ARTICLE IN PRESS

Pervasive and Mobile Computing ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

Precise execution offloading for applications with dynamic behavior in mobile cloud computing

Yongin Kwon, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Yeongpil Cho, Yunheung Paek *

Seoul National University, 1 Gwanak street, Seoul, Republic of Korea

ARTICLE INFO

Article history:
Received 29 September 2014
Received in revised form 4 September 2015
Accepted 1 October 2015
Available online xxxx

Keywords:
Mobile cloud computing
Execution offloading
Performance prediction

ABSTRACT

In order to accommodate the high demand for performance in smartphones, mobile cloud computing techniques, which aim to enhance a smartphone's performance through utilizing powerful cloud servers, were suggested. Among such techniques, execution offloading, which migrates a thread between a mobile device and a server, is often employed. In such execution offloading techniques, it is typical to dynamically decide what code part is to be offloaded through decision making algorithms. In order to achieve optimal offloading performance, however, the gain and cost of offloading must be predicted accurately for such algorithms. Previous works did not try hard to do this because it is usually expensive to make an accurate prediction. Thus in this paper, we introduce novel techniques to automatically generate accurate and efficient method-wise performance predictors for mobile applications and empirically show they enhance the performance of offloading.

1. Introduction

Smartphones have become an essential part of a modern man's life, with around a billion devices activated worldwide for the Android platform alone. With its wide range of functions, such as GPS or cameras, and general purpose processors with gigabytes of storage, it has become natural to deploy more and more complex applications on smartphones. These applications, however, require a considerable amount of energy and computational power. As a result, users have to match the increasing computational complexity of applications with newer hardware. Yet they still suffer from limited battery lifetime all the same.

Mobile cloud computing, which utilizes cloud alongside mobile devices, is a promising approach to alleviate this problem. Within a mobile cloud computing framework, mobile devices do not need powerful hardware because most of the complicated computations are handled in the cloud. This approach extends battery lifetime, enables the use of the computation power of cloud systems, which typically exceeds even the newest mobile hardware, and lessens the need to upgrade user's devices. In recent years, techniques called *mobile execution offloading*, which is the act of transferring execution between smartphones and servers during run time, were proposed as a way of implementing mobile cloud computing. When an execution of a program thread on the smartphone gets to a certain point in its code, the thread is suspended and its current state for execution is packaged and shipped to a server. There, the thread is reconstructed from the shipped state and is resumed until it reaches the point to return, where it packages and transfers its state back to the smartphone. Finally, the original thread is updated by these states and is resumed.

* Corresponding author. Tel.: +82 28801748; fax: +82 28715974.
E-mail address: ypaek@sor.snu.ac.kr (Y. Paek).

http://dx.doi.org/10.1016/j.pmcj.2015.10.001
1574-1192/

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

In ideal cases where the costs for state transfer and update can be neglected, any code region except for those using device resources like GPS or screens would benefit from remote execution. This is obvious because the server processor speed is much faster, and virtually no energy of the mobile device would be consumed while the thread runs on the server. In reality, however, the costs for state capturing and transferring may not be negligible and might even be a dominant factor that inhibits the regions from executing remotely. To mitigate the transfer cost, Yang et al. [1] dramatically reduced the size of transferred state by finding only the essential state needed to recreate a program on the server. Even with such efforts, however, the state transfer cost can still be high and inconstant in some cases, so offloading frameworks needed a way to selectively offload only when the code regions would benefit from the offloading.

It is for this reason that most mobile execution offloading frameworks implement a *dynamic code partitioning* module, which is also called the *solver*. The solver's key task is to determine which part of the program should be offloaded to the remote server for better performance by weighing the performance gains against the costs from the action of offloading at a certain point in the program. To accurately compare the gains with the costs, the solver should have an ability of predicting the program performance as precisely as possible before actual offloading is made. There have been several studies to build such solvers for their offloading frameworks. In most of the studies, they use the *history-based* prediction approach where they utilize the past profiled information as a basis for performance prediction of future runs [2–5]. For example in CloneCloud [2], they statically profile past information to make a set of decisions, called *scenarios*, which describe what code regions are to be offloaded at which runtime network condition (3G or WiFi). However, they have no regard for effects of inputs on program performance. In MAUI [3], they use the dynamically profiled information of a method as the predictor of future invocations. The history-based approach basically assumes that the program performance will be consistent regardless of the program input and environment. This assumption may hold for many applications, as empirically demonstrated by [3], in practice. However we have also found many other applications to which this does not apply because their performance is very *sensitive* to input values, that is, varying dynamically depending on the values.

To overcome the input-sensitivity problem of performance, in this work, we propose an alternative performance approach for execution offloading, which we call *feature-based* prediction. In this approach, we utilize as a basis for the prediction, the *features*, each of which characterizes a certain dynamic behavior of a program on a given set of input values. For example, the loop count can be a feature which represents how a loop behaves under the current input condition at runtime. To predict the program performance, we first go through the *off-line* phase where before actual program execution, we profile the execution behavior to establish a model which consists of a set of features that characterize a general behavior of the whole program execution. Then, in the *on-line* phase, we execute the program with real inputs, and whenever we need performance numbers of a specific region of the program for offloading, we extract the feature values for the current execution, which are in turn used to compute the output of the model, consequently representing the predicted performance of that part of the program.

We have implemented a feature-based prediction module for Android applications. To predict the program performance of input-sensitive applications, during the offline stage of the module, we execute an instrumented version of the applications on a set of training inputs which represent dynamic behaviors of them. The instrumented applications produce training outputs which include objective metrics of the program performance and the values of all possible features for each of the inputs. By using a machine learning technique [6] with the training output set, we select just a few among all possible features and build the model which is a function of (nonlinear combinations of) the features. Then, the prediction module provides our solver with a feature extractor and a model calculator to compute the feature values and the output of the model during the online stage. Finally, the solver makes an offloading decision with the output to improve the program performance. We show the impact of our work with three real applications, Chess Engine, Face Detection and Invaders. In our tests, we were able to reduce the execution time by up to 31.7% compared to previous methods. We also applied the prediction technique to an energy consumption problem. As a result, we could save the energy of smartphone by up to 57.2%. The rest of this paper is organized as follows: in Section 2, we first explain the basic concepts of execution offloading and then show how much impact prediction accuracy and global optimization have on offloading. Then in Section 3, we describe our techniques to precisely and efficiently predict various aspects of program performance for execution offloading. In Section 4, we describe our solver which utilizes our prediction techniques to offload our mobile code more precisely at runtime, attaining better performance of program execution. In Section 5, we experimentally demonstrate the effectiveness of our techniques which reduce significantly execution time or energy consumption. Finally, in Sections 6 and 7, we relate our work with others and conclude.

2. Background & motivation

In this section, we address how mobile execution offloading works and discuss how performance prediction accuracy and global optimization affect offloading precision.

2.1. Background

In order to evaluate the impact of prediction accuracy on the offloading performance, we present a simple offloading framework depicted in Fig. 1. The first step of the execution offloading is identifying which methods are remotely executable. There are two ways to identify the methods. The first is to use annotations within the source code to distinguish these

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

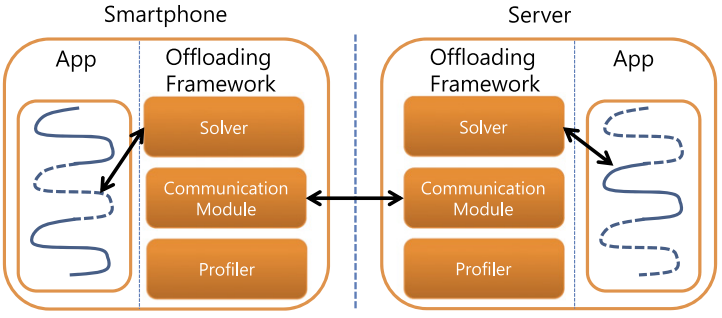

Fig. 1. A simple architecture for mobile execution offloading.

remotely executable methods (*REMs*) from the non-REMs. The second is to use static analysis to automatically identify legal choices for placing migration in the code. After identifying the REMs, when an REM is called during an execution of a program on the smartphone, the framework suspends the running thread and decides whether to offload it or not by calling the solver with the current program states and device conditions and weighing the performance gains of offloading against the costs. For example, when a user prefers fast execution, the solver predicts time for capturing and transferring the state, costs, as well as the local and remote execution time to calculate a gain. Then, the solver decides to offload the method only when the gain is bigger than the costs. If the solver decides to offload a method, the communication module of the smartphone is called to pack the program state and send the package to the server. The server-side communication module receives the states and unpacks the states to recreate the runtime environment and resume the execution of the method. When the method reaches its end, the module collects and sends only the states that are different from the original states received from the device, thereby reducing the amount of data needed to be sent back to the smartphone. The communication module of the smartphone receives the different states and compares them with the original states and applies the difference to its current program state. The program, finally, is resumed where the migrated method ended. Whenever an REM is executed, regardless of it being offloaded, the profiler measures the performance of the method. In offloading cases, it also calculates the size of the transferred state and the cost required to transfer the state.

