in an execution. One can also instrument versions of variable values in program execution to capture which variables are static and what value changes each variable has. However, this creates too many feature values and we resort to the simpler scheme.

We instrument variable values for a few reasons. First, often the variable values obtained from input parameters

and configurations are changing infrequently, and these values tend to affect program execution by changing control flow. Second, since we cannot instrument all functions (e.g., system call handlers), the values of parameters to such functions may be correlated with their execution-time contribution; in a sense, variable values enable us to perform black-box prediction for the components of a program's execution trace that we cannot analyze. The following example illustrates this case:

```
int time = setfromargs(args);
Thread.sleep(time);
```

Similarly, variable value features can be equivalent to other types of features but significantly cheaper to compute. For example, consider the following Java program snippet:

```
void main(String[] args) {
  int n = Integer.parseInt(args[0]);
  for (int i = 0; i < n; i++) {...}
}
```

This program's execution time depends on the number of times the loop iterates, but the value of `n` can be used to estimate that number without executing the loop in the feature evaluator.

Other Features: We also considered the first k values (versions) of each variable. Our intuition is that often the variable values obtained from input parameters and configurations are changing infrequently, and these values tend to affect program execution by changing control flow. We rejected this feature since the sum and average metric captures infrequently-changing variable values well, and tracking k versions incurs higher instrumentation overheads. There might be other features that are helpful to prediction; exploring such features is future work.

4 Performance Modeling

Our feature instrumentation schemes generate a large number of features (albeit linear in the size of the program for the schemes we consider). Most of these features, however, are not expected to be useful for the performance prediction. In practice we expect a small number of these features to suffice in explaining the program's execution time well, and thereby seek a compact performance model, that is, a function of (nonlinear combinations of) just a few features that accurately approximates execution time. Unfortunately, we do not know a priori this handful of features and their nonlinear combinations that predict execution time well.

For a given program, our feature instrumentation profiler outputs a data set with N samples as tuples of $\{t_i, \mathbf{v}_i\}_{i=1}^{N}$, where $t_i \in \mathbb{R}$ denotes the i^{th} observation of execution time, and $\mathbf{v}_i$ denotes the i^{th} observation of the vector of M features.

Least square regression is widely used for finding the best-fitting $\lambda(\mathbf{v}, \beta)$ to a given set of responses t_i by minimizing the sum of the squares of the residuals [23]. However, least square regression tends to overfit the data and create complex models with poor interpretability. This does not serve our purpose since we have a lot of features but desire only a small subset of them to contribute to the model.

Another challenge we faced was that linear regression with feature selection would not capture all interesting behaviors by practical programs. Many such programs have non-linear, e.g., polynomial, logarithmic, or polylogarithmic complexity. So we were interested in non-linear models, which can be inefficient for the large number of features we had to contend with.

Regression with best subset selection finds for each $K \in \{1, 2, \ldots, M\}$ the subset of size K that gives the smallest Residual Sum of Squares (RSS). However, it is a discrete optimization problem and is known to be NP-hard [23]. In recent years a number of approximate algorithms have been proposed as efficient alternatives for simultaneous feature selection and model fitting. Widely used among them are LASSO (Least Absolute Shrinkage and Selection Operator) [43] and FoBa [48], an adaptive forward-backward greedy algorithm. The former, LASSO, is based on model regularization, penalizing low-selectivity, high-complexity models. It is a convex optimization problem, so efficiently solvable [15, 27]. The latter, FoBa, is an iterative greedy pursuit algorithm: during each iteration, only a small number of features are actually involved in model fitting, adding or removing the chosen features at each iteration to reduce the RSS. As shown FoBa has nice theoretical properties and efficient inference algorithms [48].

For our system, we chose the SPORE-FoBa algorithm, which we proposed [25], to build a predictive model from collected features. In our work, we showed that SPORE-FoBa outperforms LASSO and FoBa. The FoBa component of the algorithm helps cut down the number of interesting features first, and the SPORE component builds a fixed-degree (d) polynomial of all selected features, on which it then applies sparse, polynomial regression to build the model. For example, using a degree-2 polynomial with feature vector $\mathbf{v} = [x_1\ x_2]$, we expand out $(1 + x_1 + x_2)^2$ to get terms $1,\ x_1,\ x_2,\ x_1^2,\ x_1x_2,\ x_2^2$, and use them as basis functions to construct the following function for regression:

$$f(\mathbf{v}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_1 x_2 + \beta_5 x_2^2.$$

The resulting model can capture polynomial or sub-polynomial program complexities well thanks to Taylor expansion, which characterizes the vast majority of practical programs.

