Mantis: Efficient Predictions of Execution Time, Energy Usage, Memory Usage and Network Usage on Smart Mobile Devices

Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, *Student Member, IEEE*, Byung-gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek, *Member, IEEE*

Abstract—We present Mantis, a framework for predicting the computational resource consumption (CRC) of Android applications on given inputs accurately, and efficiently. A key insight underlying Mantis is that program codes often contain features that correlate with performance and these features can be automatically computed efficiently. Mantis synergistically combines techniques from program analysis and machine learning. It constructs concise CRC models by choosing from many program execution features only a handful that are most correlated with the program's CRC metric yet can be evaluated efficiently from the program's input. We apply program slicing to reduce evaluation time of a feature and automatically generate executable code snippets for efficiently evaluating features. Our evaluation shows that Mantis predicts four CRC metrics of seven Android apps with estimation error in the range of 0-11.1 percent by executing predictor code spending at most 1.3 percent of their execution time on Galaxy Nexus.

Index Terms—Prediction, program analysis, machine learning, slicing, smartphone, computational resource consumption

1 Introduction

Predicting the consumption of computational resources, such as computation time, memory capacity, energy consumption and network characteristics, of programs on smart mobile devices has many applications such as, notifying the estimated completion time to users, achieving better scheduling and resource management, testing applications, detecting anomalies, or offloading computation [1], [2], [3]. 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 the computational resource consumption (CRC) of programs. A key aspect of such techniques is what *features*, which characterize the program's input and environment, are used to model the program's CRC. Features that are trivial and efficient to obtain, such as input parameters, input size or cpu speed, can be enough to build an accurate predictor for some applications [4]. However, in many cases, additional features need to be extracted from within an application to accurately predict its performance. Most existing CRC prediction techniques are domain-specific [5], [6], [7] or requiring expert knowledge [8], [9].

In this paper, we present Mantis, a new framework to predict online the CRC of bytecode programs on given inputs accurately, and efficiently. 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 [10]. 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.

It is tempting to exploit features that are evaluated at late stages of program execution, as such features may be strongly correlated with CRC. A drawback of naïvely using such features for predicting program CRC, 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 CRC, but to also evaluate 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* [11], [12]. 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 standalone executable programs; thus, executing them on program inputs provides both the evaluation time and the value of the corresponding feature.

We have implemented Mantis for Android applications and applied it to six CPU-intensive applications (encryptor, path routing, spam filter, chess engine, ringtone

- *Y. Kwon, H. Yi, D. Kwon, S. Yang, B.-G. Chun, and Y. Paek are with the Department of Electrical and Computer Engineering, Seoul National University, Kwanak-gu Kwanak-ro 1, Seoul. E-mail: {yikwon, hyyi, dhkwon, sjyang, bgchun, ypaek}@sor.snu.ac.kr.*
- *S. Lee is with the Department of Computer Science, University of Texas at Austin, TX 78712 USA. E-mail: sangmin@cs.utexas.edu.*
- *L. Huang and P. Maniatis are with the Research Department, Intel Labs, 2150 Shattuck, Suite 1300, Berkeley, CA 94704 USA. E-mail: {ling.huang,petros.maniatis}@intel.com.*
- *M. Naik is with the the School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332 USA. E-mail: naik@cc.gatech.edu.*

Manuscript received 7 June 2014; revised 22 Sept. 2014; accepted 18 Nov. 2014. Date of publication 17 Dec. 2014; date of current version 31 Aug. 2015. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TMC.2014.2374153

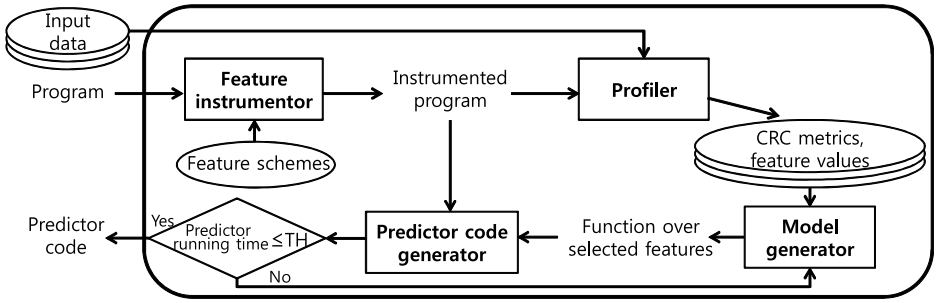


Fig. 1. The Mantis offline stage.

maker, and face detection), an I/O-intensive application (JTar) and a network-intensive application (SorTube) on four smartphone hardware platforms (Galaxy Nexus, Galaxy S2, Galaxy S3, Nexus 5). We demonstrate experimentally that, with Galaxy Nexus, Mantis can predict the execution time, the energy consumption, the accumulated memory allocation, the memory requirement and network usage of these programs with estimation error mostly under 5 percent, with a few exceptions peaking at 11.1 percent, by executing slices that spend at most 1.3 percent of the total execution time of these programs. The results for Galaxy S2 and Galaxy S3 are similar. We also briefly show the impact Mantis could have on mobile execution offloading.

We summarize the key contributions of our work:

- We propose a novel framework that automatically generates CRC 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.
- We show that Mantis can predict other CRC metrics, energy consumption, accumulated memory allocation and memory requirement, with only implementation of its own CRC metric profiler.
- We show an example of Mantis predictors enhancing the performance of smartphones.

