creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Ph.D. DISSERTATION

Automatic prediction of computational resource consumption for efficient task migration in cloud

모바일 클라우드 환경에서의 효율적인 작업 이전을 위한 자동 성능 예측 기법

BY

권 용 인

FEBRUARY 2015

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Automatic prediction of computational resource consumption for efficient task migration in cloud

모바일 클라우드 환경에서의 효율적인 작업 이전을 위한 자동 성능 예측 기법

BY

권 용 인

FEBRUARY 2015

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Automatic prediction of computational resource consumption for efficient task migration in cloud

모바일 클라우드 환경에서의 효율적인 작업 이전을 위한 자동 성능 예측 기법

지도교수 백 윤 흥

이 논문을 공학박사 학위논문으로 제출함

2014 년 11 월

서울대학교 대학원

전기 컴퓨터 공학부

권 용 인

권 용 인의 공학박사 학위논문을 인준함

2014 년 12 월

위 원 장	문수묵
부위원장	백윤흥
위 원	정교민
위 원	전병곤
위 원	김장우

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.

Moreover, existing schemes completely ignore the costs of cloud resources by assuming that idle servers are always available for free of charge. These unrealistic assumptions make each server run only a small load to achieve the guaranteed high offload performance. Therefore, these schemes cannot be applied to real-world commercial clouds which aim to minimize the operation costs by maximizing the server throughput, and then charge users for their resource usage.

Thus in this dissertation, I present Mantis, a framework for predicting the Computational Resource Consumption(CRC) of Android applications on given inputs accurately, and 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. I apply program slicing to reduce evaluation time of a feature and automatically generate executable code snippets for efficiently evaluating features. Using the techniques, I empirically

show they enhance the performance of offloading. Lately, I propose *CMcloud*, a novel cost-effective mobile-to-cloud offloading platform, which works nicely under the real-world cloud environments. CMcloud minimizes both the server costs and the user service fee by offloading as many mobile applications to a single server as possible, while satisfying the target performance of all applications.

Keywords: Mobile Cloud Computing, Smartphone, Performance prediction, Mobile Offloading

Student Number: 2010-30209

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Mobile Execution Offloading

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 exceed 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.

1.2 Dynamic Code Partitioning

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, 3, 4, 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.

1.3 Cost-effectivity of Mobile Execution Offloading

The existing offload schemes depend on many unrealistic assumptions as they blindly ignore the costs of using servers in the cloud. For example, they assume that a target server is always available for free of charge, the server's load is always idle or stable, the application has been previously profiled for the target server, and the post-offload performance matches the user-expected performance. As a result, the existing schemes naturally choose to offload a mobile application to a free server in the cloud.

Therefore, the existing offload schemes are not suitable for real-world commercial cloud environments, where the cloud provider charges the users based on their cloud

resource usage and each server aims to run as many applications as possible to maximize the server throughput or minimize the server costs (i.e., *oversubscription*). In such highly-utilized clouds, the cloud provider cannot estimate the user application's post-offload performance on target cloud servers. If a target server is running other applications, the application's profiled performance will not match the actual post-offload performance, which leads to a critical Quality-of-Service (QoS) failure. On the other hand, if the cloud provider forces to maintain the initial profiling state of servers (e.g., idle or static load,) it fails to increase the server throughput, which leads to the increased server costs and the user service fee.

As a result, the real-world commercial cloud environments require a new cost-effective offloading scheme which can satisfy two fundamental requirements. First, the cloud provider must maximize the server throughput (or reduce the server costs) by running many mobile applications per server. Second, the cloud provider must satisfy the application's user-expected post-offload performance using the minimum server resources.

1.4 Dissertation Contributions and Outline

To overcome the input-sensitivity problem of performance, in this dissertation, I propose an alternative performance approach for execution offloading, which we call *feature-based* prediction. Then, to make it work nicely under the real-world cloud environments, I propose *CMcloud*, a novel cost-effective mobile cloud platform.

In Chapter 2, I introduce Mantis, a new framework to predict execution time, energy consumption, memory allocation and memory requirement of bytecode programs on given inputs accurately and efficiently. I demonstrate how the proposed approach extracts and utilizes program features for the predictions.

In Chapter 3, I propose a new offloading solver which uses f_Mantis,an extension

of Mantis. Based on the prediction results by f_Mantis, the solver precisely offloads mobile programs in order to match the user's various needs. The solver also prevent out-of-memory error.

In Chapter 4, I propose *CMcloud*, a novel cost-effective mobile cloud platform, which works nicely under the real-world cloud environments. CMcloud exploits a novel performance modeling methodology for estimating the target application's post-offload performance on any target server, regardless of its current utilization. At the same time, CMcloud allows to offload as many applications to each server as possible without violating the applications' user-expected performance. In this way, CMcloud can offer to users its QoS-guaranteed offload service at a very low price, while minimizing the cloud operation costs.

Chapter 5 concludes this dissertation with a summary and discussion of future directions.

Chapter 2

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

2.1 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 [6, 7, 8]. 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 pro-

gram'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 [9]. However, in many cases, additional features need to be extracted from within an application to accurately predict an its performance. Most existing CRC prediction techniques are domain-specific [10, 11, 12] or requiring expert knowledge [13, 14].

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 [15]. 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* [16, 17]. Given a feature, slicing computes the set of all statements in the program that may affect the value of the feature. Precise slicing could prune large portions of the program that are irrelevant to the evaluation of features. Our slices are stand-alone executable programs; thus, executing them on program inputs provides both the evaluation 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, Ring-

tone 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%, with a few exceptions peaking at 11.1%, by executing slices that spend at most 1.3% 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 chapter is organized as follows. We present the architecture of our framework in Section 2.2. Sections 2.3 and 2.4 describe our feature instrumentation and CRC-model generation, respectively. Section 2.5 describes predictor code generation using program slicing. In Section 2.6, we present our system implementation

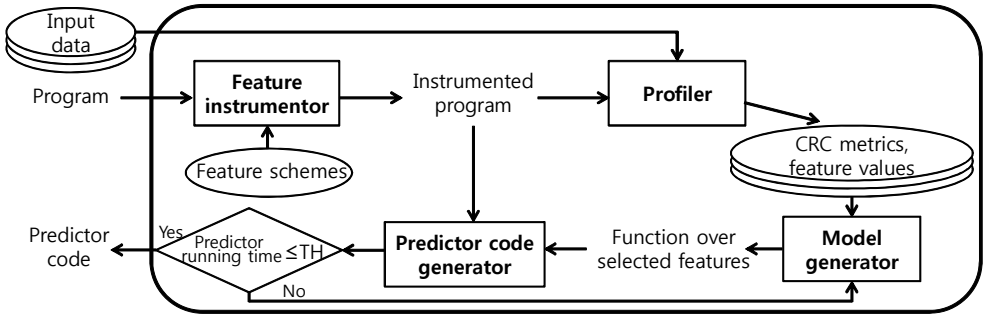


Figure 2.1 The Mantis offline stage.

and in Section 2.7, we present evaluation results. Finally, we discuss related work in Section 2.8 and conclude in Section 2.9.

2.2 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 Figure 2.1, consists of four components: a feature instrumentor, a profiler, a CRC-model

generator, and a predictor code generator.

The feature instrumentor (Section 2.3), 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 ($f_1, ..., f_M$) as per the schemes.

Next, the profiler takes the instrumented program and a set of user-supplied program inputs ($I_1, ..., I_N$). It runs the instrumented program on each of these inputs and produces, for each input I_i, a vector of feature values $(v_{i1}, ..., v_{iM})$. It also runs the 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 2.4) 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 ($f_{i1}, ..., f_{iK}$). 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 2.5) 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 ($f_{i1}, ..., f_{iK}$) from program inputs.

We apply program slicing to extract a small code snippet that computes the value of each chosen feature. A precise slicer would prune large portions of the original program that are irrelevant to evaluating a given feature and thereby provide an efficient

way to evaluate the feature. In practice, however, our framework must be able to tolerate imprecision. Besides, independent of the slicer's precision, certain features will be inherently expensive to evaluate: e.g., features whose value is computed upon program termination, rather than derived from the program's input. We define a feature as *expensive to evaluate* if the execution time of its slice exceeds a threshold (TH) expressed as a fraction of program execution time. If any of the chosen features ($f_{i1}, ..., f_{iK}$) is expensive, then via the *feedback loop* in Figure 2.1 (at the bottom), our framework reruns 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.