For a program whose execution time may dynamically

change over time as the workload changes, our performance model should evolve accordingly. The model can evolve in two ways: 1) the set of (non-linear) feature terms used in the model change; 2) with a fixed set of feature terms, their coefficients $\beta'_j s$ change. For a relatively stable program, we expect the former changes much less frequently than the latter. Using methods based on Stochastic Gradient Descent [9], it is feasible to update the set of feature terms and their coefficients $\beta'_j s$ online upon every execution time being collected.

5 Predictor Code Generation

The function output by the performance model generator is intended to efficiently predict the program's execution time on given program inputs. This requires a way to efficiently evaluate the features that appear in the function on those inputs. Many existing techniques rely on users to provide feature evaluators. A key contribution of our approach is the use of *static program slicing* [44, 46] to automatically extract from the (instrumented) program efficient feature evaluators in the form of *executable slices*—stand-alone executable programs whose sole goal is to evaluate the features. This section explains the rationale underlying our feature slicing (Section 5.1), describes the challenges of slicing and our approach to addressing them (Section 5.2), and provides the design of our slicer (Section 5.3).

5.1 Rationale

Given a program and a *slicing criterion* (p, v), where v is a program variable in scope at program point p, a *slice* is an executable sub-program of the given program that yields the same value of v at p as the given program, on all inputs. The goal of static slicing is to yield as small a sub-program as possible. It involves computing data and control dependencies for the slicing criterion, and excluding parts of the program upon which the slicing criterion is neither data- nor control-dependent.

In the absence of user intervention or slicing, a naïve approach to evaluate features would be to simply execute the (instrumented) program until all features of interest have been evaluated. This approach, however, can be grossly inefficient. Besides, our framework relies on feature evaluators to obtain the cost of each feature, so that it can iteratively reject costly features from the performance model. Thus, the naïve approach to evaluate features could grossly overestimate the cost of cheap features. We illustrate these problems with the naïve approach using two examples.

Example 1: A Java program may read a system property lazily, late in its execution, and depending upon its value decide whether or not to perform an expensive computation:

```
...;    // expensive computation S1
String s = System.getProperty(...);
if (s.equals(...)) {
  f_true++; // feature instrumentation
  ...; // expensive computation S2
}
```

In this case, feature `f_true` generated by our framework to track the number of times the above branch evaluates to true will be highly predictive of the execution time. However, the naïve approach for evaluating this feature will always perform the expensive computation denoted by `S1`. In contrast, slicing this program with slicing criterion (`p_exit`, `f_true`), where `p_exit` is the exit point of the program, will produce a feature evaluator that excludes `S1` (and `S2`), assuming the value of `f_true` is truly independent of computation `S1` and the slicer is precise enough.

Example 2: This example illustrates a case in which the computation relevant to evaluating a feature is interleaved with computation that is expensive but irrelevant to evaluating the feature. The following program opens an input text file, reads each line in the file, and performs an expensive computation on it (denoted by the call to the `process` method):

```
Reader r = new Reader(new File(name));
String s;
while ((s = r.readLine()) != null) {
  f_loop++;    // feature inst.
  process(s); // expensive computation
}
```

Assuming the number of lines in the input file is strongly correlated with the program's execution time, the only highly predictive feature available to our framework is `f_loop`, which tracks the number of iterations of the loop. The naïve approach to evaluate this feature will perform the expensive computation denoted by the `process` method in each iteration, even if the number of times the loop iterates is independent of it. Slicing this program with slicing criterion (`p_exit`, `f_loop`), on the other hand, can yield a slice that excludes the calls to `process(s)`.

The above two examples illustrate cases where the feature is fundamentally cheap to evaluate but slicing is required because the program is written in a manner that intertwines its evaluation with unrelated expensive computation.

5.2 Slicer Challenges

There are several key challenges to effective static slicing. Next we discuss these challenges and the approaches we take to address them. Three of these are posed by program artifacts—procedures, the heap, and concurrency—

and the fourth is posed by our requirement that the slices be executable.

Inter-procedural Analysis: The slicer must compute data and control dependencies efficiently and precisely. In particular, it must propagate these dependencies *context-sensitively*, that is, only along inter-procedurally realizable program paths—doing otherwise could result in inferring false dependencies and, ultimately, grossly imprecise slices. Our slicer uses existing precise and efficient inter-procedural algorithms from the literature [24, 33].

Alias Analysis: False data dependencies (and thereby false control dependencies as well) can also arise due to *aliasing*, i.e., two or more expressions pointing to the same memory location. Alias analysis is expensive. The use of an imprecise alias analysis by the slicer can lead to false dependencies. Static slicing needs may-alias information—analysis identifying expressions that may be aliases in at least some executions—to conservatively compute all data dependencies. In particular, it must generate a data dependency from an instance field write `u.f` (or an array element write `u[i]`) to a read `v.f` (or `v[i]`) in the program if `u` and `v` may-alias. Additionally, static slicing can also use must-alias information if available (expressions that are always aliases in all executions), to kill dependencies that no longer hold as a result of instance field and array element writes in the program. Our slicer uses a flow- and context-insensitive may-alias analysis with object allocation site heap abstraction [29].