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 CRC-model generation, respectively. Section 5 describes predictor code generation using program slicing. In Section 6 we present our system implementation and in Section 7 we present evaluation results. Finally, we discuss related work in Section 8 and conclude in Section 9.

2 Architecture

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

- What are good program features for CRC prediction? Among many features, which ones are relevant to CRC metrics? How do we model CRC 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 Fig. 1, consists of four components: a feature instrumentor, a profiler, a CRC-model generator, and a predictor code generator.

The feature instrumentor (Section 3), takes as input the program whose CRC 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 CRC metrics. 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 $(f_1, \ldots, f_M)$ as per the schemes.

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

The CRC-model generator (Section 4) performs sparse nonlinear regression on the feature values and CRC metrics obtained by the profiler, and produces a function (λ) that approximates the program's CRC metrics using a subset of features $(f_{i1}, \ldots, f_{iK})$. 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 CRC metrics, 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 CRC on given inputs, we need a way to efficiently evaluate each of the chosen features $(f_{i1}, \ldots, f_{iK})$ 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 ($f_{i1}, \ldots, f_{iK}$) is expensive, then via the *feedback loop* in Fig. 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 CRC, along with a feature evaluator for the chosen features.

The online stage is straightforward: it takes a program input from which the program's CRC 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 CRC 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.

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. Similarly, variable value features can be equivalent to other types of features but significantly cheaper to compute.

4 CRC 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 CRC 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 CRC 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 ith observation of execution time, and $\mathbf{v}_i$ denotes the ith 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 [13]. 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 [13]. 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 least absolute shrinkage and selection operator (LASSO) [14] and FoBa [15], 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 [16], [17]. 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 [15].

For our system, we chose the SPORE-FoBa algorithm, which we proposed [10], 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 $1, x_1, x_2, x_1^2, x_1 x_2, x_2^2$, 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 CRC 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 [18], 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 CRC model generator is intended to efficiently predict the program's CRC 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* [11], [12] 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 CRC 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: 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 example illustrates a case 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 [19], [20].

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 [21].

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. [22]). 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 (SDG). 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) [20], 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 [20], 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.

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

Augmenting System Dependence Graph. This step uses the algorithm by Reps, Horwitz, Sagiv, and Rosay [19] 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 [20] 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 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 Implementations

We have built a prototype of Mantis implementing the instrumentor, profiler, model generator and predictor code generator (Fig. 2). The prototype is built to work with Android application binaries. We implemented the feature instrumentor using Javassist [23], which is a Java bytecode rewriting library. The profiler is made of scripts automatically running the program for CRC metric data and the instrumented program for feature data on the test inputs. After the corresponding CRC profiler has gathered the profile data, it is used by the model generator, which is written in Octave [24] scripts. Finally, we implemented our predictor code generator in Java and Datalog by extending JChord [25], a static and dynamic Java program-analysis tool. JChord uses the Joeq Java compiler framework to convert the bytecode of the input Java program into a

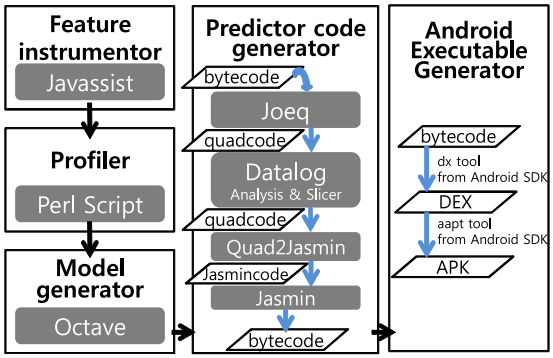


Fig. 2. Mantis prototype toolchain.

three-address-like intermediate code called quadcode, which is more suitable for analysis. The predictor code generator produces the quadcode slice, which is the smallest subprogram that could obtain the selected features. Each quad instruction is translated to a corresponding set of Jasmin [26] 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.

To show Mantis can work with various CRC metrics, we chose five CRC metrics to implement, execution time, energy consumption, accumulated memory allocation, memory requirement and network usage. Execution time is the amount of time needed for an application to run and energy consumption shows how much energy will be consumed when a program is executed. Accumulated memory allocation is the total sum of memory allocation during an application's runtime and memory requirement represents the absolute minimum amount of free memory needed to run an application. Finally, network usage shows how much data will be transferred through networks. As each CRC metric needs to be measured differently, we have implemented profilers for each metric.

7 Evaluation

7.1 Evaluation Environment

We mainly use for our experiments, Galaxy Nexus running Android 4.1.2 with dual-core 1.2 GHz CPU and 1GB RAM. We run our experiments with a server to run the instrumentor, model generator, and predictor code generator, as well as a smartphone to run the codes for profiling and generated predictor codes for slicing evaluation. The server runs Ubuntu 11.10 64-bit with a 3.1 GHz quad-core CPU, and 8 GB of 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, face detection, tar archive and SorTube—cover a broad range of CPU, I/O and network intensive functionalities found in most Android-applications. Their execution times are sensitive to inputs, challenging to model. Below we describe the applications and the input set we used for experiments in detail.