2.2. Motivation

2.2.1. Impact of prediction accuracy on mobile offloading performance

We have implemented the simple mobile offloading framework, which uses annotation within the source code to identify REMs and offloads only a single REM at once; it means that the remote execution started at the entry of any REM should be finished at the exit of the same REM. The architecture lets users choose which prediction technique to apply to the solver. We prepare four types of prediction schemes: Standalone, Remote Only Execution (*ROE*), History-based prediction and *Oracle*. The predictions of Standalone and ROE respectively make the solver decide to always run REMs locally or to always run them remotely. History-based prediction predicts the execution time of a method to be the same as the last invocation of the method. Oracle is a prediction method that always makes the optimal decision because we assume that it knows what the actual execution time will be. For simple comparison, we ignore overhead of the prediction and assume the network is stable.

As the decision of offloading a method is based on the prediction result of the gains and the costs, the accuracy of the prediction has a great impact on the precision of the offloading. In order to demonstrate the impact of prediction accuracy, we run a program solving an N-Queen problem, which has dynamic behavior depending on its inputs, on our simple offloading framework with prediction schemes of Standalone and ROE. Fig. 2 presents the execution time of the N-Queen problem running locally and being offloaded to a server, for inputs ranging from 1 queen to 12 queens. The execution time for offloading cases includes time for capturing and transferring the states. In this figure, we can see that the execution time for running the program locally increases more dramatically than that for being offloaded.

Based on the experimental results in Fig. 2, we compared the two other prediction schemes. In Table 1, we show the solving results of running the program through the inputs in order with Oracle and History-based prediction. They decide where to run the method based on the profiling results of "Standalone" and "ROE". In the columns of "Decision of Oracle" and "Decision of HB", "L" denotes the solver decides to run the method on the smartphone and "R" denotes the solver decides to run the method on the server. We present the wrong decisions with gray-color-box. In this table, while Oracle always makes a correct decision, History-based prediction makes a wrong decision only once when the input value is 11. In Fig. 3(a), we show the sum of the execution time with each prediction approach. It shows the execution time of ROE being slightly longer than that of Standalone. However, the execution time of Oracle is about one third of those and History-based prediction comes close to Oracle.

Next, we show the solving results of the program running through a randomly ordered set of the same inputs in Table 2. In the table, History-based prediction often makes wrong decisions. As seen in Fig. 3(b), the mixed input order greatly degraded the performance of History-based prediction. History-based prediction works fine for inputs that change gradually because

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

Table 1
Solving results for ordered inputs with History-based prediction.

Input	Standalone(s)	ROE(s)	Decision of Oracle	Decision of HB
1	0	8.50	L	L
2	0	8.50	L	L
3	0	8.50	L	L
4	0.01	8.52	L	L
5	0.04	8.52	L	L
6	0.02	8.50	L	L
7	0.14	8.57	L	L
8	0.37	8.80	L	L
9	1.29	8.80	L	L
10	3.16	9.15	L	L
11	15.53	9.86	R	L
12	82.89	14.01	R	R

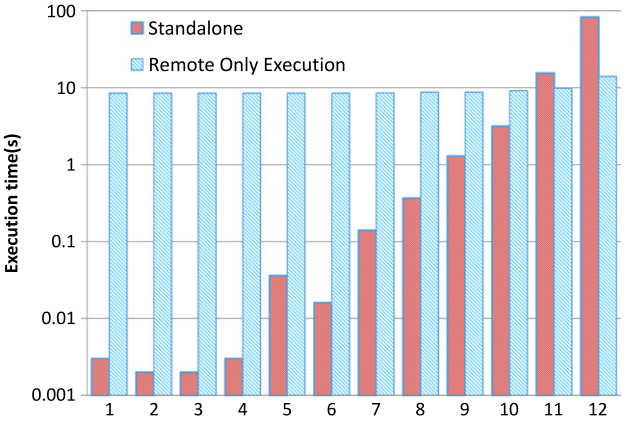

Fig. 2. Prediction errors varying the number of input samples.

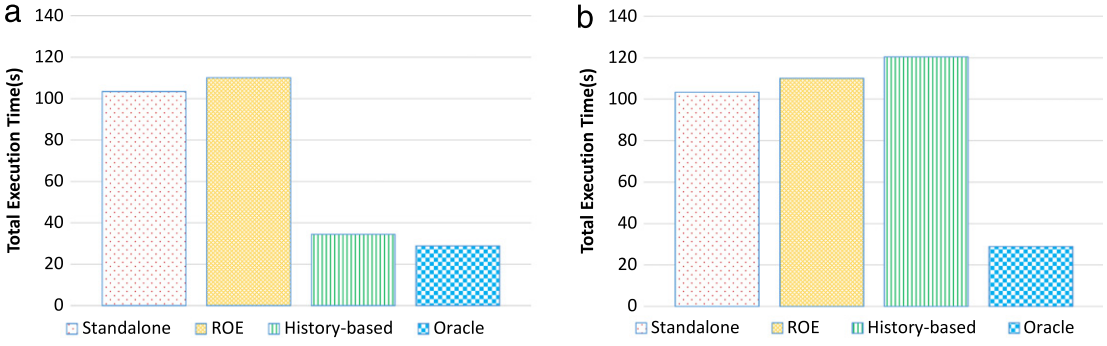

Fig. 3. Total execution time of the N-Queens program running for 4 types of decision making with (a) ordered input set and (b) randomly mixed input set.

the performance follows the gradual change, making its accuracy reliable. However, in cases where the input fluctuates, History-based prediction loses its functionality as a prediction technique, and could even lead to worse results than decisions made without any prediction at all. In the program for N-Queens problem, by the way, the input, which is the number of queens, by itself is the feature which characterizes the program's dynamic behavior. From the profiling results, we can easily find out that the gains are bigger than the costs on input values over 10. However the existing solvers do not have ability to automatically choose the feature and predict the gains and the costs with the feature. Furthermore, for real android applications, features are scattered and obscured in the whole program code. Therefore, it is needed to automate the process for feature-based prediction.

2.2.2. Global optimization for code partitioning

The solvers of previous works [2,3,5] used data collected from performance profilers to solve global optimization problems rather than local optimization problems to decide which REM should execute locally and which should execute remotely. Fig. 4 presents an example of a call graph, which also shows the need of a solver for global optimization. The time

Table 2
Solving results for randomly mixed inputs with History-based prediction.

Input	Local execution(s)	Remote execution(s)	Partition of Oracle	Partition of HB
1	0	8.50	L	L
2	0	8.50	L	L
12	82.89	14.01	R	L
4	0.01	8.52	L	R
5	0.04	8.52	L	L
11	15.53	9.86	R	L
7	0.14	8.57	L	R
8	0.37	8.80	L	L
9	1.29	8.80	L	L
10	3.16	9.15	L	L
6	0.02	8.50	L	L
3	0	8.50	L	L

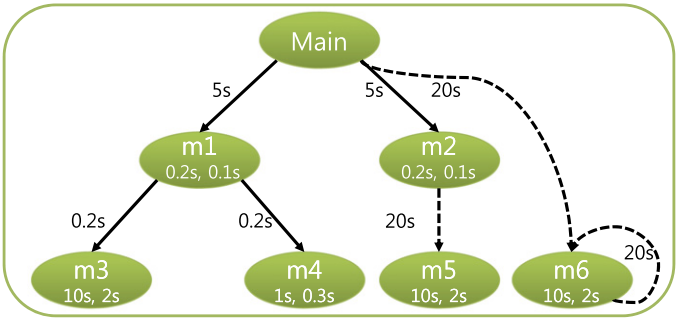

Fig. 4. An example of a call graph.

denoted on the edges represents the cost to offload the method, and the times denoted on the vertices respectively represent the execution time consumed when the method is run locally and remotely. Once a method is offloaded, no additional costs are needed to run callees remotely. To improve the performance of the program, whenever method `Main` calls method `m1`, the gain and cost on execution time are calculated and the decision where method `m1` will be run is made. In this case, it takes 11.2 s to execute the three methods `m1`, `m3` and `m4` locally. It, on the other hand, takes 7.7 s to execute the methods remotely, so a locally optimization solver decides to migrate method `m1` to the server. However, it takes only 2.9 s to execute the three methods if each of methods `m3` and `m4` is executed on remote server after method `m1` is run locally, which saves 5 s execution time-wise than migrating method m1. This result suggests that the solver should make globally optimized decisions rather than calculate gain and cost at a single method call point.