Concurrency Analysis: Multi-threaded programs pose an additional challenge to static slicing due to the possibility of inter-thread data dependencies: reads of instance fields, array elements, and static fields (i.e., global variables) are not just data-dependent on writes in the same thread, but also on writes in other threads. Precise static slicing requires a precise static race detector to compute such data dependencies. Our may-alias analysis, however, suffices for our purpose (a race detector would perform additional analyses like thread-escape analysis, may-happen-in-parallel analysis, etc.)

Executable Slices: We require slices to be executable. In contrast, most of the literature on program slicing focuses on its application to program debugging, with the goal of highlighting a small set of statements to help the programmer debug a particular problem (e.g., Sirdharan et al. [40]). As a result, their slices do not need to be executable. Ensuring that the generated slices are executable requires extensive engineering so that the run-time does not complain about malformed slices, e.g., the first statement of each constructor must be a call to the super constructor even though the body of that super constructor is sliced away, method signatures must not be altered, etc.

5.3 Slicer Design

Our slicer combines several existing algorithms to produce executable slices. The slicer operates on a three-address-like intermediate representation of the bytecode of the given program.

Computing System Dependence Graph: For each method reachable from the program's root method (e.g., `main`) by our call-graph analysis, we build a Program Dependence Graph (PDG) [24], whose nodes are statements in the body of the method and whose edges represent intra-procedural data/control dependencies between them. For uniform treatment of memory locations in subsequent steps of the slicer, this step also performs a mod-ref analysis[1] and creates additional nodes in each PDG denoting implicit arguments for heap locations and globals possibly read in the method, and return results for those possibly modified in the method.

The PDGs constructed for all methods are stitched into a System Dependence Graph (SDG) [24], which represents inter-procedural data/control dependencies. This involves creating extra edges (so-called linkage-entry and linkage-exit edges) linking actual to formal arguments and formal to actual return results, respectively.

In building PDGs, we handle Java native methods, which are built with JNI calls, specially. We implement simple stubs to represent these native methods for the static analysis. We examine the code of the native method and write a stub that has the same dependencies between the arguments of the method, the return value of the method, and the class variables used inside the method as does the native method itself. We currently perform this step manually. Once a stub for a method is written, the stub can be reused for further analyses.

Augmenting System Dependence Graph: This step uses the algorithm by Reps, Horwitz, Sagiv, and Rosay [33] to augment the SDG with summary edges, which are edges summarizing the data/control dependencies of each method in terms of its formal arguments and return results.

Two-Pass Reachability: The above two steps are more computationally expensive but are performed once and for all for a given program, independent of the slicing criterion. This step takes as input a slicing criterion and the augmented SDG, and produces as output the set of all statements on which the slicing criterion may depend. It uses the two-pass backward reachability algorithm proposed by Horwitz, Reps, and Binkley [24] on the augmented SDG.

Translation: As a final step, we translate the slicer code based on intermediate representation to bytecode.

Extra Steps for Executable Slices: A set of program

[1] This finds all expressions that a method may *mod*ify-*ref*erence directly, or via some method it transitively calls.

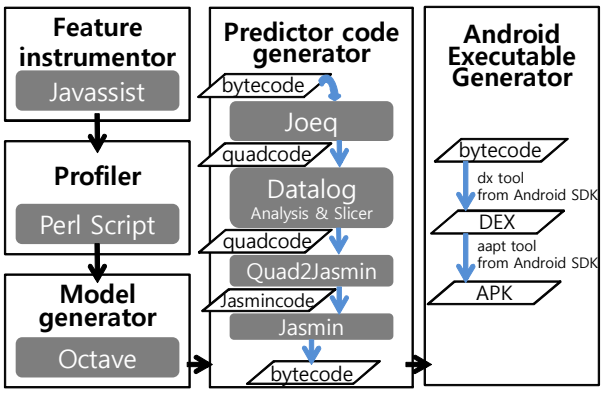

Figure 2: Mantis prototype toolchain.

statements identified by the described algorithm may not meet Java language requirements. This problem needs to be resolved to create executable slices.

First, we need to handle accesses to static fields and heap locations (instance fields and array elements). Therefore, when building an SDG, we identify all such accesses in a method and create formal-in vertices for those read and formal-out for those written along with corresponding actual-in and actual-out vertices. Second, there may be uninitialized parameters if they are not included in a slice. We opt to keep method signatures, hence we initialize them with default values. Third, there are methods not reachable from a main method but rather called from the VM directly (e.g., class initializers). These methods will not be included in a slice by the algorithm but still may affect the slicing criterion. Therefore, we do not slice out such code. Fourth, when a new object creation is in a slice, the corresponding constructor invocation may not. To address this, we create a control dependency between object creations and corresponding constructor invocations to ensure that they are also in the slice. Fifth, a constructor of a class except the Object class must include a call to a constructor of its parent class. Hence we include such calls when they are missing in a slice. Sixth, the first parameter of an instance method call is a reference to the associated object. Therefore if such a call site is in a slice, the first parameter has to be in the slice too and we ensure this.