We evaluate Mantis on randomly generated inputs for each application. These inputs achieve 95-100 percent basic-block coverage, except exception handling. We generate 1,000 random test inputs with their random range specified for each application. The predictor for each application is trained on 100 random inputs that fall within 60 percent of the specified random range. For each platform, we run Mantis to generate predictors and measure their error and running time. The threshold is set to 5 percent, which means a generated predictor is accepted only if the predictor running time is less than 5 percent 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 to 8,000 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 applications). 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 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 AI part of a chess application. Similar to many game applications, it receives the configuration of chess pieces as input and determines the best move using a 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 $900 \times 3,000$ pixels.
- *Tar Archive*. This application compresses text files to tar archive files by using JTar. JTar is a simple Java Tar library using IO streams. We use randomly generated text files, ranging from a few KBs to dozens of KBs, and randomly choose 1,000 sets of files for compression.

TABLE 1
Prediction Error and Prediction Time for CRC Predictions

	Execution Time			Energy Consumption			Memory Allocation			Memory Requirement		
Application	Error	T_{Ave}	T_{Max}	Error	T_{Ave}	T_{Max}	Error	T_{Ave}	T_{Max}	Error	T_{Ave}	T_{Max}
Encryptor	4.5%	0.18%	0.26 s	3.9%	0.18%	0.26 s	0.0%	0.18%	0.26 s	0.2%	0.00%	0 s
Path Routing	5.4%	1.34%	0.29 s	4.8%	1.34%	0.29 s	4.9%	1.34%	0.29 s	0.5%	1.34%	0.29 s
Spam Filter	3.1%	0.51%	0.35 s	2.8%	0.51%	0.35 s	0.2%	0.51%	0.35 s	2.8%	0.51%	0.35 s
Chess Engine	11.1%	1.03%	0.46 s	10.5%	1.03%	0.46 s	8.1%	1.03%	0.46 s	4.6%	0.73%	0.34 s
Ringtone Maker	4.9%	0.20%	0.24 s	4.3%	0.20%	0.24 s	0.0%	0.00%	0 s	0.1%	0.00%	0 s
Face Detection	3.8%	0.62%	0.27 s	3.6%	0.62%	0.27 s	0.1%	0.62%	0.27 s	0.6%	0.62%	0.27 s
Tar Archive	3.4%	1.24%	0.41 s	4.1%	1.24%	0.41 s	1.5%	0.15%	0.64 s	0.2%	0.00%	0 s

- *SorTube*. This is a cross-platform media streaming application with video customization for heterogeneous devices. Its server-side application pulls video data from a media server and adjusts the video format, bit rate, resolution and pixel format to match the specification of the device streaming the video. We use 1,000 videos which have different durations and sizes.

7.2 Experiment Results

7.2.1 Accurate and Efficient Prediction for Execution Time

We first evaluate the accuracy and efficiency of Mantis execution time prediction. "Execution Time" column in Table 1 reports the prediction error and running time of Mantis-generated execution time predictors. 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 "T_{Ave}" 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. The "T_{Max}" shows the actual running time of the predictor. We show the longest prediction time among the 1,000 inputs.

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

Mantis generated interpretable and intuitive prediction models by only choosing one or two among the many detected features unlike non-parametric methods. Table 2 shows the total number of features initially detected, the number of features actually chosen to build the prediction model, the selected features and the generated polynomial prediction model of execution time. 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 percent of the original application time. As the expensive feature is not usable, Mantis chooses alternative features: the number of nodes in the first 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 percent.

7.2.2 Accurate and Efficient Prediction for Energy Consumption

"Energy Consumption" column in Table 1 shows the prediction error and running time of Mantis-generated energy consumption predictors. As the table shows, the generated predictors obtained error rates under 5 percent with the exception of the predictor for chess engine and all of them

TABLE 2
The Total Number of Features Initially Detected, the Number of Chosen Features, Selected Features and Generated Prediction Models for Execution Time Prediction

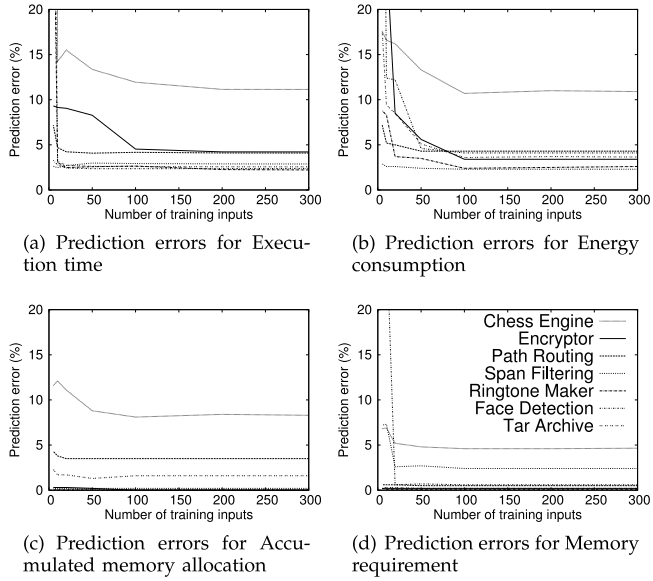
	No. of features			
Application	Total	Chosen	Selected features	Generated model
Encryptor	28	2	Matrix-key size (f_1), Loop count of encryption (f_2)	$c_0 {f_1}^2 f_2 + c_1 {f_1}^2 + c_2 f_2 + c_3$
Path Routing	68	1	Build map loop count (f_1)	$c_0 {f_1}^2 + c_1 f_1 + c_2$
Spam Filter	55	1	Inner loop count of sorting (f_1)	$c_0 f_1 + c_1$
Chess Engine	1,084	2	# of 1-depth nodes (f_1), # of pieces (f_2)	$c_0 {f_1}^3 + c_1 f_1 f_2 + c_2 {f_2}^2 + c_3$
Ringtone Maker	74	1	Cut interval length (f_1)	$c_0 f_1 + c_1$
Face Detection	107	2	Width of image (f_1), Height of image (f_2)	$c_0 f_1 f_2 + c_1 {f_2}^2 + c_2$
Tar Archive	122	1	Total sum of the size of the text files	$c_0 + c_1 f_1$