2.3 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.

2.4 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 i^{th} observation of execution time, and $\mathbf{v}_i$ denotes the i^{th} observation of the vector of M features.

Least square regression is widely used for finding the best-fitting $\lambda(\mathbf{v}, \beta)$ to a given set of responses t_i by minimizing the sum of the squares of the residuals [18]. 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 [18]. In recent years a number of approximate algorithms have been proposed as efficient alternatives for simultaneous feature selection and model fitting. Widely used among them are LASSO (Least Absolute Shrinkage and Selection Operator) [19] and FoBa [20], 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 [21, 22]. 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 [20].

For our system, we chose the SPORE-FoBa algorithm, which we proposed [15], to build a predictive model from collected features. In our work, we showed that SPORE-FoBa outperforms LASSO and FoBa. The FoBa component of the algorithm helps cut down the number of interesting features first, and the SPORE component builds a fixed-degree (d) polynomial of all selected features, on which it then applies sparse, polynomial regression to build the model. For example, using a degree-2 polynomial with feature vector $\mathbf{v} = [x_1\ x_2]$, we expand out $(1 + x_1 + x_2)^2$ to get terms $1,\ x_1,\ x_2,\ x_1^2,\ x_1x_2,\ x_2^2$, and use them as basis functions to construct the following function for regression:

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

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

For a program whose execution time may dynamically change over time as the workload changes, our 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 [23], it is feasible to update the set of feature terms and their coefficients $\beta_j's$ online upon every execution time being collected.

2.5 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* [16, 17] 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 2.5.1), describes the challenges of slicing and our approach to addressing them (Section 2.5.2), and provides the design of our slicer (Section 2.5.3).

2.5.1 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 1: A Java program may read a system property lazily, late in its execution, and depending upon its value decide whether or not to perform an expensive computation:

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

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

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

```
Reader r = new Reader(new File(name));
String s;
while ((s = r.readLine()) != null) {
```

```
f_loop++; // feature inst.
process(s); // expensive computation
}
```

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

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

2.5.2 Slicer Challenges

There are several key challenges to effective static slicing. Next we discuss these challenges and the approaches we take to address them. Three of these are posed by program artifacts—procedures, the heap, and concurrency—and the fourth is posed by our requirement that the slices be executable.

Inter-procedural Analysis: The slicer must compute data and control dependencies efficiently and precisely. In particular, it must propagate these dependencies *context-sensitively*, that is, only along inter-procedurally realizable program paths—doing otherwise could result in inferring false dependencies and, ultimately, grossly imprecise slices. Our slicer uses existing precise and efficient inter-procedural algorithms from the literature [24, 25].

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 [26].

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.[27]). 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.

2.5.3 Slicer Design

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

Computing System Dependence Graph: For each method reachable from the program's root method (e.g., `main`) by our call-graph analysis, we build a Program Dependence Graph (PDG) [25], whose nodes are statements in the body of the method and whose edges represent intra-procedural data/control dependencies between them. For uniform treatment of memory locations in subsequent steps of the slicer, this step also performs a mod-ref analysis[1] and creates additional nodes in each PDG denoting implicit arguments for heap locations and globals possibly read in the method, and return results for those possibly modified in the method.

The PDGs constructed for all methods are stitched into a System Dependence Graph (SDG) [25], 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

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

be reused for further analyses.

Augmenting System Dependence Graph: This step uses the algorithm by Reps, Horwitz, Sagiv, and Rosay [24] 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 [25] 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 correspond-

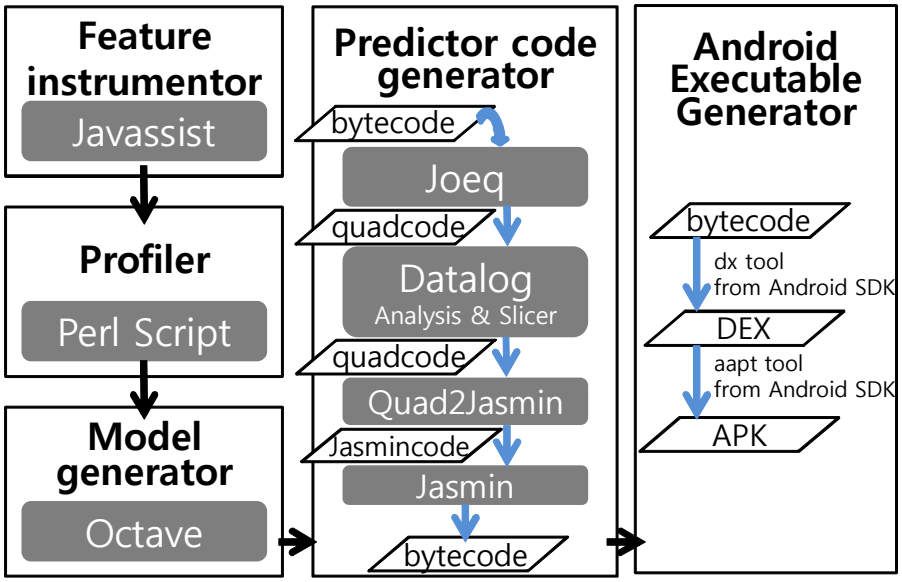


Figure 2.2 Mantis prototype toolchain.

ing 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.

2.6 Implementations

We have built a prototype of Mantis implementing the instrumentor, profiler, model generator and predictor code generator (Figure 2.2). The prototype is built to work with Android application binaries. We implemented the feature instrumentor using Javassist [28], 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 [29]

scripts. Finally, we implemented our predictor code generator in Java and Datalog by extending JChord [30], 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 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 [31] 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 needs its own personally tailored profiler. The following explains in detail how each profiler is implemented and run.

Execution time Profiler: To profile execution time, we insert a simple code to notify us when the program finishes in the end of the program. Then for each run of the pro-

gram on the test input, we record the time when we start the program and record the time when we are notified that the program has ended. We use the difference between these two recorded values as the execution time for that particular run.

Energy Consumption Profiler: As the energy consumption measured by the Android system is not accurate enough, we used an external device [32] to attain a more accurate measurement of the energy consumed by a single run of an application. In order to get the energy consumption data of a single application run, we first ready the application and its input on the device, then we start the external device and a few moments later we run the application. After the application finishes we stop the external device and save the data, which is the energy consumption of that particular run. As with the other profilers the process is fully automated.

Accumulated memory allocation Profiler: For the profiling of accumulated memory allocation, we utilized the Debug class from Android. As the method getGlobalAllocSize from the Debug class gives the global size of objects allocated by the runtime between a start and stop point, we simply instrument the target application to insert the starting and stopping point at the beginning and the end of the program respectively, and to call the getGlobalAllocSize method right after the stop point. By running this the newly instrumented code, we can collect the value returned from getGlobalAllocSize as the accumulated memory allocation.

Memory requirement Profiler: In Android 2.3(gingerbread) or higher, which includes the version we used, when heap allocation fails, the dalvik garbage collector concurrently runs to reclaim memory space which will not be accessed any more. We have modified the dalvik vm to execute garbage collection whenever heap allocation is requested, even when the allocation succeeds, so that heap space is always main-

tained at a minimum size, just barely enough to run the program. We have extracted the memory requirement of each run from analysing the garbage collection log.

2.7 Evaluation

2.7.1 Evaluation Environment

We mainly use for our experiments, Galaxy Nexus running Android 4.1.2 with dual-core 1.2Ghz 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 3.1GHz quad-core CPU, and 8GB 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% 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% 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%, which means a generated predictor is accepted only if the predictor running time is less than 5% of the original program's completion time.

- *Encryptor*: This encrypts a file using a matrix as a key. Inputs are the file and

the matrix key. We use 1,000 files, each with its own matrix key. File size ranges from 10 KB to 8000 KB, and keys are 200×200 square matrices.

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

Table 2.1 Prediction error and prediction time for execution time.