To explain in more detail the impact of prediction techniques on precision of the offloading, see the methods `m2` and `m5` in Fig. 4. The dotted edge between methods `m2` and `m5` represents that method `m5` may not be called by method `m2` depending on its state. Whether or not method `m5` will be run is a critical factor when calculating the gain or cost of migrating method `m2`. Migrating method `m2` when method `m5` will be run could be greatly beneficial. When method `m5` is not called, however, it leads to a net loss in execution time. Running method `m2` locally and deciding to migrate method `m5` if it is called is not plausible either, as the transfer cost for method `m5` is too high compared to its local execution time. Additionally, as `m6` is a recursive method, it is another critical factor for the offloading to estimate how many times the method will be executed. From this example, we can see that a prediction technique for those critical factors is needed for global optimization solver. However, to the best of our knowledge, there is no work done to consider these factors on global optimization.

3. f_Mantis : Automatical generation of accurate and efficient performance predictor for mobile execution offloading

In this section, we discuss our technique, *f_Mantis*, that helps to efficiently predict the performance of mobile applications. f_Mantis is the extension of Mantis [7], which generates a performance predictor for a mobile application off-line (i.e. before the program is run by a user). We improve and modify the techniques of Mantis in order to apply it to mobile execution offloading. The details are following.

- f_Mantis generates predictor which efficiently predicts various types of performance metrics including execution time, energy consumption, memory usage and state size.
- The predictor predicts whether or not a certain method will be executed.
- The predictor predicts not only the performance of a whole application but also that of each method of the application.
- The predictor can be run during the execution of an application.

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

3.1. Performance predictor generation overview

In Fig. 5, we show the architecture of f_Mantis. In the off-line phase, f_Mantis consists of four components: a *feature instrumentor*, a *profiler*, a *model generator* and a *predictor generator*. The feature instrumentor takes as input an application whose performance is to be predicted and instruments the application to collect the values of features as per the performance metrics and methods. Next, the profiler takes the instrumented application and a set of user-supplied program inputs. It runs the instrumented program on each of these inputs and produces a vector of features. It also measures the performance metrics on each input. Model Generator then performs sparse linear regression on the feature values and performance metrics obtained by the profiler, and produces a function that approximates the program's performance metrics using a subset of the features. As a final step, for each function, the predictor generator produces a *feature extractor* to extract chosen feature values, which are used in the predictor function just before running target method, and it also produces a *model calculator* which makes predictions for a given REM.

3.2. Feature instrumentor

The performance of an application is determined by its inputs and state. These could be the features on their own, but in most cases, other features implicated by them are more usable to characterize dynamic behavior of the application. Usable program features, which are candidates for a basis of prediction, can be found in a program in various forms. Some of these features may not be immediately visible in the original code, so we instrument the program in order to acquire them. We consider four instrumentation schemes for such features: branch counts, loop counts, method-call counts and variable values.

The scheme for branch counts 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 */ }
else {
    /* light 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`). Therefore, we instrument the code as follows.

```
if (b == true) {
    f_true ++;     // feature instrumentation for true branch
    /* heavy computation */ }
else {
    f_false ++;     // feature instrumentation for false branch
    /* light computation */ }
```

The scheme for loop count generates, for each loop occurring in the program, a feature counting the number of times that it iterates in an execution. The scheme for method call count generates a feature counting the number of calls to each procedure. Clearly, each such feature is potentially correlated with program performance. The scheme for variable value 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. 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 tend to affect program execution by changing control flow. Second, since we cannot instrument all methods (e.g., system call handlers), the values of parameters to such methods may be correlated with their program performance.

3.3. Profiler

The profiler computes feature values with test inputs by running the instrumented code. Each REM in the application has its own feature values, which are accumulated or modified until the corresponding method is finished. At the same time, performance metrics of each REM on both server and smartphone are recorded as well. Our architecture handles five metrics, which are execution time, energy consumption, memory requirement, transferring state size and method call count. The program code is instrumented to leave runtime performance information as following.

When the profiler meets an REM, the state needed to be transferred to the server is serialized to profile the size for the migration. Then, by recording the timestamps at the entry point and return point of the method, the execution time is profiled. The memory usage and the power consumption for the method are recorded as well. The method call count is profiled by the instrumented code. Our profiler outputs datasets for each target method 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 the vector of the performance metrics, and v_i denotes the ith observation of the vector of M features.

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

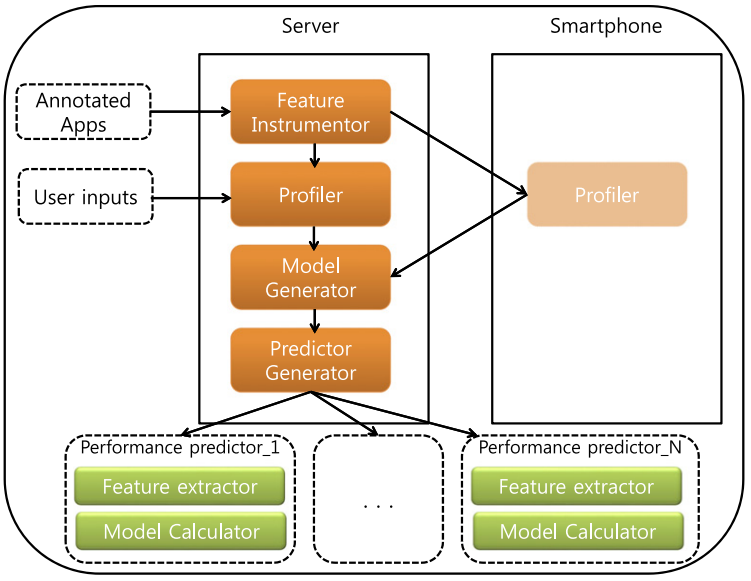

Fig. 5. Overview of f_Mantis architecture.

3.4. Model generator

The profiler produces a large number of features. Most of these features, however, are not expected to be useful for performance prediction, due to the fact that in most cases, only a handful of features are enough to make a good prediction. 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 [8]. However, least square regression tends to overfit the data and create complex models with poor interpretability.

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 [8]. In recent years a number of approximate algorithms have been proposed as efficient alternatives for simultaneous feature selection and model fitting [9,10].

To seek compact performance models, which are functions of just a few features that accurately approximate performance metrics, we chose the SPORE-FoBa algorithm [6]. 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. The function output by the performance model generator is intended to precisely predict the program methods performance metrics on given program states.

3.5. Predictor generator

In order to evaluate features that appear in the generated model function and make a prediction based on the function at runtime, the predictor generator produces a feature extractor and a model calculator for each predictor. The feature extractor and the model calculator are executable codes which extract the feature values for the function of the performance model and make predictions at runtime. To generate the feature extractor, the predictor generator draws call-graphs and control dependency graphs to analyze the dependency between features and methods. There are two cases that can occur, which are shown in the following program code. In the code, the method call `goo` in method `foo` is the only remotely executable point, and at this point, whether it should be migrated to the server or not should be decided. To make the decision, the performance of method `goo` should be predicted with features. The first case is when the chosen features appear before the call of the method needing prediction, which would be within the code segment annotated computation S1 in the program.

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

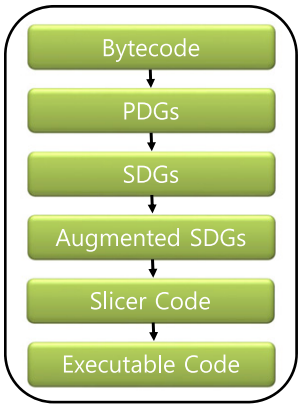

Fig. 6. Design of the slicer.