need only around 1 percent of the application's actual running time. The reason for the higher, yet quite acceptable, error rate on chess engine is the same with the case of the execution time predictor. The number of detected features are the same as Table 2 and the chosen features are the same as well. The generated models, however, are slightly different from the table. This is because there is usually a strong correlation between execution time and energy consumption, so the features that would affect execution time are likely to affect the energy consumption of an application as well, and the applications we tested all showed this correlation. And as the chosen features are the same, the prediction time, the time to acquire the feature values, is the same as well.

7.2.3 Accurate and Efficient Prediction for Memory Usage

"Memory Allocation" column in Table 1 shows the behaviors of the accumulated memory allocation predictors. As we can see, the overall error rate is lower, mostly under 4 percent and chess engine at 8.1 percent, than that of the execution time or energy consumption predictors. This is because the execution time and energy consumption of an application can slightly differ even on the same input and under the same environment, resulting in giving slightly different execution time or energy consumption data when profiled. On the contrary, memory allocation always gives the same profiled value on the same input, thus, Mantis is able to generate a more accurate prediction model.

The prediction time for the accumulated memory allocation predictors are the same as or less than that of the former two predictors, the ones for execution time and energy consumption, depending on the chosen features. When the chosen features are the same as the other predictors, the prediction time is the same. However the predictor for ringtone maker chose fewer features than before, and in the case of Tar Archive a different feature was chosen, both resulting in different prediction times compared to those of the predictors for the former two CRC metrics. Tar Archive's predictor for memory usage chose a different feature from what its predictor for execution time and energy consumption choose. The newly chosen feature was the number of text files to compress and as it is a cheaper feature than the total sum of text files, which was used in the other predictors, the prediction time is less than that of the other predictors. The reason why only the number of text files is relevant to memory allocation is because Tar Archive makes an object for each file and there is a fixed sized buffer for each object, which means the memory used for each file is always constant. In the case of ringtone maker the prediction time is 0. The reason for this is that the generated prediction model for accumulated memory allocation only has a constant term, so running a sliced version of the program to obtain feature values is unneeded. This is due to the fact that ringtone maker uses a fixed sized buffer to handle its source file and output file regardless of its input, which in turn means it always uses a fixed amount of memory. As shown in 'memory requirement" column in Table 1, the predictors generated for memory requirement prediction show similar tendencies to those of the other predictors. Most of



(a) Prediction errors for Execution time

(b) Prediction errors for Energy consumption

(c) Prediction errors for Accumulated memory allocation

(d) Prediction errors for Memory requirement

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

the predictors for memory requirement achieved low error rates with the exception of chess engine, yet even in that case the error rate is under 5 percent. This is due to the same reasons as stated before, which is that the usage of memory does not fluctuate as much as the execution time or energy consumption of an application, which in turn leads to a more accurate predictor generated by Mantis. So even when it is hard to predict the tendency of chess engine the prediction error for memory requirement is lower than that for execution time or energy consumption. The overhead to run the memory requirement predictors are around 1 percent. Though in the cases of encryptor, ringtone maker, and tar archive the prediction times are 0. The reason for this is that there are no chosen features for these applications' memory requirement predictors, which in turn means the minimum required memory for these applications stay the same regardless of the input.

7.2.4 Accurate and Efficient Prediction for Network Usage

We have tested Mantis to predict the network usage metric of SorTube. Mantis turned out to be similarly effective on predicting network usage as it is on other metrics, with 2.5 percent prediction error while requiring 0.10 percent of the original application's running time.

7.2.5 The Effect of the Number of Training Samples

We show the effect of the number of training samples on prediction errors in Fig. 3. In most cases, the curve of their prediction error plateaus before 50 input samples for training. Furthermore, even in the cases where the error rate improves beyond using 50 input samples, 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.

TABLE 3
Prediction Error of Mantis and Mantis-Linear for Execution Time Prediction

Application	Mantis(%)	Linear(%)
Encryptor	4.5	6.6
Path Routing	5.4	10.6
Spam Filter	3.1	3.1
Chess Engine	11.1	16.2
Ringtone Maker	4.9	4.9
Face Detection	3.8	52.7
Tar Archive	3.4	3.4

Mantis-linear uses only linear terms (f_i's) for model generation.

7.2.6 Benefit of Non-Linear Terms on Prediction Accuracy

Table 3 shows the prediction error rates of the models built by Mantis and Mantis-linear for execution time. Mantis-linear uses only linear terms (f_i's) for model generation. For encryptor, path routing, chess engine, and face detection, non-linear terms improve prediction accuracy significantly since Mantis-linear does not capture the interaction between features. The other cases show the two having the same accuracy because Mantis generated linear models for the predictors of those applications.

7.2.7 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 for execution time with those of predictors built with *partial execution (PE)*. Partial execution runs the instrumented program only until the point where we obtain the chosen feature values.