6 Evaluation

We have built a prototype of Mantis implementing the instrumentor, profiler, model generator and predictor code generator (Figure 2). The prototype is built to work with Android application binaries. We implemented the feature instrumentor using Javassist [2], which is a Java bytecode rewriting library. The profiler is built using scripts to run the instrumented program on the test inputs and then the results are used by the model generator, which is written in Octave [4] scripts. Finally, we implemented our predictor code generator in Java and Datalog by extending JChord [3], a static and dynamic Java program-analysis tool. JChord uses the Joeq Java compiler framework to convert the bytecode of the input Java program, one class file at a time, into a three-address-like intermediate code called quadcode, which is more suitable for analysis. The predictor code generator produces the Joeq quadcode slice, which is the smallest subprogram that could obtain the selected features. Each quad instruction is translated to a corresponding set of Jasmin [1] assembly code, and then the Jasmin compiler generates the final Java bytecode.

We have applied the prototype to Android applications. Before Android applications are translated to Dalvik Executables (DEX), their Java source code is first compiled into Java bytecode. Mantis works with this bytecode and translates it to DEX to run on the device. Mantis could work with DEX directly, as soon as a translator from DEX to Joeq becomes available.

6.1 Experimental Setup

We run our experiments with a machine to run the instrumentor, model generator, and predictor code generator, as well as a smartphone to run the original and instrumented codes for profiling and generated predictor codes for slicing evaluation. The machine runs Ubuntu 11.10 64-bit with a 3.1GHz quad-core CPU, and 8GB of RAM. The smartphone is a Galaxy Nexus running Android 4.1.2 with dual-core 1.2Ghz CPU and 1GB RAM. All experiments were done using Java SE 64-bit 1.6.0_30.

The selected applications — Encryptor, Path Routing, Spam Filter, Chess Engine, Ringtone Maker and Face Detection — cover a broad range of CPU-intensive Android-application functionalities. Their execution times are sensitive to inputs, so challenging to model. Below we describe the applications and the input dataset we used for experiments in detail.

We evaluate Mantis on 1,000 randomly generated inputs for each application. These inputs achieve 95-100% basic-block coverage, only missing exception handling. We train our predictor on 100 inputs, and use the rest to test the predictor model. For each platform, we run Mantis to generate predictors and measure the prediction error and running time. The threshold is set to 5%, which means a generated predictor is accepted only if the predictor running time is less than 5% of the original program's completion time.

Encryptor: This encrypts a file using a matrix as a key. Inputs are the file and the matrix key. We use 1,000 files, each with its own matrix key. File size ranges from 10 KB to 8000 KB, and keys are 200×200 square matrices.

Path Routing: This computes the shortest path from one point to another on a map (as in navigation and game ap-

Application	Prediction error (%)	Prediction time (%)	No. of detected features	No. of chosen features
Encryptor	3.6	0.18	28	2
Path Routing	4.2	1.34	68	1
Spam Filter	2.8	0.51	55	1
Chess Engine	11.9	1.03	1084	2
Ringtone Maker	2.2	0.20	74	1
Face Detection	4.9	0.62	107	2

Table 1: Prediction error, prediction time, the total number of features initially detected and the number of chosen features.

Application	Selected features	Generated model
Encryptor	Matrix-key size (f_1), Loop count of encryption (f_2)	$c_0{f_1}^2 f_2 + c_1{f_1}^2 + c_2 f_2 + c_3$
Path Routing	Build map loop count (f_1)	$c_0{f_1}^2 + c_1 f_1 + c_2$
Spam Filter	Inner loop count of sorting (f_1)	$c_0 f_1 + c_1$
Chess Engine	No. of game-tree leaves (f_1), No. of chess pieces (f_2)	$c_0{f_1}^3 + c_1 f_1 f_2 + c_2{f_2}^2 + c_3$
Ringtone Maker	Cut interval length (f_1)	$c_0 f_1 + c_1$
Face Detection	Width of image (f_1), Height of image (f_2)	$c_0 f_1 f_2 + c_1{f_2}^2 + c_2$

Table 2: Selected features and generated prediction models.

plications). We use 1,000 maps, each with 100-200 locations, and random paths among them. We queried a route for a single random pair of locations for each map.

Spam Filter: This application filters spam messages based on a collective database. At initialization, it collects the phone numbers of spam senders from several online databases and sorts them. Then it removes white-listed numbers (from the user's phonebook) and builds a database. Subsequently, messages from senders in the database are blocked. We test Mantis with the initialization step; filtering has constant duration. We use 1,000 databases, each with 2,500 to 20,000 phone numbers.