In this case, the generator instruments the original code to calculate the feature and save it to be used for the prediction. The second case is when the chosen features appear during the execution of the method needing prediction, which would be within the code segment annotated computation S2. In this case, we might have to run the method until the feature values are obtained in order to predict its performance, which could need the method to be run entirely in the worst cases. We avoid this problem by extracting the feature values from the method efficiently, using *static program slicing* [11,12].

```
foo(...) {
  ...; // computation S1
  goo(...);   //REM
  ...;
}
goo(...) {
  ...; // computation S2
}
```

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 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 the following example code.

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

Our slicer combines several existing algorithms to produce executable slices. Fig. 6 shows the design of the slicer. For each method reachable from the target method, we build a *Program Dependence Graph (PDG)* [13], whose nodes are statements in the body of the method and whose edges represent intra-procedural data/control dependencies between them. The PDGs constructed for all methods are stitched into a *System Dependence Graph (SDG),* [13], 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. We 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 [14]. Next step takes as input a slicing criterion and the augmented SDG, and produces as output the set of all statements on

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

which the slicing criterion may depend using two-pass backward reachability algorithm [13]. As a final step, we translate the slicer code to be executable.

4. Dynamic code partitioning with predictor generated by f_Mantis

In this section, we propose a new solver, which makes offloading decisions based on outputs of predictors generated by f_Mantis. The solver precisely offloads REMs in order to match the user's various needs.

4.1. Architecture for our solver

Fig. 7 presents the architecture of the solver. Before running an application, we obtain the call graph $G = (V, E)$ for the application. Each vertex $v \in V$ represents a method in the call stack of the application, each edge $e = (u, v)$ represents an invocation of method v from method u. When the program thread reaches an REM, the offloading framework relays information, such as program state or method call parameters, of the method to the solver. Within the solver, at first, an *REMs finder* draws a subgraph $G' = (V', E')$, which contains all vertices and edges that are reachable from the vertex of the REM, of graph G and finds the remotely executable vertices from the graph. For each remotely executable vertex $v \in V'$ and edge $e = (u, v) \in E'$, we predict its performance and costs as follows: the energy consumption for local execution E_v^l, the local execution time T_v^l, the remote execution time T_v^r, the energy cost for remote execution $B_{u,v}$, the time cost for remote execution $C_{u,v}$ and the times that method v will be called from method $uR_{u,v}$. For each of the predictions, a feature extractor calculates the feature values needed for the prediction. As the feature extractor shares the parameters and states with the original thread, this process might modify the program state or heap objects. In other words, it may alter the behavior of the original program after extracting the features. To avoid this, we instrument the feature extractor to backup the original state. After the prediction, a *state restorator* restores the backed up state. A model calculator then makes a prediction using the extracted feature values.

According to the user's need, a *decision making* module solves one of the following two integer linear programming problems for global optimization, which is based on the MAUI solver [3]. It solves the problem for various solving priorities, such as reducing the execution time, the energy consumption or the memory usage and preventing errors of memory. The solving result of the module is I_v which will be 0 if method v should be run locally and 1 if it should run remotely.

$$\text{Maximize} \sum_{v \in V} I_v \times E_v^l \times R_{(\_,v)} - \sum_{(u,v) \in E} |I_u - I_v| \times C_{(u,v)} \times R_{(u,v)}$$

$$\text{Such that} : \sum_{v \in V} \left((1 - I_v) \times T_v^l + \left(I_v \times T_v^r\right)\right) \times R_{(\_,v)} + \sum_{(u,v) \in E} |I_u - I_v| \times B_{(u,v)} \times R_{(u,v)} < L_t \tag{1}$$

$$\text{Minimize} \sum_{v \in V} \left((1 - I_v) \times T_v^l + \left(I_v \times T_v^r\right)\right) \times R_{(\_,v)} + \sum_{(u,v) \in E} |I_u - I_v| \times B_{(u,v)} \times R_{(u,v)}$$

$$\text{Such that} : \sum_{v \in V} I_v \times E_v^l \times R_{(\_,v)} - \sum_{(u,v) \in E} |I_u - I_v| \times C_{(u,v)} \times R_{(u,v)} > L_e. \tag{2}$$

We provide additional solving options as well. To prevent out-of-memory error, when an REM is to be called, we predict the memory usage of the method. If this predicted memory usage exceeds the available memory on the device, we execute the method remotely.

5. Evaluation

5.1. Implementation

We have built f_Mantis, implementing the feature instrumentor, profiler, model generator and predictor generator (Fig. 5). These are built to work with Android application binaries. We implemented the feature instrumentor using Javassist [15], which is a Java bytecode library. The profiler is made of scripts automatically running the program to profile for execution time, energy consumption, transfer state size and memory consumption on test inputs. To profile energy consumption, we used a Monsoon power monitor in order to obtain accurate data on energy consumption. When the profiler runs, the monsoon power monitor leaves a continuous data that contains data for all separate test runs done during the profile. The profiler cross examines this data with the execution times for each test run to obtain the energy consumption of each test run.

Also, the profiler instruments programs to extract feature data. The data obtained by the profiler is then used by the model generator, which is written in Octave [16] scripts, to build a prediction model of the given program. Based on this model, the predictor code generator generates a predictor code. We implemented our predictor code generator in Java and Datalog by extending JChord [17], a static and dynamic Java program-analysis tool. JChord converts the bytecode of the input Java program into a three-address-like intermediate code7. Then, the generator produces a slice, which is the smallest code

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

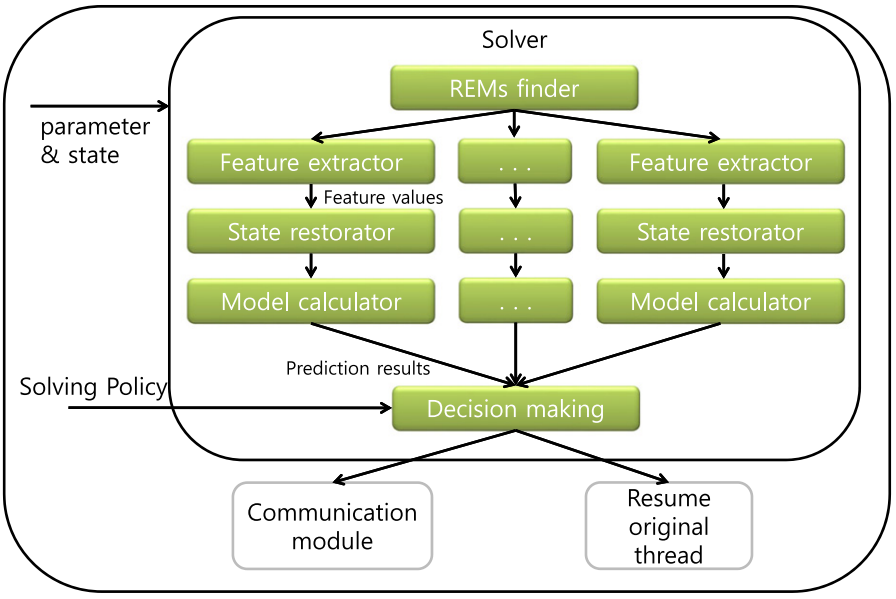

Fig. 7. Architecture for the dynamic solver.

that could obtain the selected features. The slice is translated to Jasmin [18] assembly code, and then the Jasmin compiler generates the final Java bytecode which can be executed at runtime to make a prediction.

To evaluate the performance of our predictor and dynamic partitioning process, we have built a mobile execution offloading framework, which is called *SorMob*. SorMob runs on the Android platform as an application and does mobile offloading with Aspect-Oriented programming (AOP). We used AspectJ [19], which is an AOP tool for java, to insert the profiler, solver and communication module codes at the beginning and end of a method annotated as offloadable. This is done at compile time and enables SorMob for the target application. To execute a migrated thread, its state needs to be transferred to the server side. The state is accumulated by Kryo [20], which is an open source java serializer.