Mantis reduces the prediction time significantly in most cases, as 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.

The ones that show around 100 percent for PE prediction are the worst cases for PE since the last updates of the chosen feature values occur near the end of their execution. In contrast, in the cases that show lower prediction time than Mantis, the PE predictor 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.

TABLE 4
Prediction Time of Mantis and PE for Execution Time Prediction

Application	Mantis (%)	PE (%)
Encryptor	0.18	100.08
Path Routing	1.34	17.76
Spam Filter	0.51	99.39
Chess Engine	1.03	69.63
Ringtone Maker	0.20	0.04
Face Detection	0.62	0.17
Tar Archive	1.24	0.53

TABLE 5
Prediction Error of Mantis and BE for Execution Time Prediction

Application	Mantis (%)	BE (%)
Encryptor	4.5	57.1
Path Routing	5.4	69.3
Spam Filter	3.1	39.7
Chess Engine	11.1	28.1
Ringtone Maker	4.9	4.9
Face Detection	3.8	3.8
Tar Archive	3.4	3.4

7.2.8 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 (BE)*. Bounded execution 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 models built by Mantis in most cases. This is because BE cannot exploit as many features as Mantis. As a result, in several cases, no usable feature can be obtained by BE; thus, BE creates a prediction model with only a constant term for the applications.

7.2.9 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 execution time predictor with two additional smartphones: Galaxy S2 and Galaxy S3, respectively. Galaxy S2 has a dual-core 1.2 GHz CPU and 1 GB RAM, running Android 4.0.3. Galaxy S3 has a quad-core 1.4 GHz CPU and 1 GB 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. Here, Mantis builds models similar to the ones 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. We also verify the prediction error and time for the other CRC metrics

TABLE 6
Prediction Error and Time of Execution Time Running with Galaxy S2 and Galaxy S3

	Galaxy S2		Galaxy S3	
Application	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 7
Prediction Errors for a Multi-Threaded Chess Engine Application with Nexus 5

# of Threads	Error(%)	Speed-up
Single	3.8	1X
Dual	6.0	1.37X
Quad	7.9	1.36X

with the additional smartphones are almost same as the result of execution time prediction. The result shows that Mantis generates predictors robustly with different hardware platforms.

7.2.10 Prediction Errors for Multi-Threaded Applications

We show the effect of multi-threads on prediction error in Table 7. To compare the prediction accuracy of different amount of concurrent threads in an application, we modified chess engine to run as a multi-threaded application. We run it on Nexus 5 which has a quad-core 2.3 GHz CPU and 2 GB RAM. As shown in the table, Mantis' prediction accuracy falls a little as the number of threads are increased. An interesting point here is that the application does not seem to benefit much from multi-threading above two threads, as the speed-up is about the same. This seems to be a result of the Android OS kernel limiting the usage of CPU cores to sustain the device's power.

7.2.11 Offline Stage Processing Time

Table 8 presents 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 iterations 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 up to around 13 hours because of many rejected features.

7.3 Applying Mantis in Frameworks

Input based CRC prediction on smart mobile devices has many possible applications. Users no longer need to cluelessly wait for a program to finish, systems could schedule applications more efficiently and unusual behaviors of an application might even be predicted beforehand as well.

TABLE 8
Mantis Offline Stage Processing Time for Execution Time Prediction (in Seconds)

Application	P.	M.	S.	T.	Total	Iters
Encryptor	2,373	18	117	391	2.9k	3
Path Routing	363	28	114	14	0.5k	3
Spam Filter	135	10	66	3	0.2k	2
Chess Engine	6,624	10k	6,016	23k	46k	83
Ringtone Maker	2,074	19	4,565	2	6.7k	1
Face Detection	1,437	13	6,412	179	8k	4
Tar Archive	2,718	80	125	913	3.8k	8

While there are many applications where Mantis could be beneficial, in this section we show how the Mantis predictor could enhance mobile execution offloading.

Mobile execution offloading [27] is the act of transferring the execution of a program at runtime from a smart mobile device to a resource-rich server in order to counter the limitations of smart mobile devices, such as limited battery power or limited computational resources. However, there are costs, namely time and energy costs, to transfer an execution from a device to a server. Our previous work [28] focuses on reducing such costs, yet it can still be detrimental to offload in some cases. This is why a program should only be transferred when the benefit of offloading the program outweighs the cost of doing so. Thus making performance prediction important in mobile execution offloading. The gains and costs of offloading must be predicted in order to make a good decision of whether or not to offload. At the same time, these gains and costs of offloading could change when the input of a program changes, making it important for making predictions that could reflect such changes as well.

Existing works [1], [2] somewhat address this by using profiled data from past executions to predict the outcome of the current execution in order to make a decision on offloading. This technique works well in some cases, where the current program input or device conditions are similar to previous executions, it often fails to give a usable prediction in general cases. Meanwhile, [29] exploits cloud with evidence-based learning methods to provide sophisticated decision process. To solve the problem, we use the Mantis predictor in mobile execution offloading. To simplify the offloading problem to focus on the impact of prediction results, the modified framework makes offloading decisions based on the following equation, whose goal is only to minimize the execution time of chess engine. We simplify the whole offloading problem to the most basic form in order to focus on the impact of decision making

$$Minimize\big((1-l)\times T^{l}+l\times(T^{r}+C)\big) \tag{1}$$

l will be 0 if the program should be run locally, and 1 if it should run remotely on a given input. T^{l} is the predicted execution time on the smartphone, T^{r} that on the server and C is the predicted cost of offloading. Because the states to be transferred between a mobile device and a server is constant, the value of C does not affected by input-change. We tested our own chess engine predictor generated with Mantis as well as a simplified predictor similar to those in prior works [1], [2] to acquire the predictions for the equation.