Chess Engine: This is the decision part of a chess application. Similarly to many game applications, it receives the configuration of chess pieces as input and determines the best move using the Minimax algorithm. We set the game-tree depth to three. We use 1,000 randomly generated chess-piece configurations, each with up to 32 chess pieces.

Ringtone maker: This generates customized ringtones. Its input is a wav-format file and a time interval within the file. The application extracts that interval from the audio file and generates a new mp3 ringtone. We use 1,000 wav files, ranging from 1 to 10 minutes, and intervals starting at random positions and of lengths between 10 and 30 seconds.

Face Detection: This detects faces in an image by using the OpenCV library. It outputs a copy of the image, outlining faces with a red box. We use 1,000 images, of sizes between 100×100 and 900×3000 pixels.

6.2 Experiment Results

Accurate and Efficient Prediction: We first evaluate the accuracy and efficiency of Mantis prediction. Table 1 reports the prediction error and predictor running time of Mantis-generated predictors, the total number of features initially detected, and the number of features actually chosen to build the prediction model for each application. The "prediction error" column measures the accuracy of our prediction. Let $A(i)$ and $E(i)$ denote the actual and predicted execution times, respectively, computed on input i. Then, this column denotes the prediction error of our approach as the average value of $|A(i) - E(i)|/A(i)$ over all inputs i. The "prediction time" measures how long the predictor runs compared to the original program. Let $P(i)$ denote the time to execute the predictor. This column denotes the average value of $P(i)/A(i)$ over all inputs i.

Mantis achieves accuracy with prediction error within 5% in most cases, while each predictor runs around 1% of the original application's execution time, which is well under the 5% limit we assigned to Mantis.

We also show the effect of the number of training samples on prediction errors in Figure 3. For four applications, the curve of their prediction error plateaus before 50 input samples for training. For Chess Engine and Encryptor, the curve plateaus around 100 input samples for training. Since there is little to gain after the curve plateaus, we only use 100 input samples for training Mantis. Even for bigger input datasets of 10,000 samples, we only need about 100 input samples for training to obtain similar prediction accuracy.

Mantis generated interpretable and intuitive prediction

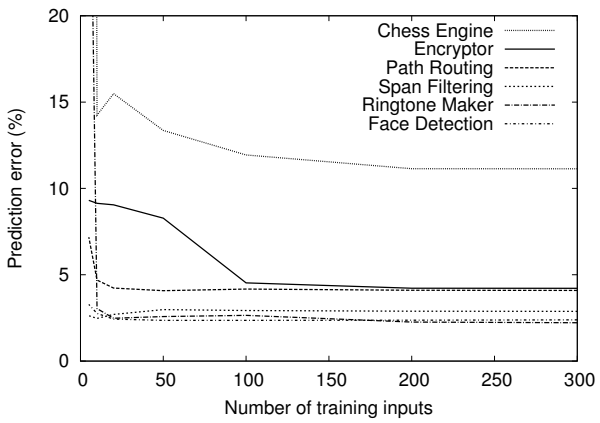

Figure 3: Prediction errors varying the number of input samples. The y-axis is truncated to 20 for clarity.

models by only choosing one or two among the many detected features unlike non-parametric methods. Table 2 shows the selected features and the generated polynomial prediction model for each application. In the model, c_n represents a constant real coefficient generated by the model generator and f_n represents the selected feature. The selected features are important factors in execution time, and they often interact in a non-linear way, which Mantis captures accurately. For example, for Encryptor, Mantis uses non-linear feature terms ($f_1^2 f_2$, f_1^2) to predict the execution time accurately.

Now we explain why Chess Engine has a higher error rate. Its execution time is related to the number of leaf nodes in the game tree. However, this feature can only be obtained late in the application execution and is dependent on almost all code that comes before it. Therefore, Mantis rejects this feature because it is too expensive. Note that we set the limit of predictor execution time to be 5% of the original application time. As the expensive feature is not usable, Mantis chooses alternative features: the number of nodes in the second level of the game tree and the number of chess pieces left; these features can capture the behavior of the number of leaf nodes in the game tree. Although they can only give a rough estimate of the number of leaf nodes in the game tree, the prediction error is still around only 12%.

Benefit of Non-linear Terms on Prediction Accuracy: Table 3 shows the prediction error rates of the models built by Mantis and Mantis-linear. Mantis-linear uses only linear terms (f_i's) for model generation. For Encryptor, Path Routing, and Face Detection, non-linear terms improve prediction accuracy significantly since Mantis-linear does not capture the interaction between features.

Benefit of Slicing on Prediction Time: Next we discuss how slicing improves the prediction time. In Table 4, we compare the prediction times of Mantis-generated predictors with those of predictors built with *partial execution*. Partial Execution (PE) runs the instrumented program only until the point where we obtain the chosen feature values.