5.2. Evaluation environment

We run our experiments with a machine to run the predictor generator and the offloading server, as well as a smartphone to run the codes for profiling, generated predictor code and benchmarks for offloading evaluation. The machine runs Ubuntu 11.10 64-bit with a 3.1 GHz quad-core CPU, and 8 GB of RAM. To run the offloading server, the machine runs a virtual machine for the Android environment. The smartphone is a Galaxy Nexus running Android 4.1.2 with a dual-core 1.2 Ghz CPU and 1 GB RAM. All experiments were done using Java SE 64-bit 1.6.0 _ 30.

We have chosen three real applications – Chess Engine, Face Detection, Invaders – to evaluate our offloading performance. Unlike input insensitive applications, these three applications show different behavior on different inputs. Therefore, accurate performance prediction for these applications is needed for precise execution offloading. Below we describe the applications and the input dataset we used for their evaluation in detail.

5.3. Experimental results

5.3.1. Chess engine

Chess engine is the decision making part of a chess game application. Similar to the decision making process of many game applications, it receives the configuration of chess pieces as input and determines the best move using a Minimax algorithm. The Minimax algorithm is based on utilizing a game tree, whose nodes, in this application, represent the state of the game and edges the move of a chess piece. The Chess Engine starts at the root node, which represents the current state of the game, and builds a child node for each and every possible move allowed on the current board until it reaches a predefined depth in the game tree. It then selects the best move, which is the edge that leads to the child node with the best probable outcome. This is calculated by considering all the possible outcomes of the child node, which in turn repeatedly considers its own child nodes until the inputted depth of the game tree is met. We set the game-tree depth to three or four for this experiment and used 100 randomly generated chess-piece configurations to train the performance predictor.

Fig. 8 is a part of the call graph for Chess Engine. The number in bracket means the number of times that the method is invoked for a certain input. Dotted edges represent that the callee can be invoked multiple times or may not be invoked from the caller. We tried to predict the performance of Chess Engine from the features which include the number of times that

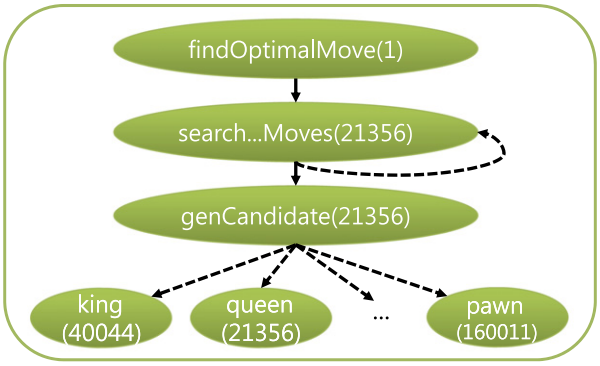

Fig. 8. A part of the call graph for Chess Engine.

Table 3
Performance prediction results for Chess Engine.

Prediction target	Prediction error (%)	Prediction time (%)	No. of chosen features	Generated model
Exec. time of dev.	15.18	0.03	3	$c_1{f_1}^{13}{f_3}^2 + c_2{f_1}^{11} + c_3{f_1}^{14}{f_2}^4$
Exec. time of serv.	15.22	0.03	3	$c_1{f_1}^{13}{f_3}^2 + c_2{f_1}^{13}f_2 + c_3{f_1}^{15}{f_3}^5$
Energy cons. of dev.	15.85	0.03	3	$c_1 + c_2f_1 + c_3{f_2}^2 + c_4f_1{f_3}^2 + c_5f_1{f_2}^3$
Transferred state size	0	0	0	c_1
Memory usage	4.55	0.25	2	$c_1 + c_2f_2 2 + c_3f_1 2f_2 + c_4f_1f_2$

each of the leaf nodes is invoked. The prediction results were so satisfactory to be used for the solver. The average prediction time, however, was about 40% of execution time for original program, which was too huge to efficiently offload. To make offloading framework efficient, we reject such features to use for the performance prediction. We choose the features that make the average prediction time under 5% of execution time for original program, even though the prediction may be less accurate.

With the constraints, we generated predictors for five performance metrics: execution time on the smartphone, execution time on the server, energy consumption on the smartphone, state transfer size and memory usage for all REMs. Table 3 shows the prediction error, prediction time, number of chosen features and generated model for each predictor of an REM, `findOptimalMove`. 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. All five predictors chose only a few features from 905 candidate features to generate a predictor model. The feature f_1 indicates the depth of the game tree, f_2 the number of pieces in the input and f_3 the number of child nodes of the root node. The predictors for both execution times and energy consumption use all of the aforementioned three features and predict with an error rate around 15%. The prediction error and prediction time of the predictor for transfer state size are both 0. This is due to the fact that when this method is offloaded to the server, the same amount of state is transferred regardless of the input.

Users usually want to reduce energy consumption as well as execution time. So, we set the solving policy for Chess engine to only migrate when offloading would benefit both execution time and energy consumption. To see the impact of our f_mantis predictor on the efficiency of mobile execution offloading, we compared it with the following basic techniques: Standalone, ROE, History-based prediction, PE (partial execution) and Oracle. Standalone, ROE, History-based prediction and Oracle are the same as described in Section 2. The PE predictor predicts by extracting features from the code and modeling a prediction function with it, just as f_mantis does. The only difference is that the PE predictor does not utilize slicing techniques, so it needs to run the original code until it can obtain all of its chosen features.

Fig. 9 shows the results of code partitioning for twenty inputs with History-based and f_Mantis prediction. In this figure, the line 'Original' represents the profile results for the execution time and each prediction technique's line represents the predicted execution time. The circle in the figure represents the decision whether the method should run at remote server or not is correct. Contrary to this, the cross represents the decision is incorrect. The figure shows code partitioning with f_Mantis always makes correct decision. On the other hand, code partitioning with History-based prediction often makes wrong decision.

Table 4 shows how efficient it is to make offloading decisions with the f_Mantis predictor. It only has an 2.2% overhead in execution time and 3.5% in energy consumption compared to Oracle. On the other hand, History-based prediction shows worse results than remote-only execution and while PE prediction made the same predictions as f_Mantis, its prediction overhead was too large.

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

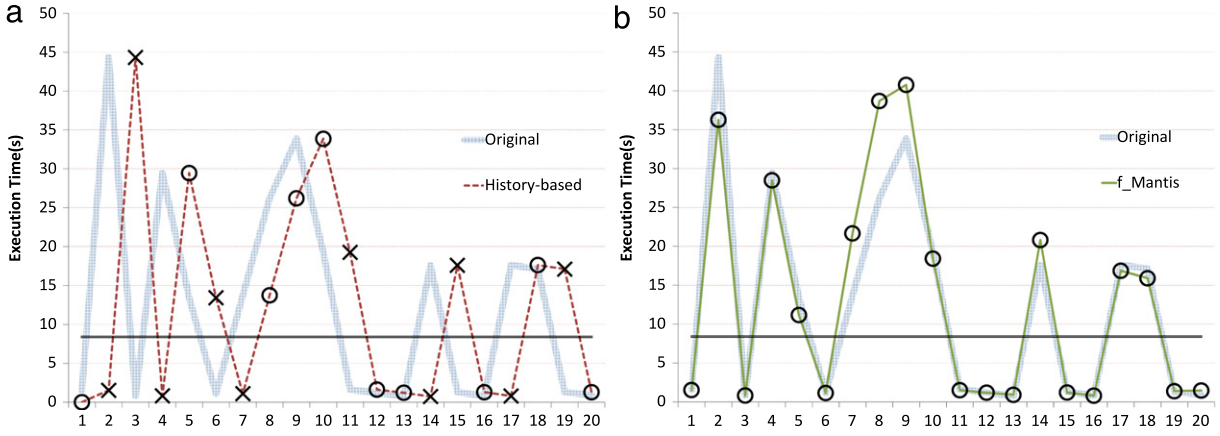

Fig. 9. The results of code partitioning for Chess Engine with (a) History-based and (b) f_Mantis prediction.

Table 4
Offloading results using each prediction techniques for Chess Engine.

	Oracle	ROE	Standalone	HB	PE	f_Mantis
Total execution time (s)	35,950	46,328	53,807	50,095	73,856	36,746
Total energy consumption (mAh)	2,733	3,822	4,945	4,372	2,863	2,828

Table 5
Performance prediction results for Face Detection.