Table 9 shows the profiling and offloading results. The table shows five random cases from our test set of 1,000 inputs. The "mobile" and "server" columns are results from offline profiling and represents the execution time of chess engine being run only on the mobile device and only on the server, respectively. The "PE" and "BE" columns show the offloading framework using a PE and BE predictors, respectively. Finally, the "Mantis" column shows the offloading framework using a Mantis predictor while including the predictor's prediction time. As we can see, even with the prediction overhead, our Mantis predictor greatly enhanced

TABLE 9
The Performance of Mobile Execution Offloading with Decision Making Based on the Prediction Techniques (ms)

Input	Mobile	Server	PE	BE	Mantis
1	1834	3151	3103	1844	1852
2	7355	3883	8959	7398	3922
3	1943	3031	3315	1969	1962
4	8926	3992	10210	4024	4031
5	3314	3189	5617	3356	3347
total	23372	17246	31204	18591	15014

the execution offloading framework. The average performance increase Mantis brought to the total input set was 35.7 percent over Mobile only execution and 14.8 percent better over server only execution.

8 Related Work

Much research has been devoted to modeling system behavior as a means of prediction for databases [5], [6], cluster computing [30], [31], networking [32], [33], [34], program optimization [35], [36], mapping parallelism [37] 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 [38], [39], database query execution-time and resource prediction [5], [6], performance prediction for streaming applications based on control flow characterization [40], violations of service-level agreements (SLAs) for cloud and web services [30], [31], 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 [41] to model program complexity [8], to improve hardware simulation specificity [42], [43], and to find bugs cooperatively [44]. 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 [32], [33], [34].

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 [45], [46], [47], [48], [49] 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 [8], simulation to derive worst-case running time [50], and symbolic execution and abstract evaluation to derive either worst-case inputs for a program [51], or asymptotic bounds on worst-case complexity [52], [53]. In contrast, our goal is to automatically generate an online, accurate predictor of the performance of particular invocations of a general-purpose program.

Among many algorithms to predict CRC metrics [54], [55], [56] Mantis's machine learning algorithm for prediction is based on our earlier work [10] which is to select just a few useful features from hundreds or thousands of features detected in an application. In our prior work, we computed program features manually. In this work, we introduce program slicing to compute features cheaply and generate predictors automatically, apply our system to Android smartphone applications on multiple hardware platforms, and evaluate the benefits of slicing thoroughly.

9 Conclusion

In this paper, we presented Mantis, a framework that automatically generates program CRC predictors that can estimate CRC 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 our prototype implementation of Mantis generates good predictors that estimate five CRC metrics with high accuracy and short prediction time. Mantis can automatically generate predictors that estimate five CRC metrics accurately and efficiently for smart mobile applications. Furthermore, the evaluation shows Mantis could enhance mobile execution offloading.

The greatest weakness of Mantis is that it requires a large amount of off-line processing time. Generating a minimum input set that represents an application's dynamic behavior for profiling and using static analysis to acknowledge expensive features without actually running them for testing would potentially shorten the off-line processing time while giving a more precise predictor as well. Building a separate predictor for every device/server an application may potentially use is burdensome as well. Some techniques [57] to predict an application's performance by scaling the prediction from a reference device according to the new target device's specification would help in this matter.

Acknowledgments

This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning (MSIP) / National Research Foundation of Korea (NRF)(Grant NRF-2008-0062609), the IT R&D program of MSIP/KEIT [K10047212, Development of homomorphic encryption supporting arithmetics on ciphertexts of size less than 1kB and its applications], IDEC, the Business for Cooperative R&D between Industry, Academy, and Research Institute (Grant C0218072) funded by Korea Small and Medium Business Administration in 2014, and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2014R1A2A1A10051792). Yunheung Paek is the corresponding author.

REFERENCES

[1] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making smartphones last longer with code offload," in *Proc. 8th Int. Conf. Mobile Syst., Appl., Serv.*, 2010, pp. 49–62.

[2] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 301–314.

[3] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. (2012). Comet: Code offload by migrating execution transparently in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, pp. 93–106 [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387890

[4] W. Smith, "Prediction services for distributed computing," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Mar. 2007, pp. 1–10.

[5] C. Gupta, A. Mehta, and U. Dayal, "PQR: Predicting query execution times for autonomous workload management," in *Proc. Int. Conf. Auton. Comput.*, 2008, pp. 13–22.

[6] 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 *Proc. IEEE Int. Conf. Data Eng.*, 2009, pp. 592–603.

[7] P. Barlet-Ros, G. Iannaccone, J. Sanjuas-Cuxart, D. Amores-Lopez, and J. Sole-Pareta, "Load shedding in network monitoring applications," in *Proc. USENIX Annu. Tech. Conf.*, 2007, pp. 5:1–5:14.

[8] S. Goldsmith, A. Aiken, and D. Wilkerson, "Measuring empirical computational complexity," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 395–404.