Application	Mantis pred. error (%)	Mantis-linear pred. error (%)
Encryptor	3.6	6.6
Path Routing	4.2	13.8
Spam Filter	2.8	2.8
Chess Engine	11.9	13.2
Ringtone Maker	2.2	2.2
Face Detection	4.9	52.7

Table 3: Prediction error of Mantis and Mantis-linear. Mantis-linear uses only linear terms (f_i's) for model generation.

Application	Mantis pred. time (%)	PE pred. time (%)
Encryptor	0.20	100.08
Path Routing	1.30	17.76
Spam Filter	0.50	99.39
Chess Engine	1.03	69.63
Ringtone Maker	0.20	0.04
Face Detection	0.61	0.17

Table 4: Prediction time of Mantis and PE.

Mantis reduces the prediction time significantly for Encryptor, Path Routing, Spam Filter, and Chess Engine. For these applications, PE predictors need to run a large piece of code, which includes code that is unrelated to the chosen features until their values are obtained.

Spam Filter and Encryptor are the worst cases for PE since the last updates of the chosen feature values occur near the end of their execution. In contrast, Ringtone Maker and Face Detection can obtain the chosen feature values cheaply even without slicing. This is because the values for the chosen features can be obtained at the very beginning in the application's execution. In fact, the Mantis-generated predictors of these applications take longer than PE because the generated code is less optimized than the code generated directly by the compiler.

Benefit of Slicing on Prediction Accuracy: To show the effect of slicing on prediction accuracy under a prediction time limit, we compare our results with those obtained using *bounded execution*. Bounded Execution (BE) gathers features by running an instrumented application for only a short period of time, which is the same as the time a Mantis-generated predictor would run. It then uses these gathered features with the Mantis model generator to build a prediction model.

As shown in Table 5, the prediction error rates of the models built by BE are much higher than those of the

Application	Galaxy S2		Galaxy S3	
	Prediction error (%)	Prediction time (%)	Prediction error (%)	Prediction time (%)
Encryptor	4.6	0.35	3.4	0.08
Path Routing	4.1	3.07	4.2	1.28
Spam Filter	5.4	1.52	2.2	0.52
Chess Engine	9.7	1.42	13.2	1.38
Ringtone Maker	3.7	0.51	4.8	0.20
Face Detection	5.1	1.28	5.0	0.69

Table 6: Prediction error and time of Mantis running with Galaxy S2 and Galaxy S3.

Application	Mantis pred. error (%)	BE pred. error (%)
Encryptor	3.6	56.0
Path Routing	4.2	64.0
Spam Filter	2.8	36.2
Chess Engine	11.9	26.1
Ringtone Maker	2.2	2.2
Face Detection	4.9	4.9

Table 5: Prediction error of Mantis and BE.

models built by Mantis. This is because BE cannot exploit as many features as Mantis. For Spam Filter and Encryptor, no usable feature can be obtained by BE; thus, BE creates a prediction model with only a constant term for each of the two applications.

Prediction on Different Hardware Platforms: Next we evaluate whether Mantis generates accurate and efficient predictors on three different hardware platforms. Table 6 shows the results of Mantis with two additional smartphones: Galaxy S2 and Galaxy S3. Galaxy S2 has a dual-core 1.2Ghz CPU and 1GB RAM, running Android 4.0.3. Galaxy S3 has a quad-core 1.4Ghz CPU and 1GB RAM, running Android 4.0.4. As shown in the table, Mantis achieves low prediction errors and short prediction times with Galaxy S2 and Galaxy S3 as well. For each application, Mantis builds a model similar to the one generated for Galaxy Nexus. The chosen features for each device are the same as or equivalent (e.g., there can be multiple instrumented variables with the same value) to the chosen features for Galaxy Nexus, while the model coefficients are changed to match the speed of each device. The result shows that Mantis generates predictors robustly with different hardware platforms.

Prediction under Background Load: Finally, we evaluate how the predictors perform under changing environmental loads. Table 7 shows how much effect CPU-intensive loads have on the performance of Mantis predictors for Galaxy Nexus. The application execution times under the background CPU-intensive load are compared to the predicted execution times of Mantis predictors generated with an idle smartphone. The background load is generated by the SysBench package [5], which consists of a configurable number of events that compute prime numbers. For our evaluation, the load is configured to initially have a steady 50%, 75%, or 99% CPU usage. The test applications run in the foreground.

Application	Pred. error (%) for the x% background CPU load			
	x=0	x=50	x=75	x=99
Encryptor	3.6	7.5	10.5	21.3
Path Routing	4.2	5.3	5.8	6.7
Spam Filter	2.8	4.7	5.2	5.8
Chess Engine	11.9	13.5	15.3	15.8
Ringtone Maker	2.2	2.3	3.0	3.1
Face Detection	4.9	5.3	5.6	5.8

Table 7: Prediction error of Mantis-generated predictors for Galaxy Nexus under background CPU-intensive loads.