Application	Error	T_{Ave}	T_{Max}
Encryptor	4.5%	0.18%	0.26s
Path Routing	5.4%	1.34%	0.29s
Spam Filter	3.1%	0.51%	0.35s
Chess Engine	11.1%	1.03%	0.46s
Ringtone Maker	4.9%	0.20%	0.24s
Face Detection	3.8%	0.62%	0.27s
Tar Archive	3.4%	1.24%	0.41s

images, of sizes between 100×100 and 900×3000 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.

- *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.

2.7.2 Experiment Results

Accurate and Efficient Prediction for Execution time:

We first evaluate the accuracy and efficiency of Mantis execution time prediction. Table 2.1 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

Table 2.2 The total number of features initially detected, the number of chosen features, selected features for execution time prediction.

Application	No. of features		Selected features
	Total	Chosen	
Encryptor	28	2	Matrix-key size (f_1), Loop count of encryption (f_2)
Path Routing	68	1	Build map loop count (f_1)
Spam Filter	55	1	Inner loop count of sorting (f_1)
Chess Engine	1084	2	# of 1-depth nodes (f_1), # of pieces (f_2)
Ringtone Maker	74	1	Cut interval length (f_1)
Face Detection	107	2	Width of image (f_1), Height of image (f_2)
Tar Archive	122	1	Total sum of the size of the text files

Table 2.3 Generated prediction models for execution time prediction.

Application	Generated model
Encryptor	$c_0 {f_1}^2 f_2 + c_1 {f_1}^2 + c_2 f_2 + c_3$
Path Routing	$c_0 {f_1}^2 + c_1 f_1 + c_2$
Spam Filter	$c_0 f_1 + c_1$
Chess Engine	$c_0 {f_1}^3 + c_1 f_1 f_2 + c_2 {f_2}^2 + c_3$
Ringtone Maker	$c_0 f_1 + c_1$
Face Detection	$c_0 f_1 f_2 + c_1 {f_2}^2 + c_2$
Tar Archive	$c_0 + c_1 f_1$

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

Mantis generated interpretable and intuitive prediction models by only choosing one or two among the many detected features unlike non-parametric methods. Table 2.2 shows the total number of features initially detected, the number of features actually chosen to build the prediction model and the selected features for execution

Table 2.4 Prediction error and prediction time for energy consumption.

Application	Error	T_{Ave}	T_{Max}
Encryptor	3.9%	0.18%	0.26s
Path Routing	4.8%	1.34%	0.29s
Spam Filter	2.8%	0.51%	0.35s
Chess Engine	10.5%	1.03%	0.46s
Ringtone Maker	4.3%	0.20%	0.24
Face Detection	3.6%	0.62%	0.27s
Tar Archive	4.1%	1.24%	0.41s

time prediction. And Table 2.3 shows 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% 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%.

Accurate and Efficient Prediction for Energy consumption:

Table 2.4 shows the prediction error and running time of Mantis-generated en-

Table 2.5 Prediction error and prediction time for memory allocation.

Application	Error	T_{Ave}	T_{Max}
Encryptor	0.0%	0.18%	0.26s
Path Routing	4.9%	1.34%	0.29s
Spam Filter	0.2%	0.51%	0.35s
Chess Engine	8.1%	1.03%	0.46s
Ringtone Maker	0.0%	0.00%	0s
Face Detection	0.1%	0.62%	0.27s
Tar Archive	1.5%	0.15%	0.64s

ergy consumption predictors. As the table shows, the generated predictors obtained error rates under 5% with the exception of the predictor for Chess Engine and all of them need only around 1% 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 2.2 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.

Accurate and Efficient Prediction for Memory Usage:

Table 2.5 shows the behaviors of the accumulated memory allocation predictors. As we can see, the overall error rate is lower, mostly under 4% and chess engine at 8.1%, 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.

Table 2.6, the predictors generated for memory requirement prediction show similar tendencies to those of the other predictors. Most of 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%. This is due to the the same reasons as stated before, which is that the usage of memory does not fluctuate as much as the execution time

Table 2.6 Prediction error and prediction time for memory requirement.

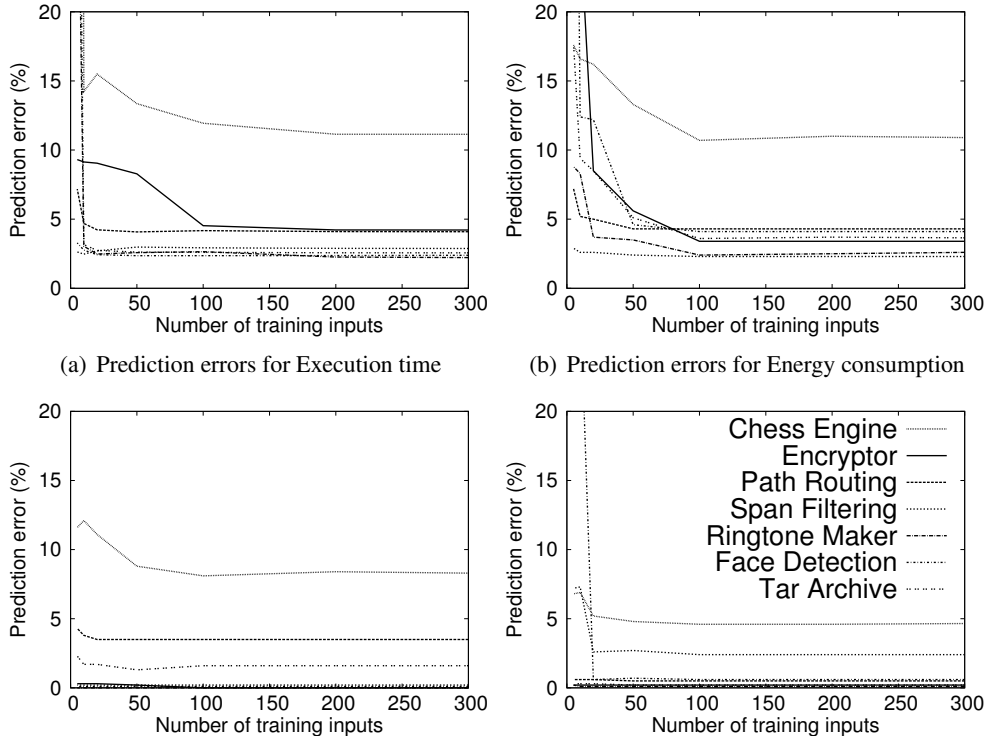
Application	Error	T_{Ave}	T_{Max}
Encryptor	0.2%	0.00%	0s
Path Routing	0.5%	1.34%	0.29s
Spam Filter	2.8%	0.51%	0.35s
Chess Engine	4.6%	0.73%	0.34s
Ringtone Maker	0.1%	0.00%	0s
Face Detection	0.6%	0.62%	0.27s
Tar Archive	0.2%	0.00%	0s

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%. 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.

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% prediction error while requiring 0.10% of the original application's running time.

The effect of the number of training samples:

We show the effect of the number of training samples on prediction errors in Figure 2.3. 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.



(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

Figure 2.3 Prediction errors varying the number of input samples. The y-axis is truncated to 20 for clarity.

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.

Benefit of Non-linear Terms on Prediction Accuracy:

Table 2.7 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 im-

Table 2.7 Prediction error of Mantis and Mantis-linear for execution time prediction. Mantis-linear uses only linear terms (f_i's) for model generation.

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

Table 2.8 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

prove 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.

Benefit of Slicing on Prediction Time:

Next we discuss how slicing improves the prediction time. In Table 2.8, we compare the prediction times of Mantis-generated predictors for execution time with those of predictors built with *partial execution*. Partial Execution (PE) runs the instrumented program only until the point where we obtain the chosen feature values.

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

Table 2.9 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

features until their values are obtained.

The ones that show around 100% 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.

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

As shown in Table 2.9, 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

Table 2.10 Prediction error and time of execution time running with Galaxy S2 and Galaxy S3.

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

feature can be obtained by BE; thus, BE creates a prediction model with only a constant term for the applications.