Prediction target	Prediction error (%)	Prediction time (%)	No. of chosen features	Generated model
Exec. time of dev.	4.45	0.6	2	$c_1 + c_2 f_1 f_2 + c_3 {f_1}^2 f_2 + c_4 f_1$
Exec. time of serv.	4.74	0.6	2	$c_1 + c_2 f_1 f_2 + c_3 {f_1}^2 f_2 + c_4 {f_1}^3 f_2 + c_5 {f_1}^2$
Energy cons. of dev.	6.57	0.6	2	$c_1 + c_2 f_1 f_2 + c_3 f_1 + c_4 {f_1}^2$
Transferred state size	0.01	0.6	2	$c_1 + c_2 f_3$

The Android dalvik VM has a different memory limit for applications depending on the version of the OS or the configuration set by the manufacturer. This occasionally leads to unintentional OOM (out-of-memory) errors, when an application requires more memory than allowed. Mobile execution offloading can allow an application to utilize the vast memory of a server instead of getting an OOM error. Thinkair restarts a program on a server when it receives an OOM error. The f_Mantis predictor, on the other hand, can predict whether it will run out of memory beforehand and start it on the server rather than restarting an application when it gets an error.

Table 3 shows the prediction results of memory usage which would be used in the situation mentioned above. The predictor uses a model made of only two features, and runs with 4.55% prediction error and 0.25% prediction time overhead. With the predictor, we ran 1000 inputs that could possibly get an OOM error. 142 of the inputs actually made the application run out of memory when run on the device. The f_Mantis predictor successfully predicted 136 of these inputs beforehand and offloaded them to a memory-rich server.

5.3.2. Face detection

This application detects faces in an image by using the OpenCV library [21]. It outputs a copy of the image, outlining faces with a red box. As the application might need to accept a continuous stream of images from a video to detect the faces in it, execution time is a critical factor. First, the application receives a jpeg image as input and transforms it to a bitmap image for analysis. As the original jpeg image is smaller than the converted bitmap image, it would usually be advantageous to offload the application before the image conversion. Therefore, we annotated the point as an REM.

To generate predictors for Face Detection, we used 100 randomly cut images with a size between 100 × 100 and 1000 × 3000 pixels for training data. As seen in Table 5, the predictor generator selected 2 features, the width (f_1) and height (f_2) of the original image, from 107 candidate features for the predictors of execution time and energy consumption. For the prediction of state transfer cost, file size of the original jpeg image (f_3) was selected.

Table 5 shows the results of performance prediction for the REM. Using the predictor, we run Face Detection to be offload the server with 1000 inputs. As the pixel count of the bitmap image dominates the execution time and energy consumption of Face Detection, the predictors for them show high accuracy with error rates around 5%. The predictor for transferred state size also shows high accuracy because the original image, whose size is the feature, is the majority of the state.

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

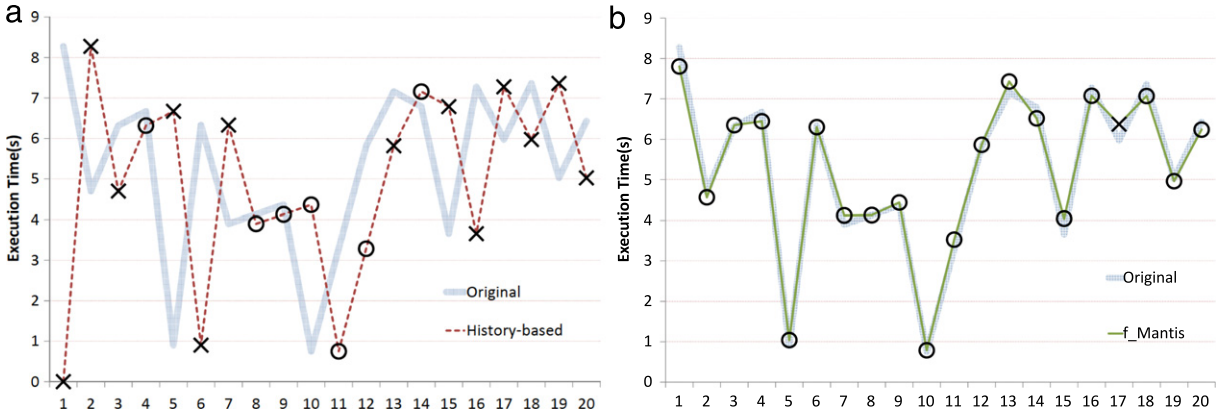

Fig. 10. The results of code partitioning for Face Detection with (a) History-based and (b) f_Mantis prediction.

Table 6
Offloading results using each prediction technique for Face Detection.

	Oracle	ROE	Standalone	HB	f_Mantis
Total execution time (s)	4346	6157	6077	6108	4385
Total energy consumption (mAh)	479	596	844	742	483

Table 7
Performance prediction results for Invader.

Prediction target	Prediction error (%)	Prediction time (%)	No. of chosen features	Generated model
Exec. time of dev.	10.06	0.0	2	$c_1 + c_2 f_1 + c_3 f_1 f_2$
Exec. time of serv.	5.37	0.0	0	c_1
Energy cons. of dev.	11.74	0.0	2	$c_1 + c_2 f_1 + c_3 f + 2 + c_4 f_2 f_1{}^3 + c_5 f_2{}^3 f_1$
Transferred state size	0.00	0.0	0	c_1

Fig. 10 shows the results of code partitioning for twenty inputs with History-based and f_Mantis prediction. In this graph, code partitioning with f_Mantis makes wrong decision only once. However, code partitioning with History-based prediction makes right decisions for under half inputs. As seen in Table 6, f_Mantis shows performance that almost matches Oracle, while the performance of offloading with History-based prediction shows little improvement over ROE or Standalone.

5.3.3. Invaders

This application is a demo 3D game in libGDX [22], a cross-platform game development framework. It receives the device's gyrosensor and touchscreen data as input. It uses these inputs to calculate a change in the game state and renders a frame to display, which contains objects representing data in the game state. The rendering part of this process takes up most of the application's execution time, and the rendering time for each frame depends on the game state. The rendering process is repeated continuously while running the application and its speed is measured as frame per second (FPS). As the computational power of the server is better than the device, we expected to get a higher FPS for offloading the render function to the server. However, the transfer cost for offloading Invaders, sending game state to the server and receiving rendered images from the server, turned out to be too large for offloading to be beneficial with the original SorMob framework. Especially, when the human eye requires at least a frame rate of 20 fps to accept images as a continuous flow, the time overhead, which is around a few seconds for each frame, is unacceptable. To address this problem, we altered SorMob to build a framework fit to run interactive applications similar to Invaders. After an initial offloading environment setup for an application is done, state transferring, rendering and image transfer are done in a software pipeline. Through this new framework, we were able to achieve a lower bound of 20 fps regardless of the state of Invaders. The energy consumption, however, is higher than just running the application on the device due to the constant network traffic. So the framework decides to offload only when the predicted rendering time will result in an FPS under 20.

We used the execution time and energy consumption of rendering 100 random game states as training data. Table 7 shows the prediction results of the predictor which are generated for the performance of the rendering method. The features f_1 and f_2, used for the predictors for the execution time and the energy consumption of the device, are the number of object1 (enemy ships) and the number of object2 (defensive obstacles) respectively. As these features are easily obtainable by just checking two integer values, the prediction time becomes negligible. The predictors for server execution time and state transfer size use constants as their prediction models, thus, their prediction time overhead also became trivial. The reason why the state transfer size predictor uses a constant as its prediction model is because the only state transferred to

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

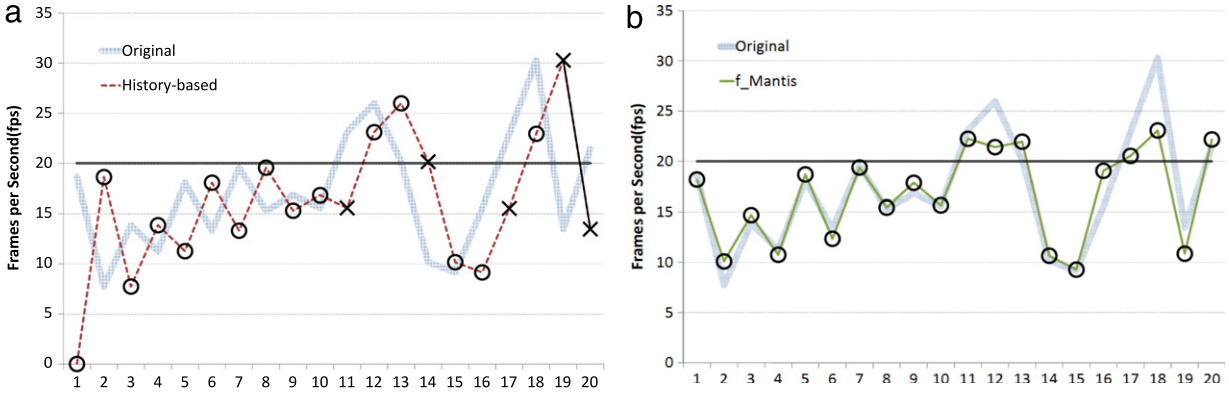

Fig. 11. The results of code partitioning for Invader with (a) History-based and (b) f_Mantis prediction.