As shown in the table, in most cases background load has only a moderate effect on the accuracy of Mantis predictors. This is mainly due to Android's scheduling policy, which gives a higher priority to the process that is actively running on the screen, or foreground, compared with the other processes running in the background. We observed that when an application was started and brought to the foreground, the Android system secured enough CPU time for the process to run smoothly by reducing the CPU occupancy of the background load.

However, the prediction error for Encryptor increases as the CPU load increases. Unlike the other applications, Encryptor creates a larger number of heap objects and calls Garbage Collection (GC) more often. We also observed that GC slows down under the heavy load, resulting in a slowdown of Encryptor's total execution time. This in turn makes it difficult for the Mantis predictor to predict the Encryptor execution time accurately under a heavy load. An extension of Mantis is to include environmental factors (e.g., the background CPU load) as features in Mantis prediction models.

Mantis Offline Stage Processing Time: Table 8 presents

Application	Profiling	Model gen.	Slicing	Test	Total	Iterations
Encryptor	2373	18	117	391	2900	3
Path Routing	363	28	114	14	519	3
Spam Filter	135	10	66	3	214	2
Chess Engine	6624	10229	6016	23142	46011	83
Ringtone Maker	2074	19	4565	2	6659	1
Face Detection	1437	13	6412	179	8041	4

Table 8: Mantis offline stage processing time (in seconds).

Mantis offline stage processing (profiling, model generation, slicing, and testing) time for all input training data. The total time is the sum of times of all steps. The last column shows how many times Mantis ran the model generation and slicing part due to rejected features. For the applications excluding Chess Engine, the total time is less than a few hours, the profiling part dominates, and the number of iterations in the feedback loop is small. Chess Engine's offline processing time takes 12.8 hours because of many rejected features. We leave speeding up this process as future work.

Summary: We have demonstrated that our prototype implemenation of Mantis generates good predictors for our test programs that estimate running time with high accuracy and very low cost. We have also compared our approach to simpler, intuitive approaches based on Partial Execution and Bounded Execution, showing that Mantis does significantly better in almost all cases, and as well in the few cases where Partial Execution happened upon good prediction features. Finally, we showed that Mantis predictors are accurate on three different hardware platforms and are little affected by variability in background CPU load.

7 Related Work

Much research has been devoted to modeling system behavior as a means of prediction for databases [16, 21], cluster computing [8, 39], networking [12, 31, 41], program optimization [26, 42], etc.

Prediction of basic program characteristics, execution time, or even resource consumption, has been used broadly to improve scheduling, provisioning, and optimization. Example domains include prediction of library and benchmark performance [28, 45], database query execution-time and resource prediction [16, 21], performance prediction for streaming applications based on control flow characterization [6], violations of Service-Level Agreements (SLAs) for cloud and web services [8, 39], and load balancing for network monitoring infrastructures [7]. Such work demonstrates significant benefits from prediction, but focuses on problem domains that have identifiable features (e.g., operator counts in database queries, or network packet header values) based on expert knowledge, use domain-specific feature extraction that may not apply to general-purpose programs, or require high correlation between simple features (e.g., input size) and execution time.

Delving further into extraction of non-trivial features, research has explored extracting predictors from execution traces to model program complexity [17], to improve hardware simulation specificity [37, 38], and to find bugs cooperatively [32]. There has also been research on multi-component systems (e.g., content-distribution networks) where the whole system may not be observable in one place. For example, extracting component dependencies (web objects in a distributed web service) can be useful for what-if analysis to predict how changing network configuration will impact user-perceived or global performance [12, 31, 41].

A large body of work has targeted worst-case behavior prediction, either focusing on identifying the inputs that cause it, or on estimating a tight upper bound [22, 30, 35, 36, 47] in embedded and/or real-time systems. Such efforts are helped by the fact that, by construction, the systems are more amenable to such analysis, for instance thanks to finite bounds on loop sizes. Other work focuses on modeling algorithmic complexity [17], simulation to derive worst-case running time [34], and symbolic execution and abstract evaluation to derive either worst-case inputs for a program [11], or asymptotic bounds on worst-case complexity [19, 20]. In contrast, our goal is to automatically generate an online, accurate predictor of the performance of particular invocations of a general-purpose program.

Finally, Mantis's machine learning algorithm for prediction is based on our earlier work [25]. In the prior work, we computed program features manually. In this work, we introduce program slicing to compute features cheaply and generate predictors automatically, apply our whole system to Android smartphone applications on multiple hardware platforms, and evaluate the benefits of slicing thoroughly.