Prediction on Different Hardware Platforms:

Next we evaluate whether Mantis generates accurate and efficient predictors on three different hardware platforms. Table 2.10 shows the results of Mantis execution time predictor with two additional smartphones: Galaxy S2 and Galaxy S3, respectively. Galaxy S2 has a dual-core 1.2Ghz CPU and 1GB RAM, running Android 4.0.3. Galaxy S3 has a quad-core 1.4Ghz CPU and 1GB RAM, running Android 4.0.4. As shown in the table, Mantis achieves low prediction errors and short prediction times with Galaxy S2 and Galaxy S3 as well. 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 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.

Table 2.11 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

Table 2.12 Mantis offline stage processing time for execution time prediction(in seconds).

Application	P.	M.	S.	T.	Total	Iters
Encryptor	2373	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	6624	10k	6016	23k	46k	83
Ringtone Maker	2074	19	4565	2	6.7k	1
Face Detection	1437	13	6412	179	8k	4
Tar Archive	2718	80	125	913	3.8k	8

Prediction errors for multi-threaded applications:

We show the effect of multi-threads on prediction error in Table 2.11. 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 2.3Ghz CPU and 2GB 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.

Mantis Offline Stage Processing Time: Table 2.12 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.

Summary: We have demonstrated that our prototype implementation of Mantis generates good predictors that estimate four CRC metrics with high accuracy and short prediction time. We have also compared our approach to simpler, intuitive approaches based on Partial Execution and Bounded Execution, showing that Mantis works significantly better in almost all cases, and as well in even the few cases where Partial Execution happened upon good prediction features. Finally, we showed that Mantis predictors are accurate on three different hardware platforms.

2.8 Related Work

Much research has been devoted to modeling system behavior as a means of prediction for databases [10, 11], cluster computing [33, 34], networking [35, 36, 37], program optimization [38, 39], mapping parallelism [40] 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 [41, 42], database query execution-time and resource prediction [10, 11], performance prediction for streaming applications based on control flow characterization [43], violations of Service-Level Agreements (SLAs) for cloud and web services [33, 34], and load balancing for network monitoring infrastructures [12]. 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 [44] to model program complexity [13], to improve hardware simulation specificity [45, 46], and to find bugs cooperatively [47]. 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 [35, 36, 37].

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 [48, 49, 50, 51, 52] 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 [13], simulation to derive worst-case running time [53], and symbolic execution and abstract evaluation to derive either worst-case inputs for a program [54], or asymptotic bounds on worst-case complexity [55, 56]. 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 [57, 58, 59] Mantis's machine learning algorithm for prediction is based on our earlier work [15] 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.

2.9 Conclusion

In this chapter, I 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 I 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. My evaluation shows that my 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.

Chapter 3

Precise Execution Offloading for Applications with Dynamic Behavior in Mobile Cloud Computing

3.1 Introduction

The objective of this work is to propose a new solver which uses the feature-based prediction technique introduced in Chapter 2. We have implemented the solver and 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 [15] 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 chapter is organized as follows: in Section 3.2, 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 3.3, we describe our techniques to precisely and efficiently predict various aspects of program performance for execution offloading. And in section 3.4, we describe our solver which utilize our prediction techniques to offload our mobile code more precisely at runtime, attaining better performance of program execution. In Section 3.5, we experimentally demonstrate the effectiveness of our techniques which reduce significantly execution time or energy consumption. Finally, in Section 3.6 and 3.7, we relate our work with others and conclude.

3.2 Background & Motivation

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

3.2.1 Background

In order to evaluate the impact of prediction accuracy on the offloading performance, we present a simple offloading framework depicted in Figure 3.1. 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

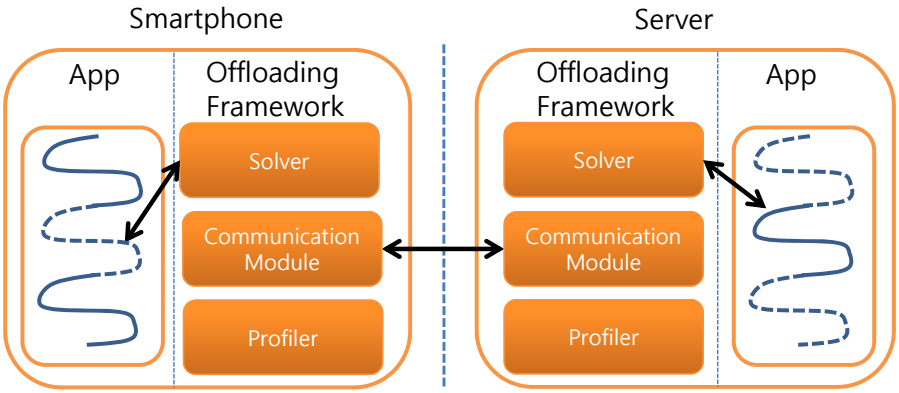


Figure 3.1 A simple architecture for mobile execution offloading.

code to distinguish these 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

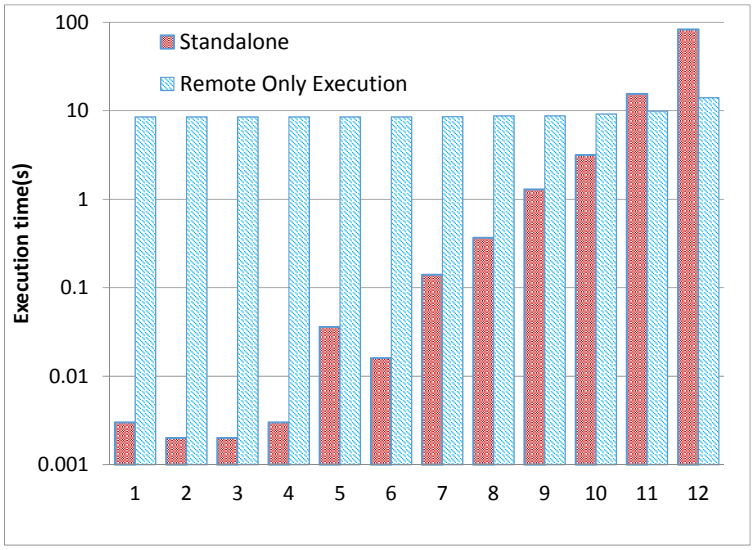


Figure 3.2 Prediction errors varying the number of input samples.

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.

3.2.2 Motivation

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 lo-

Table 3.1 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

cally 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. Figure 3.2 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 increase more dramatically than that for being offloaded.

Based on the experimental results in Figure 3.2, we compared the two other pre-

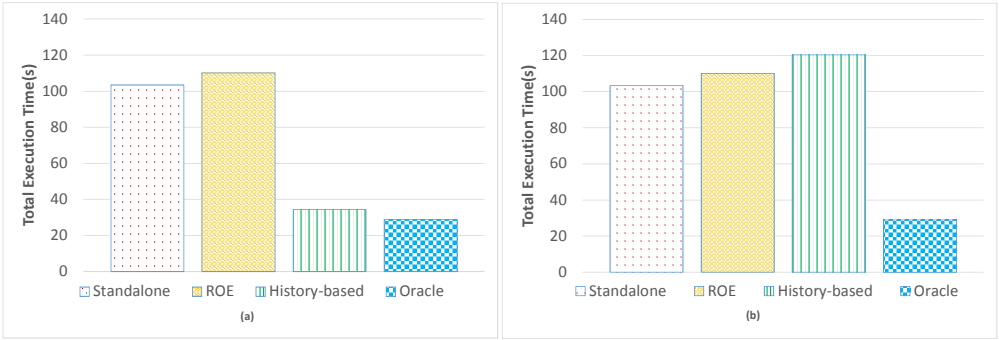


Figure 3.3 Total execution time of the N-Queens program running for 4 types of decision making with (a)ordered input set and (b)randomly mixed input set

diction schemes. In Table 3.1, we show the solving results of running the program through the inputs in order with Oracle and History-based prediction. They decides 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” denoted the solver decides to run the method on the server. We presents 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 Figure 3.3(a), we show the sum of the execution time with each prediction approaches. 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 come close to Oracle.

Next, we show the solving results of the program running through a randomly ordered set of the same inputs in table 3.2. In the table, History-based prediction often makes wrong decisions. As seen in Figure 3.3(b), the mixed input order greatly degraded the performance of History-based prediction. History-based prediction works fine for inputs that change gradually because the performance follows the gradual

Table 3.2 solving results for randomly mixed inputs with History-based prediction

Input	Local execution(s)	Remote execution(s)	Oracle	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

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.