[9] E. Brewer, "High-level optimization via automated statistical modeling," in *Proc. 5th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 1995, pp. 80–91.

[10] 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 *Proc. Conf. Neural Inf. Process. Syst.*, 2010, pp. 883–891.

[11] M. Weiser, "Program slicing," in *Proc. 5th Int. Conf. Softw. Eng.*, 1981, pp. 439–449.

[12] F. Tip, "A survey of program slicing techniques," *J. Program. Lang.*, vol. 3, no. 3, pp. 121–189, 1995.

[13] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. New York, NY, USA: Springer, 2009.

[14] R. Tibshirani, "Regression shrinkage and selection via the lasso," *J. Roy. Statist. Soc B.*, vol. 73, pp. 273–282, 1996.

[15] T. Zhang, "Adaptive forward-backward greedy algorithm for sparse learning with linear models," in *Proc. Conf. Neural Inf. Process. Syst.*, 2008, pp. 1921–1928.

[16] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least angle regression," *Ann. Statist.*, vol. 32, no. 2, pp. 407–499, 2002.

[17] 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. Select. Topics Signal Process.*, vol. 1, no. 4, pp. 606–617, Dec. 2007.

[18] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. Int. Conf. Comput. Statist.*, 2010, pp. 177–187.

[19] T. W. Reps, S. Horwitz, S. Sagiv, and G. Rosay, "Speeding up slicing," in *Proc. 2nd ACM SIGSOFT Symp. Found. Softw. Eng.*, 1994, pp. 11–20.

[20] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1988, pp. 35–46.

[21] O. Lhoták, "Program analysis using binary decision diagrams," Ph. D. dissertation, School Comput. Sci., McGill Univ., Montreal, QC, Canada, 2006.

[22] M. Sridharan, S. Fink, and R. Bodik, "Thin slicing," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2007, pp. 112–122.

[23] Javassist (2012). [Online]. Available: www.csg.is.titech.ac.jp/~chiba/javassist, product page

[24] Octave (2013). [Online]. Available: www.gnu.org/software/octave, product page

[25] JChord (2012). [Online]. Available: code.google.com/p/jchord, product page

[26] Jasmin (2004). [Online]. Available: jasmin.sourceforge.net, product page

[27] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, " A survey of computation offloading for mobile systems," *Mobile Netw. Appl.*, vol. 18, no. 1, pp. 129–140, 2013.

[28] S. Yang, D. Kwon, H. Yi, Y. Cho, Y. Kwon, and Y. Paek, "Techniques to minimize state transfer costs for dynamic execution offloading in mobile cloud computing," *IEEE Trans. Mobile Comput.*, vol. 13, no. 11, pp. 2648–2660, Nov. 2014.

[29] H. Flores and S. Srirama. (2013). Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning in *Proc. 4th ACM Workshop Mobile Cloud Comput. Serv.*, pp. 9–16. [Online]. Available: http://doi.acm.org/10.1145/2482981.2482984

[30] 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 *Proc. Conf. Hot Topics Cloud Comput.*, 2009, p. 12.

[31] P. Shivam, S. Babu, and J. S. Chase, "Learning application models for utility resource planning," in *Proc. IEEE 3rd Int. Conf. Auton. Comput.*, 2006, pp. 255–264.

[32] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang, "WebProphet: Automating performance prediction for web services," in *Proc. 7th USENIX Conf. Netw. Syst. Des. Implementation*, 2010, p. 10.

[33] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, "Answering what-if deployment and configuration questions with wise," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2008, pp. 99–110.

[34] 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 *Proc. IEEE Conf. Comput. Commun.*, 2009, pp. 2258–2266.

[35] K. Tian, Y. Jiang, E. Zhang, and X. Shen, "An input-centric paradigm for program dynamic optimizations," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2010, pp. 125–139.

[36] 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 *Proc. 8th Annu. IEEE/ACM Int. Symp. Code Generation Optimization*, 2010, pp. 248–256.

[37] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," *SIGPLAN Not.*, vol. 44, no. 4, pp. 75–84, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504189

[38] K. Vaswani, M. Thazhuthaveetil, Y. Srikant, and P. Joseph, "Microarchitecture sensitive empirical models for compiler optimizations," in *Proc. Int. Symp. Code Generation Optimization*, 2007, pp. 131–143.

[39] B. Lee and D. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proc. 12th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2006, pp. 185–194.

[40] F. Aleen, M. Sharif, and S. Pande, "Input-driven dynamic execution behavior prediction of streaming applications," in *Proc. 15th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2010, pp. 315–324.

[41] Y. Jiang, E. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao. (2010). Exploiting statistical correlations for proactive prediction of program behaviors in *Proc. 8th Annu. IEEE/ACM Int. Symp. Code Generation Optim.*, pp. 248–256. [Online]. Available: http://doi.acm.org/10.1145/1772954.1772989

[42] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2001, pp. 3–14.

[43] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. 10th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2002, pp. 45–57.

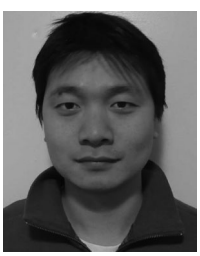
[44] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 15–26.

[45] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," in *Proc. IEEE 27th Int. Real-Time Systems Symp.*, 2006, pp. 57–66.