Table 8
Offloading results using each prediction techniques for Invader.

	Oracle	ROE	Standalone	HB	f_Mantis
Average FPS (frame)	20.02	20.08	15.71	18.91	19.98
Total energy consumption (uAh)	6732	7553	4630	6094	6642
Taken input count	827	758	575	681	808
STDEV	1.33	1.62	3.91	3.00	1.36

offload rendering is the game state and rendered image, which means their size is always constant. The predictor for server execution time, on the other hand, uses a constant model because, regardless of the amount of objects to display, the server can easily render an image with the size of the resolution of a mobile device. As seen in the offloading results of Table 7, the predictor proved to be accurate enough.

To show how well a predictor generated by our framework would work, we tweaked the Invader to render a random game state every 0.05 s. Then, in our modified framework, whenever the game state is changed, a prediction would be made to decide whether to render locally or remotely. Fig. 11 shows the results of code partitioning for twenty inputs with History-based and f_Mantis prediction. In this figures, f_Mantis always makes correct decision whether to run the method remotely. History-based prediction, however, often makes wrong decisions. Table 8 shows the results of running the tweaked Invader for 50 s in our modified framework. The row 'Taken input count' shows how many game states were successfully acknowledged and rendered by the application. Some state changes are missed by the framework when the rendering of the former frame takes too long and the state is changed again, because it is set to change every 0.05 s, before it could be read to be rendered.

When running the application stand alone, the FPS is around 15, which will appear clunky to the user. When always rendering the application remotely, the FPS is kept stable around 20. However, the energy consumption reaches 1.6 times of that of running stand alone. Oracle shows an ideal case of offloading, yet even it can only render 827 out of 1000 state changes. This is because some states require a little bit more time than 0.05 s to render even on the server. ROE rendered 758 out of 1000 and Standalone a mere 575. Offloading with f_Mantis, on the other hand, shows nearly the same performance as Oracle in execution time, energy consumption and even taken input count. History-based prediction seems to show good enough performance with a high average FPS, but when we look at its standard deviation in FPS, which is over twice the amount of Oracle's, we can see that its performance is not stable.

6. Related work

Much research has been devoted to modeling system behavior as a means of prediction for databases [23,24], cluster computing [25,26], networking [27–29], program optimization [30,31], 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 [32,33], database query execution-time and resource prediction [23,24], performance prediction for streaming applications based on control flow characterization [34], violations of Service-Level Agreements (SLAs) for cloud and web services [25,26], and load balancing for network monitoring infrastructures [35]. 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.

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

Delving further into extraction of non-trivial features, research has explored extracting predictors from execution traces to model program complexity [36], to improve hardware simulation specificity [37,38], and to find bugs cooperatively [39]. 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 [27–29].

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 [40–44] 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 [36], simulation to derive worst-case running time [45], and symbolic execution and abstract evaluation to derive either worst-case inputs for a program [46], or asymptotic bounds on worst-case complexity [47,48]. In contrast, our goal is to automatically generate an online, accurate predictor of the performance of particular invocations of a general-purpose program.

Finally, our predictor is based on our earlier work [7]. In the prior work, we introduce program slicing to compute features cheaply and generate predictors automatically, apply the whole system to Android smartphone applications on multiple hardware platforms, and evaluate the benefits of slicing thoroughly. In this work, we modified Mantis to do method-wise performance prediction at runtime.

Previous work has proposed many techniques that aim to empower mobile device with computational infrastructures. Satyanarayanan et al. [49,50] proposed one of the earliest studies which migrated the full VM or a small VM overlay along with a process running on the device. As huge size of the data are transferred, this technique is not applicable on mobile devices.

In order to realize offloading mobile computation to a server over wireless LAN or even 3G networks, many recent works have proposed process-level migration approaches, which is transferring only the state and related heap objects of a process. Some of those use static partitioning schemes. In static partitioning schemes, the decision whether to offload or not is fixed regardless of the execution environment. So, a performance predictor is not needed as there is no need to estimate a program's characteristic. Ryan et al. [51] proposed a solution for optimal partitioning of sensor network application code between sensor nodes and servers. It statically partitions the code using profile-based approach to reduce the use of CPU and network bandwidth. Cuckoo [52] is a framework for offloading mobile device applications to a cloud server. For this framework to work, applications need to be re-written to match their programming model. After the application code is generated during compile time, the Cuckoo framework always tries to offload the application to a cloud server.

To elastically offload a program to a server, a majority of studies introduced dynamic partitioning schemes. CloneCloud [2] suggests an elastic execution offloading approach for the Android System. CloneCloud uses a profile-based approach for partitioning. It automatically partitions the program into parts that should be offloaded and parts that should run locally at a certain network bandwidth from profiled data. Therefore, CloneCloud reduces the overhead of modifying the application code. However, it needs to transfer a rather large size of state to the server. Giurgiu et al. [53] also use a profile-based approach to dynamically distribute different layers of an application between the server and the smartphone. It automatically determines which application module should be offloaded in order to get high performance or low cost. These approaches rely on profiled application performance, limiting them to only work well on a set number of offloading scenarios, unable to react to any change in application performance, which might be caused by different inputs or network speed from the profiled run.

At run time, OLIE [54] monitors the current memory status and network bandwidth and it decides whether it should offload the application or not to overcome memory resource constraints of mobile devices. However, OLIE does not predict future memory requirements nor does it consider execution time or energy consumption.

MAUI [3] decides at run-time which method should be offloaded in order to achieve optimal energy consumption by predicting the performance of the application and transferring time of states, which is based on the transferred state size and network speed, as long as the decision does not increase execution time. More recently, ThinkAir [4] suggests an offloading framework that migrates smartphone applications to the cloud. It allocates more than one clone VM image to exploit parallelism and relieve the lack of memory space. It chooses the round-trip time or the time to transfer a certain amount of data as features and builds the model for network speed using the features, which is similar prediction method to feature-based approach. These features can be commonly applied regardless of types of target program and are relatively easy to obtain. On the other hand, it is difficult to automatically choose the features and create the model for program performance, as there are no features which can be commonly applied to every program.

COMET [55] adopted distributed shared memory to expand the range of offloadable code and consequently, allows multiple threads to be offloaded simultaneously. Inspired by MAUI, Kovachev et al. [5] proposed more sophisticated techniques for profiling, monitoring and partitioning. These approaches use historical prediction based on monitoring to lower the runtime overhead of prediction, which makes it hard for them to make a good prediction when a program's performance fluctuates greatly.

Odessa [56] dynamically partitions applications using a greedy algorithm, and adaptively makes offloading decisions. In the paper, Moo-Ryong Ra et al. showed various factors, especially input variability, can affect application performance. In order to offload effectively, accurate predictions of execution are required on both the smartphone and the server. Therefore,

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

Odessa periodically acquires information from a low overhead run-time profiler to estimate the bottleneck in the current configuration.

7. Conclusion

In this paper we proposed a mobile offloading solver with f_Mantis, which is a runtime performance prediction generator. Our results show that f_Mantis can accurately predict the gains and costs of offloading a method at a certain point in a program in consideration of different program inputs or device states. By utilizing f_Mantis, we showed the possibility of a new solver to make precise offloading decisions which further reduces the execution time or energy consumption of an application.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2014R1A2A1A10051792), IDEC, the Brain Korea 21 Plus Project in 2015 and Inter-University Semiconductor Research Center (ISRC).