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-

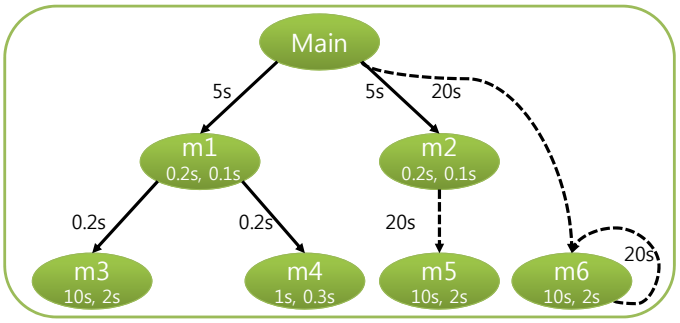


Figure 3.4 An example of a call graph

ure 3.4 presents an example of a call graph, which also shows the need of a solver for global optimization. The time 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 seconds to execute the three methods `m1`, `m3` and `m4` locally. it, on the other hand, takes 7.7 seconds to execute the methods remotely, so a locally optimization solver decides to migrate method `m1` to the server. However, it takes only 2.9 seconds 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 seconds 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 method `m2` and `m5` in Figure 3.4. The dotted edge between method `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.

3.3 f_Mantis : Automatically generation of accurate and efficient performance predictor for mobile execution offloading

In this section, we discuss our technique, *f_Mantis*, that help to efficiently predict the performance of mobile applications. f_Mantis is the extension of Mantis in Chapter 2, 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.

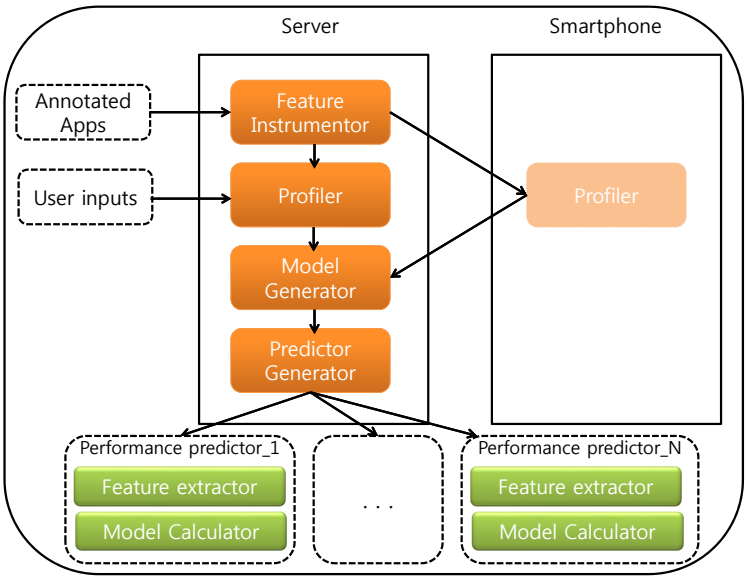


Figure 3.5 Overview of f_Mantis architecture.

3.3.1 Performance predictor generation overview

In Figure 3.5, 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 func-

tion just before running target method. And it also produces a *model calculator* which makes predictions for a given REM. The feature instrumentor and model generator are addressed in Section 2.3 and 2.4. In this section, we address the modification of the profiler and predictor generator in detail.

3.3.2 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 is recorded as well. The method call count is profiled by the instrumented code. Our profiler outputs data sets 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 i^{th} observation of the vector of the performance metrics, and v_i denotes the i^{th} observation of the vector of M features.

3.3.3 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 is 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. 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* [16, 17].

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

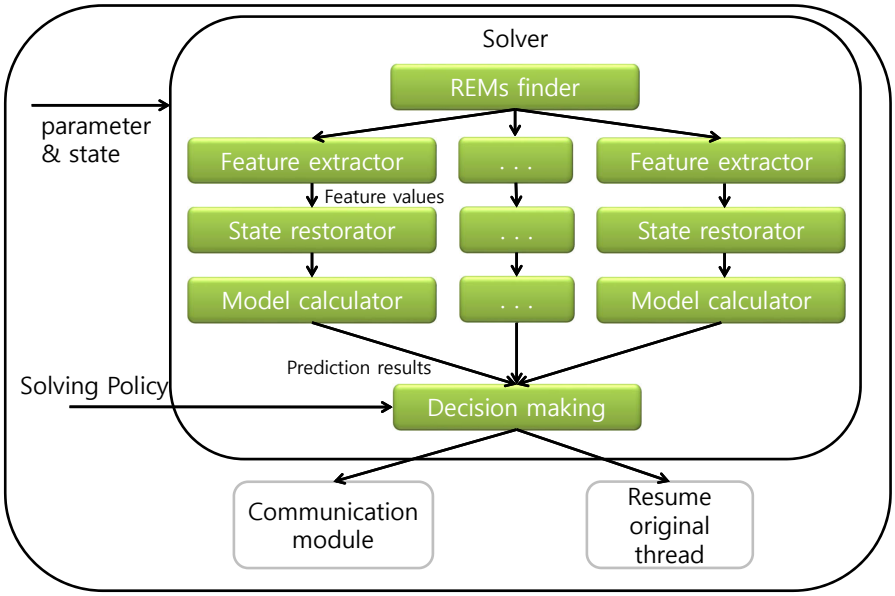


Figure 3.6 Architecture for the dynamic solver

3.4 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.

3.4.1 Architecture for our solver

Figure 3.6 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* draw 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 follow: 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 u $R_{u,v}$. 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* restore 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 priority, 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.

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

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

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

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

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.

3.5 Evaluation

3.5.1 Implementation

We have built f_Mantis, implementing the feature instrumentor, profiler, model generator and predictor generator (Figure 3.5). These are built to work with Android application binaries. We implemented the feature instrumentor using Javassist [28], 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 [29] 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 [30], 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 that could obtain the selected features. The slice is translated to Jasmin [31] 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 [61], 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 [62], which is an open source java serializer.

3.5.2 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 3.1GHz quad-core CPU, and 8GB 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 1.2Ghz CPU and 1GB 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 per-

formance 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.

3.5.3 Experimental results

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 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.

Figure 3.7 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 represents 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 each of the leaf nodes is invoked. The prediction results was 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

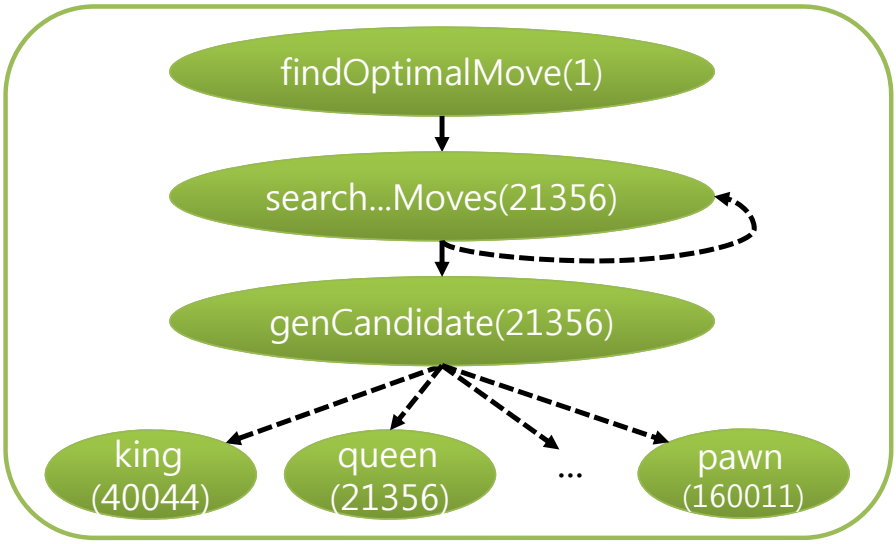


Figure 3.7 A part of the call graph for Chess Engine

Table 3.3 Performance prediction results for Chess Engine

Prediction target	Prediction error (%)	Prediction time (%)	No. of chosen features
exec. time of dev.	15.18	0.03	3
exec. time of serv.	15.22	0.03	3
energy cons. of dev.	15.85	0.03	3
transferred state size	0	0	0
memory usage	4.55	0.25	2