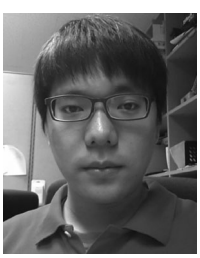
[46] Y.-T. S. Li and S. Malik, *Performance Analysis of Real-Time Embedded Software*. Norwell, MA, USA: Kluwer, 1999.

[47] R. Wilhelm, "Determining bounds on execution times," in *Handbook on Embedded Systems*, Boca Raton, FL, USA: CRC Press, 2005.
[48] S. Seshia and A. Rakhlin, "Game-theoretic timing analysis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2008, pp. 575–582.
[49] S. Seshia and A. Rakhli, "Quantitative analysis of systems using game-theoretic learning," *ACM Trans. Embedded Comput. Syst.*, vol. 11, pp. 55:1–55:27, 2010.
[50] R. Rugina and K. E. Schauser, "Predicting the running times of parallel programs by simulation," in *Proc. 1st Merged Int. Parallel Process. Symp./Symp. Parallel Distrib. Process.*, 1998, pp. 654–660.
[51] J. Burnim, S. Juvekar, and K. Sen, "WISE: Automated test generation for worst-case complexity," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 463–473.
[52] B. Gulavani and S. Gulwani, "A numerical abstract domain based on expression abstraction and max operator with application in timing analysis," in *Proc. 20th Int. Conf. Comput. Aided Verification*, 2008, pp. 370–384.
[53] S. Gulwani, K. Mehra, and T. Chilimbi, "SPEED: Precise and efficient static estimation of program computational complexity," in *Proc. 36th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 2009, pp. 127–139.
[54] A. Matsunaga and J. A. B. Fortes, "On the use of machine learning to predict the time and resources consumed by applications," in *Proc. 10th IEEE/ACM Int. Conf. Cluster, Cloud Grid Comput.*, 2010, pp. 495–504.
[55] H. Leather, E. Bonilla, and M. O'Boyle, "Automatic feature generation for machine learning based optimizing compilation," in *Proc. 7th Annu. IEEE/ACM Int. Symp. Code Generation Optimization*, Mar. 2009, pp. 81–91.
[56] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2009, pp. 45–55.
[57] J. Kim, J. Kim, J. Kim, S. Yang, Y. Cho, Y. Kwon, Y. Paek, and D. Chae, "CMcloud: Cloud platform for cost-effective offloading of mobile applications," in *Proc. 14th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2014, pp. 434–444.

Yongin Kwon received the BSc degree in electrical and electronic engineering from the KAIST, Korea, in 2008. He received the MSc degree and is currently working toward the PhD degree in electrical and computer engineering from the Seoul National University, Korea. His research interests include mobile cloud computing and embedded system.



Sangmin Lee is working toward the PhD degree in the Computer Science Department, University of Texas at Austin. His research spans a broad range of systems including privacy-preserving app platforms, distributed systems, cloud computing, and virtualization.

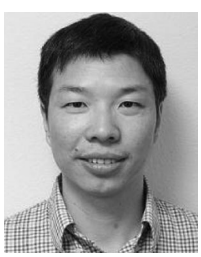


Hayoon Yi received the BSc degree in electrical and computing engineering from the Seoul National University, Korea, in 2012. He is currently working toward the PhD degree in electrical and computing engineering from the Seoul National University, Korea. His research interests include mobile cloud computing and compiler.

Donghyun Kwon received the BSc degree in electrical and computing engineering from the Seoul National University, Korea, in 2012. He is currently working toward the PhD degree in electrical and computing engineering from the Seoul National University, Korea. His research interests include mobile cloud computing and compiler.

Seungjun Yang received the BSc degree in electrical engineering from Seoul National University, Korea, in 2008. He is currently working toward the PhD degree in electrical and computer engineering from Seoul National University, Korea. His research interests include mobile cloud computing, compiler, and system engineering. He is a student member of the IEEE.

Byung-gon Chun received the BSc and MSc degrees in electronic engineering from Seoul National University, in 1994 and 1996, respectively. He received the another MSc degree in computer science from Stanford University in 2002 and the PhD degree from the University of California, Berkeley, in 2007. He is currently a professor in the Department of Electrical and Computing Engineering, Seoul National University, Korea.



Ling Huang received the BS and MS degrees from the Beijing University of Aeronautics and Astroautics (BUAA) in China. He received the PhD degree in computer science from the University of California at Berkeley. During the PhD study, he was affiliated with RadLab. He is currently a research scientist at Intel Labs.

Petros Maniatis received the BSc degree with honors from the Department of Informatics, University of Athens in Greece and the MSc and PhD degrees from the Computer Science Department, Stanford University. He is a senior research scientist at Intel Labs. His research interests lie primarily in the confluence of distributed systems, security, and fault tolerance.

Mayur Naik received the BE degree from BITS Pilani, India (1995-1999), the MS degree from Purdue University (2001-2003), and the PhD degree from Stanford University (2003-2007), all in computer science. He is currently an assistant professor of computer science at Georgia Tech. He did research in the areas of programming languages and software engineering. Previously, he was a researcher at Intel Research in Berkeley (2008-2011).

Yunheung Paek received the BSc and MSc degrees in computer engineering from Seoul National University, Korea, in 1988 and 1990, respectively. He received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1997. He is currently a professor in the Department of Electrical and Computing Engineering, Seoul National University, Korea. His research interests include mobile cloud computing, embedded security systems and retargetable compiler. He is a member of the IEEE.