8 Conclusion

In this paper, we presented Mantis, a framework that automatically generates program performance predictors that

can estimate performance accurately and efficiently. Mantis combines program slicing and sparse regression in a novel way. The key insight is that we can extract information from program executions, even when it occurs late in execution, cheaply by using program slicing and generate efficient feature evaluators in the form of executable slices. Our evaluation shows that Mantis can automatically generate predictors that estimate execution time accurately and efficiently for smartphone applications. As future work, we plan to explore how to extend Mantis to predict other metrics like resource consumption and evaluate Mantis for diverse applications.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and our shepherd, Paul Leach, for his guidance.

References

[1] Jasmin. `jasmin.sourceforge.net`.

[2] Javassist. `www.csg.is.titech.ac.jp/~chiba/javassist`.

[3] JChord. `code.google.com/p/jchord`.

[4] Octave. `www.gnu.org/software/octave`.

[5] Sysbench: a system performance benchmark. `http://sysbench.sourceforge.net/`.

[6] F. Aleen, M. Sharif, and S. Pande. Input-Driven Dynamic Execution Behavior Prediction of Streaming Applications. In *PPoPP*, 2010.

[7] P. Barlet-Ros, G. Iannaccone, J. Sanjuas-Cuxart, D. Amores-Lopez, and J. Sole-Pareta. Load Shedding in Network Monitoring Applications. In *USENIX*, 2007.

[8] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. In *HotCloud*, 2009.

[9] L. Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *COMPSTAT*, 2010.

[10] E. Brewer. High-Level Optimization via Automated Statistical Modeling. In *PPoPP*, 1995.

[11] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, 2009.

[12] S. Chen, K. Joshi, M. A. Hiltunen, W. H. Sanders, and R. D. Schlichting. Link Gradients: Predicting the Impact of Network Latency on Multitier Applications. In *INFOCOM*, 2009.

[13] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *EuroSys*, 2011.

[14] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *MobiSys*, 2010.

[15] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least Angle Regression. *Annals of Statistics*, 32(2), 2002.

[16] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*, 2009.

[17] S. Goldsmith, A. Aiken, and D. Wilkerson. Measuring Empirical Computational Complexity. In *FSE*, 2007.

[18] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code Offload by Migrating Execution Transparently. In *OSDI*, 2012.

[19] B. Gulavani and S. Gulwani. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *CAV*, 2008.

[20] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *POPL*, 2009.

[21] C. Gupta, A. Mehta, and U. Dayal. PQR: Predicting Query Execution Times for Autonomous Workload Management. In *ICAC*, 2008.

[22] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *RTSS*, 2006.

[23] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2009.

[24] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *PLDI*, 1988.

[25] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. In *NIPS*, 2010.

[26] Y. Jiang, E. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao. Exploiting statistical correlations for proactive prediction of program behaviors. In *CGO*, 2010.

[27] S.-J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky. An Interior-Point Method for Large-Scale l1-Regularized Least Squares. *IEEE J-STSP*, 1(4), 2007.

[28] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS*, 2006.

[29] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, School of Computer Science, McGill University, 2006.

[30] Y.-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwar Academic Publishers, 1999.

[31] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: Automating Performance Prediction for Web Services. In *NSDI*, 2010.

[32] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *PLDI*, 2005.

[33] T. W. Reps, S. Horwitz, S. Sagiv, and G. Rosay. Speeding up Slicing. In *FSE*, 1994.

[34] R. Rugina and K. E. Schauser. Predicting the Running Times of Parallel Programs by Simulation. In *IPPS/SPDP*, 1998.

[35] S. Seshia and A. Rakhlin. Game-Theoretic Timing Analysis. In *ICCAD*, 2008.

[36] S. Seshia and A. Rakhlin. Quantitative Analysis of Systems Using Game-Theoretic Learning. *ACM TECS*, 2010.

[37] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *PACT*, 2001.

[38] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS*, 2002.

[39] P. Shivam, S. Babu, and J. S. Chase. Learning Application Models for Utility Resource Planning. In *ICAC*, 2006.

[40] M. Sridharan, S. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.

[41] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar. Answering What-If Deployment and Configuration Questions with WISE. In *SIGCOMM*, 2008.

[42] K. Tian, Y. Jiang, E. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *OOPSLA*, 2010.

[43] R. Tibshirani. Regression Shrinkage and selection via the Lasso. *J. Royal. Statist. Soc B.*, 1996.

[44] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3), 1995.

[45] K. Vaswani, M. Thazhuthaveetil, Y. Srikant, and P. Joseph. Microarchitecture Sensitive Empirical Models for Compiler Optimizations. In *CGO*, 2007.

[46] M. Weiser. Program Slicing. In *ICSE*, 1981.

[47] R. Wilhelm. Determining Bounds on Execution Times. *Handbook on Embedded Systems*, 2005.

[48] T. Zhang. Adaptive forward-backward greedy algorithm for sparse learning with linear models. In *NIPS*, 2008.