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 3.3 shows the prediction error, prediction time, number of chosen features and generated model for each predictor of an REM, `findOptimalMove`. The "prediction error" column

Table 3.4 Performance prediction models for Chess Engine

Prediction target	Generated model
exec. time of dev.	$c_1 {f_1}^{13} {f_3}^2 + c_2 {f_1}^{11} + c_3 {f_1}^{14} {f_2}^4$
exec. time of serv.	$c_1 {f_1}^{13} {f_3}^2 + c_2 {f_1}^{13} f_2 + c_3 {f_1}^{15} {f_3}^5$
energy cons. of dev.	$c_1 + c_2 f_1 + c_3 {f_2}^2 + c_4 f_1 {f_3}^2 + c_5 f_1 {f_2}^3$
transferred state size	c_1
memory usage	$c_1 + c_2 f_2 2 + c_3 f_1 2 f_2 + c_4 f_1 f_2$

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

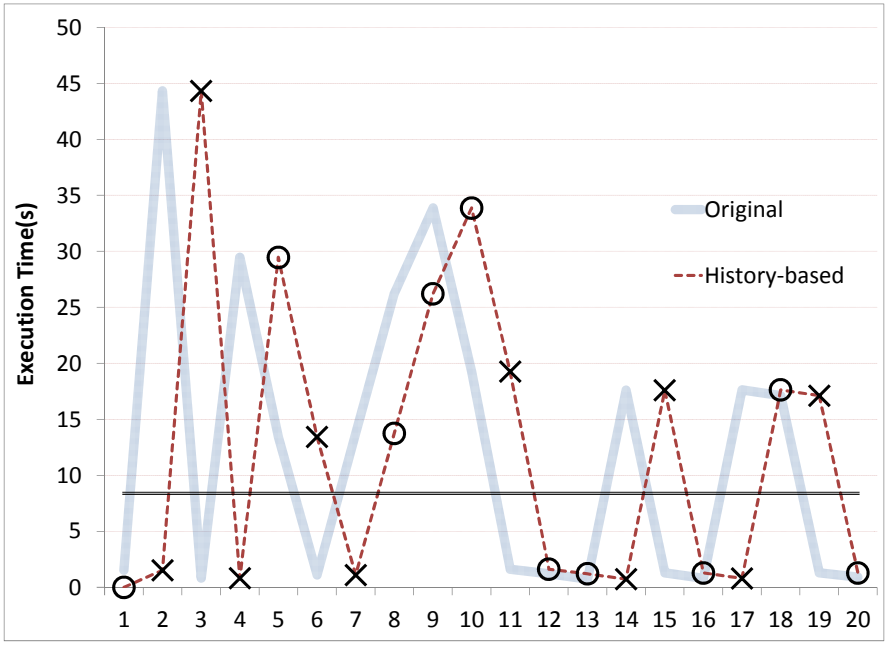


Figure 3.8 The results of code partitioning for Chess Engine with History-based prediction.

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.

Figure 3.8 and 3.9 show 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. On the 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 3.5 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 con-

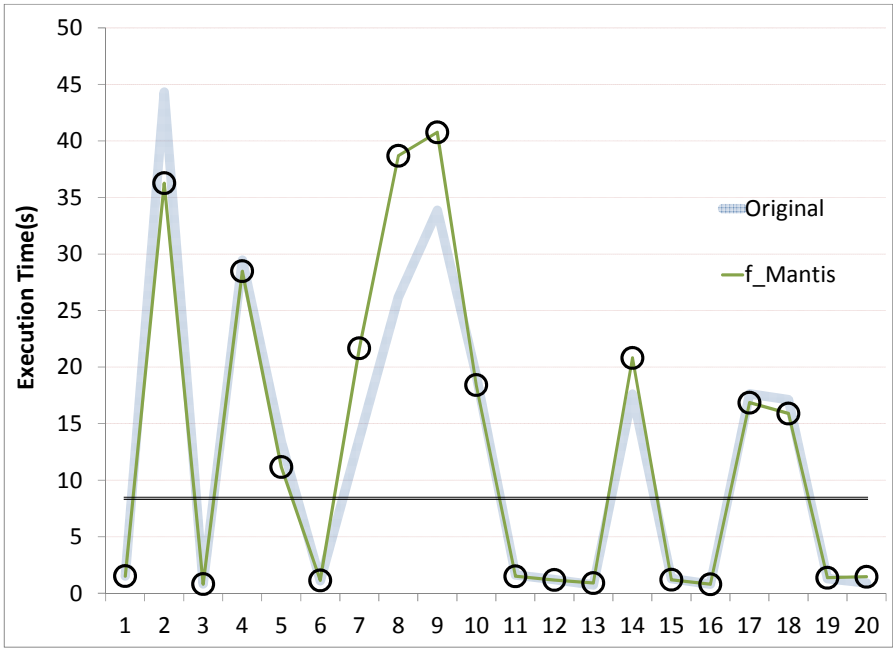


Figure 3.9 The results of code partitioning for Chess Engine with f_Mantis prediction.

sumption 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.

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 3.3 shows the prediction results of memory usage which would be used in

Table 3.5 Offloading results using each prediction techniques for Chess Engine

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

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 1,000 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.

Face Detection

This application detects faces in an image by using the OpenCV library [63]. 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 X 100 and 1,000 X 3,000 pixels for training data. As seen in Table 3.6, 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

Table 3.6 Performance prediction results for Face Detection

Prediction target	Prediction error (%)	Prediction time (%)	No. of chosen features
exec. time of dev.	4.45	0.6	2
exec. time of serv.	4.74	0.6	2
energy cons. of dev.	6.57	0.6	2
transferred state size	0.01	0.6	2

Table 3.7 Performance prediction models for Face Detection

Prediction target	Generated model
exec. time of dev.	$c_1 + c_2 f_1 f_2 + c_3 {f_1}^2 f_2 + c_4 f_1$
exec. time of serv.	$c_1 + c_2 f_1 f_2 + c_3 {f_1}^2 f_2 + c_4 {f_1}^3 f_2 + c_5 {f_1}^2$
energy cons. of dev.	$c_1 + c_2 f_1 f_2 + c_3 f_1 + c_4 {f_1}^2$
transferred state size	$c_1 + c_2 f_3$

jpeg image (f_3) was selected.

Table 3.6 shows the results of performance prediction for the REM. Using the predictor, we run Face Detection to be offload the server with 1,000 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.

Figure 3.10 and 3.11 show the results of code partitioning for twenty inputs with History-based and f_Mantis prediction. In this graph, code partitioning with f_Mantis

Table 3.8 Offloading results using each prediction techniques for Face Detection

	Total execution time(s)	Total energy consumption(mAh)
Oracle	4,346	479
RO	6,157	596
StandAlone	6,077	844
HB	6,108	742
f_Mantis	4,385	483

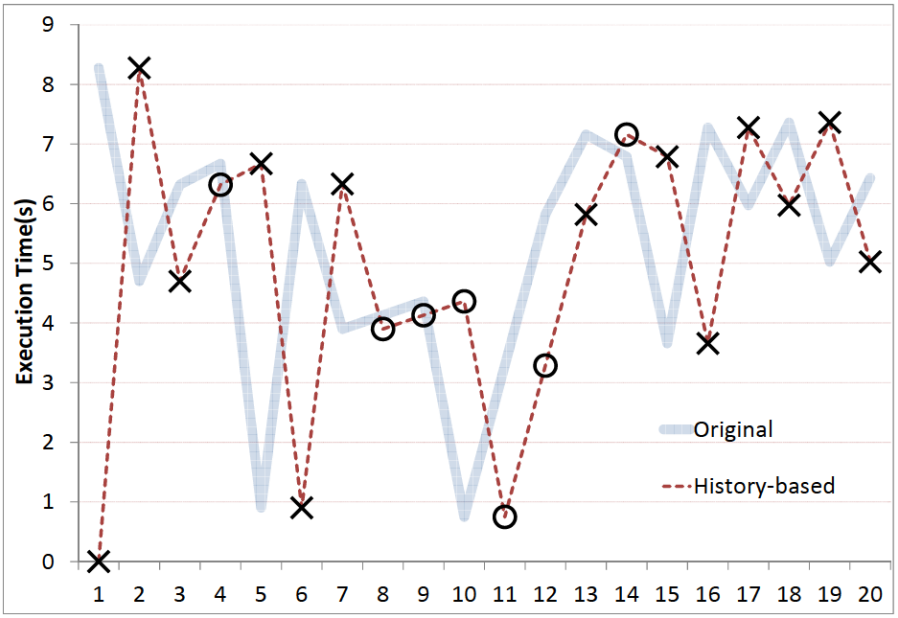


Figure 3.10 The results of code partitioning for Face Detection with History-based prediction.