References

[1] S. Yang, Y. Kwon, Y. Cho, H. Yi, D. Kwon, J. Youn, Y. Paek, Fast dynamic execution offloading for efficient mobile cloud computing, in: 2013 IEEE International Conference on Pervasive Computing and Communications, PerCom, vol. 0, 2013, pp. 20–28. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/PerCom.2013.6526710.
[2] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti, CloneCloud: elastic execution between mobile device and cloud, in: Proc. ACM EuroSys, 2011, pp. 301–314. http://dx.doi.org/10.1145/1966445.1966473.
[3] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl, MAUI: making smartphones last longer with code offload, in: Proc. ACM MobiSys, 2010, pp. 49–62. http://dx.doi.org/10.1145/1814433.1814441.
[4] S. Kosta, A. Aucinas, P. Hui, R. Mortier, X. Zhang, ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading, in: Proc. IEEE INFOCOM, 2012, pp. 945–953. http://dx.doi.org/10.1109/INFCOM.2012.6195845.
[5] D. Kovachev, T. Yu, R. Klamma, Adaptive computation offloading from mobile devices into the cloud, in: Proc. IEEE ISPA, 2012, pp. 784–791. http://dx.doi.org/10.1109/ISPA.2012.115.
[6] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, M. Naik, Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression, in: NIPS, 2010.
[7] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, Y. Paek, Mantis: Automatic performance prediction for smartphone applications, in: Presented as part of the 2013 USENIX Annual Technical Conference, USENIX, San Jose, CA, 2013, pp. 297–308. URL https://www.usenix.org/conference/atc13/technical-sessions/papers/kwon.
[8] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, Springer, 2009.
[9] R. Tibshirani, Regression shrinkage and selection via the lasso, J. Royal. Statist. Soc. B.
[10] T. Zhang, Adaptive Forward-backward Greedy Algorithm for Sparse Learning with Linear Models, in: NIPS, 2008.
[11] M. Weiser, Program Slicing, 1981.
[12] F. Tip, A survey of program slicing techniques, J. Program. Lang. 3 (3).
[13] S. Horwitz, T. Reps, D. Binkley, Interprocedural Slicing Using Dependence Graphs, 1988.
[14] T.W. Reps, S. Horwitz, S. Sagiv, G. Rosay, Speeding Up Slicing, 1994.
[15] Javassist, www.csg.is.titech.ac.jp/~chiba/javassist, product page (2012).
[16] Octave, www.gnu.org/software/octave, product page (2013).
[17] JChord, code.google.com/p/jchord, product page (2012).
[18] Jasmin, jasmin.sourceforge.net, product page (2004).
[19] AspectJ, https://eclipse.org/aspectj/, product page (2005).
[20] Kyro, code.google.com/p/kryo/, product page (2005).
[21] OpenCV library, http://sourcefouge.net/projects/opencvlibrary/, product page (2005).
[22] libgdx, http://libgdx.badlogicgames.com/, product page (2005).
[23] C. Gupta, A. Mehta, U. Dayal, PQR: Predicting Query Execution Times for Autonomous Workload Management, 2008.
[24] A. Ganapathi, H. Kuno, U. Dayal, J.L. Wiener, A. Fox, M. Jordan, D. Patterson, Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning, 2009.
[25] P. Bodik, R. Griffith, C. Sutton, A. Fox, M.I. Jordan, D.A. Patterson, Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters, 2009.
[26] P. Shivam, S. Babu, J.S. Chase, Learning Application Models for Utility Resource Planning, 2006.
[27] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, Y.-M. Wang, WebProphet: Automating Performance Prediction for Web Services, 2010.
[28] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, M. Ammar, Answering What-if Deployment and Configuration Questions with Wise, 2008.
[29] S. Chen, K. Joshi, M.A. Hiltunen, W.H. Sanders, R.D. Schlichting, Link Gradients: Predicting the Impact of Network Latency on Multitier Applications, 2009.
[30] K. Tian, Y. Jiang, E. Zhang, X. Shen, An Input-centric Paradigm for Program Dynamic Optimizations, 2010.
[31] Y. Jiang, E. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, Y. Gao, Exploiting Statistical Correlations for Proactive Prediction of Program Behaviors, 2010.
[32] K. Vaswani, M. Thazhuthaveetil, Y. Srikant, P. Joseph, Microarchitecture Sensitive Empirical Models for Compiler Optimizations, 2007.
[33] B. Lee, D. Brooks, Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction, 2006.
[34] F. Aleen, M. Sharif, S. Pande, Input-driven Dynamic Execution Behavior Prediction of Streaming Applications, 2010.
[35] P. Barlet-Ros, G. Iannaccone, J. Sanjuas-Cuxart, D. Amores-Lopez, J. Sole-Pareta, Load Shedding in Network Monitoring Applications, 2007.
[36] S. Goldsmith, A. Aiken, D. Wilkerson, Measuring Empirical Computational Complexity, 2007.
[37] T. Sherwood, E. Perelman, B. Calder, Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications, 2001.
[38] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically Characterizing Large Scale Program Behavior, 2002.
[39] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, M.I. Jordan, Scalable Statistical Bug Isolation, 2005.
[40] J. Gustafsson, A. Ermedahl, C. Sandberg, B. Lisper, Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution, in: RTSS, 2006.
[41] Y.-T. S. Li, S. Malik, Performance Analysis of Real-Time Embedded Software, Kluwer Academic Publishers, 1999.
[42] R. Wilhelm, Determining bounds on execution times, in: Handbook on Embedded Systems.

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001

[43] S. Seshia, A. Rakhlin, Game-theoretic timing analysis, in: ICCAD, 2008.
[44] S. Seshia, A. Rakhlin, Quantitative analysis of systems using game-theoretic learning, ACM TECS.
[45] R. Rugina, K.E. Schauser, Predicting the running times of parallel programs by simulation, in: IPPS/SPDP, 1998.
[46] J. Burnim, S. Juvekar, K. Sen, WISE: Automated Test Generation for Worst-case Complexity, 2009.
[47] B. Gulavani, S. Gulwani, A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis, 2008.
[48] S. Gulwani, K. Mehra, T. Chilimbi, SPEED: Precise and Efficient Static Estimation of Program Computational Complexity, 2009.
[49] M. Satyanarayanan, B. Gilbert, M. Toups, N. Tolia, A. Surie, D.R. O'Hallaron, A. Wolbach, J. Harkes, A. Perrig, D.J. Farber, M.A. Kozuch, C.J. Helfrich, P. Nath, H.A. Lagar-Cavilla, Pervasive personal computing in an internet suspend/resume system, IEEE Internet Comput. 11 (2) (2007) 16–25. http://dx.doi.org/10.1109/MIC.2007.46.
[50] M. Satyanarayanan, P. Bahl, R. Caceres, N. Davies, The case for VM-based Cloudlets in mobile computing, IEEE Pervasive Comput. 8 (4) (2009) 14–23. http://dx.doi.org/10.1109/MPRV.2009.82.
[51] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, S. Madden, Wishbone: profile-based partitioning for sensornet applications, in: Proc. USENIX NSDI, 2009, pp. 395–408.
[52] R. Kemp, N. Palmer, T. Kielmann, H.E. Bal, Cuckoo: A computation offloading framework for smartphones, in: Proc. MobiCASE, in: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 76, Springer, 2010, pp. 59–79. URL http://dblp.uni-trier.de/db/conf/mobicase/mobicase2010.html.
[53] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, G. Alonso, Calling the cloud: enabling mobile phones as interfaces to cloud applications, in: Proc. ACM/IFIP/USENIX Middleware, 2009, pp. 83–102.
[54] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, D. Milojicic, Adaptive offloading inference for delivering applications in pervasive computing environments, in: Proc. IEEE PerCom, 2003, pp. 107–114. http://dx.doi.org/10.1109/PERCOM.2003.1192732.
[55] M.S. Gordon, D.A. Jamshidi, S. Mahlke, Z.M. Mao, X. Chen, COMET: Code offload by migrating execution transparently, in: Proc. ACM OSDI, USENIX Association, Berkeley, CA, USA, 2012, pp. 93–106. URL http://dl.acm.org/citation.cfm?id=2387880.2387890.
[56] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, R. Govindan, Odessa: enabling interactive perception applications on mobile devices, in: Proc. ACM MobiSys, 2011, pp. 43–56.

Please cite this article in press as: Y. Kwon, et al., Precise execution offloading for applications with dynamic behavior in mobile cloud computing, Pervasive and Mobile Computing (2015), http://dx.doi.org/10.1016/j.pmcj.2015.10.001