makes wrong decision only once. However, code partitioning with History-based prediction makes right decisions for under half inputs. As seen in Table 3.8, f_Mantis shows performance that almost matches Oracle, while the performance of offloading with History based prediction shows little improvement over ROE or Standalone

Invaders

This application is a demo 3D game in libGDX [64], 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

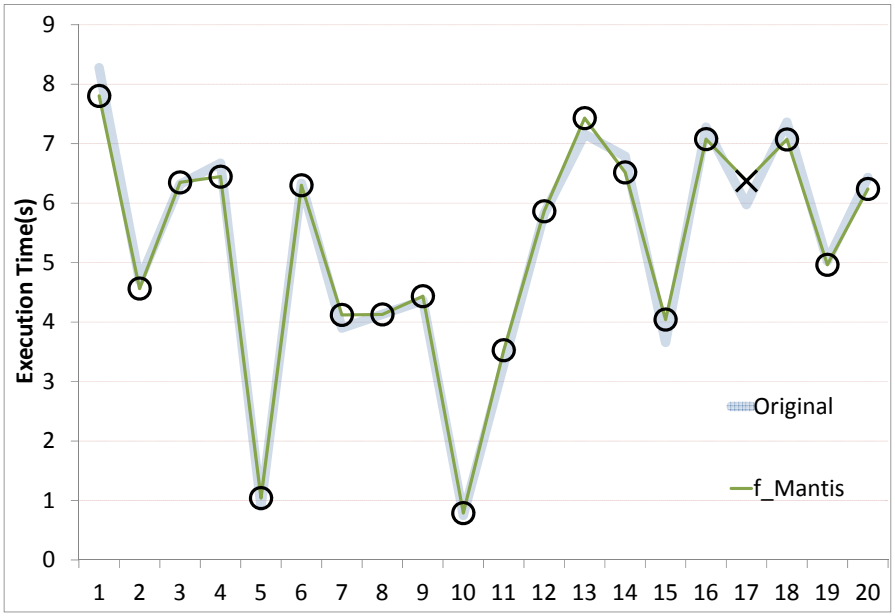


Figure 3.11 The results of code partitioning for Face Detection with f_Mantis prediction.

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 is 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

Table 3.9 Performance prediction results for Invader

Prediction target	Prediction error (%)	Prediction time (%)	No. of chosen features
exec. time of dev.	10.06	0.0	2
exec. time of serv.	5.37	0.0	0
energy cons. of dev.	11.74	0.0	2
transferred state size	0.00	0.0	0

Table 3.10 Performance prediction models for Invader

Prediction target	Generated model
exec. time of dev.	$c_1 + c_2 f_1 + c_3 f_1 f_2$
exec. time of serv.	c_1
energy cons. of dev.	$c_1 + c_2 f_1 + c_3 f + 2 + c_4 f_2 {f_1}^3 + c_5 {f_2}^3 f_1$
transferred state size.	c_1

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 3.9 shows the prediction results of the predictor which 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 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. And as seen in the offloading results of Table 3.9, the predictor proved to be accurate enough.

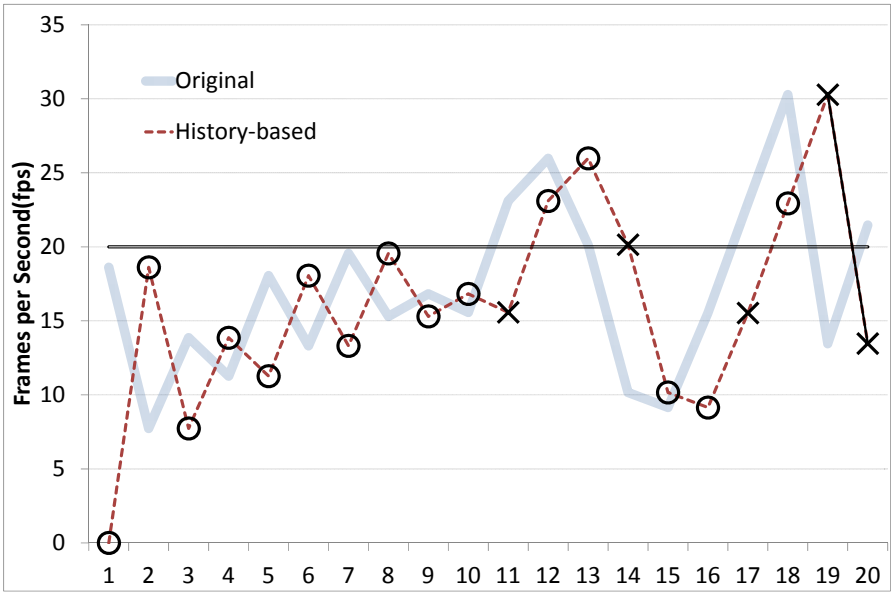


Figure 3.12 The results of code partitioning for Invader with History-based prediction.

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 seconds. Then, in our modified framework, whenever the game state is changed, a prediction would be made to decide whether to render locally or remotely. Figure 3.12 and 3.13 show 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 3.11 shows the results of running the tweaked Invader for 50 seconds 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 seconds, before it could be read to be rendered.

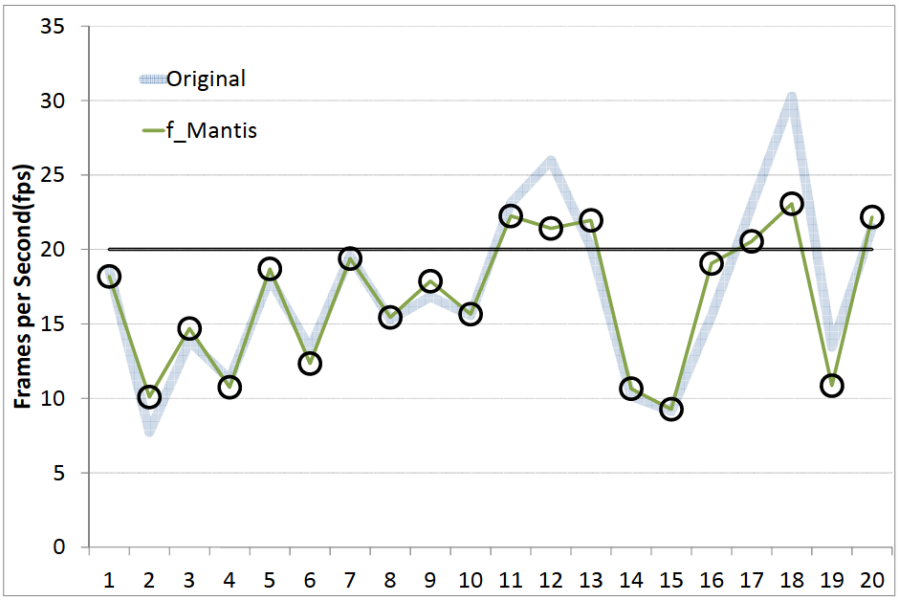


Figure 3.13 The results of code partitioning for Invader with f_Mantis prediction.

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 1,000 state changes. This is because some states require a little bit more time than 0.05 seconds to render even on the sever. Remote only rendered 758 out of 1,000 and stand alone 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.

Table 3.11 Offloading results using each prediction techniques for Invader

	Oracle	RO	StandAlone	HB	f_Mantis
Average FPS(frame)	20.02	20.08	15.71	18.91	19.98
Total energy consumption(uAh)	6,732	7,553	4,630	6,094	6,642
Taken input count	827	758	575	681	808
STDEV	1.33	1.62	3.91	3.00	1.36

3.6 Related work

Much research has been devoted to modeling system behavior as a means of prediction for databases [10, 11], cluster computing [33, 34], networking [35, 36, 37], program optimization [38, 39], 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 [41, 42], database query execution-time and resource prediction [10, 11], performance prediction for streaming applications based on control flow characterization [43], violations of Service-Level Agreements (SLAs) for cloud and web services [33, 34], and load balancing for network monitoring infrastructures [12]. Such work demonstrates significant benefits from prediction, but focuses on problem domains that have identifiable features (e.g., operator counts in database queries, or network packet header values) based on expert knowledge, use domain-specific feature extraction that may not apply to general-purpose programs, or require high correlation between simple features (e.g., input size) and execution time.

Delving further into extraction of non-trivial features, research has explored extracting predictors from execution traces to model program complexity [13], to improve hardware simulation specificity [45, 46], and to find bugs cooperatively [47]. 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 [35, 36, 37].

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 [48, 49, 50, 51, 52] 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 [13], simulation to derive worst-case running time [53], and symbolic execution and abstract evaluation to derive either worst-case inputs for a program [54], or asymptotic bounds on worst-case complexity [55, 56]. 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 [65]. 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. [66, 67] 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. [68] 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 [69] 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. [70] also uses 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 [71] 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. They 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 [72] 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 [73] 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, Odessa periodically acquires information from a low overhead run-time profiler to estimate the bottleneck in the current configuration.

3.7 Conclusion

In this chapter, 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. And 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.

Chapter 4

CMcloud: Cloud Platform for Cost-Effective Offloading of Mobile Applications

4.1 Introduction

In this chapter, we propose *CMcloud*, a novel cost-effective mobile cloud platform, which works nicely under the real-world cloud environments. The key idea of CMcloud is to exploit a novel performance modeling methodology for estimating the target application's post-offload performance accurately on any target server, regardless of its current utilization. At the same time, CMcloud allows to offload as many applications to each server as possible without violating the applications' user-expected performance. If the target performance cannot be achieved using the currently allocated server due to inaccurate performance estimations, CMcloud performs fast inter-server live migrations to achieve the target performance. In this way, CMcloud can offer to users its QoS-guaranteed offload service at a very low price, while minimizing the cloud operation costs.

CMcloud operation assumes the following working environments. First, CMcloud is given the target application's performance profiled on both the user device and a reference-model cloud server. Such static profiling assumption of CMcloud is similar to that of existing offload schemes [6, 7], and thus it does not incur any extra profiling overheads compared to the existing schemes. Second, CMcloud allows to run as many applications on each server as possible to minimize the cloud operation costs.

Based on the environments, CMcloud works as follows. First, on receiving an offload request, CMcloud applies a sophisticated architecture performance modeling to find the most cost-effective target server whose remaining resources are just large enough to achieve the target performance. CMcloud finds the most cost-effective target server by accurately predicting the application's performance by estimating how the performance profiled on the reference server would change on the target server, regardless of its current utilization. Next, CMcloud performs offloading and starts to monitor the application's progress. If CMcloud detects any failure in achieving the target performance due to either inaccurate estimations or unexpected performance contentions, it performs inter-server live migrations to achieve the target offload performance. In this way, CMcloud provides the most cost-effective offloading service to users without violating the QoS of the offloaded applications.

To the best of our knowledge, CMcloud is the first mobile cloud platform to provide the cost-effective offloading service by taking into account the costs of cloud operation and the quality of offload services. Our example implementation on top of a 8-node (16 sockets) Android-x86 / KVM [74] with QEMU 1.4.0 / Ubuntu 12.04 64bit platform shows that CMcloud can improve the server throughput by 111% over a conventional static light-load scheme (or a 2.9x per-socket throughput.) Alternatively, CMcloud reduces the number of service failures by 80% over a static high-load scheme, while even improving the throughput by 16%.

Our work makes the following contributions:

- **Novel design.** We propose CMcloud, a novel cost-effective mobile cloud which exploits a performance modeling theory and inter-server migration capability.
- **High performance.** CMcloud significantly improves the server throughput over the conventional static load schemes (e.g., 2.9x per-socket throughput.)
- **Low costs.** CMcloud maximizes the server throughput or minimizes the server costs, while guaranteeing the user-expected offload performance.
- **Easy applicability.** CMcloud requires only a single reference-machine profiling to find the most cost-effective server, regardless of its current utilization.
- **Strong results.** Our results show that CMcloud can achieve 16% higher throughput over a heavy-load scheme, while reducing 80% of service failures.
- **Low profiling overhead.** CMcloud does not need to profile every application for all possible servers, their load states and inputs.

The rest of this chapter is organized as follows: in Section 4.2, we first explain limitations of existing schemes. Then in Section 4.3, we describe CMcloud's key design goals and its basic operation and architecture model. And in Section 4.4, we describe CMcloud operatino mechanisms in detail. In Section 4.5, we experimentally demonstrate the effectiveness of CMcloud. Finally, in Section 4.6 and 4.7, we relate our work and conclude.

4.2 Backgrounds and Limitations

To motivate our CMcloud platform, this section introduces conventional offload schemes and their key limitations.

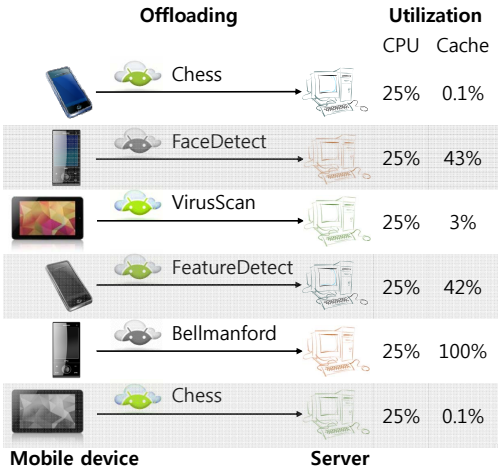


Figure 4.1 Offloading to multiple idle servers.

4.2.1 Basic Offload Mechanisms

The recent seminal works on the mobile-to-cloud offloading [6, 7] propose to run mobile applications on high-performance servers. Even though their detailed implementations can differ based on the code modification scope (e.g., user application, kernel,) and the offload granularity (e.g., functions, threads,) they are generally implemented as follows. First, the cloud provider must have profiled the target application's performance and power consumption on both the mobile device and the target server. Next, on receiving an offload request, the cloud provider compares the application's profiled performance on the mobile device and the target server. If any performance improvement is expected, which is likely to be the case unless the communication latency becomes an obvious bottleneck, the cloud provider offloads the application to the target server, and moves it back to the mobile device after the user-specified execution region is completed.

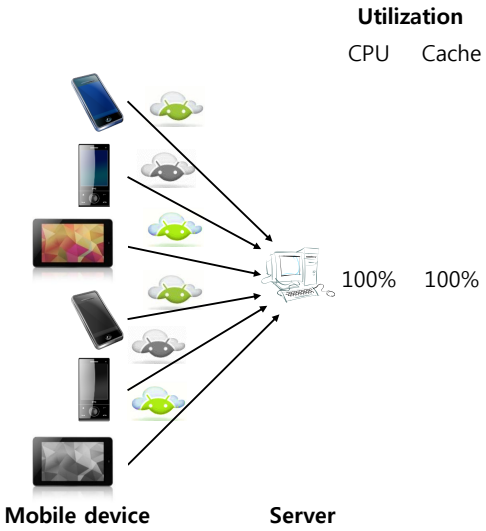


Figure 4.2 Offloading to a busy server

4.2.2 Limitations of Existing schemes

However, as the existing schemes do not consider the user's service purchasing costs nor the cloud provider's server operation costs, they cannot be applied to the real-world cloud environments, where the cloud provider aims to maximize the server throughput or to minimize the server costs and charges the users based on their cloud resource usage.

Costs of offload services. The existing schemes completely ignore the costs of offload service by assuming that servers are provided for free and they run only one mobile application or maintain a same static load per server. Therefore, they always perform offloading as long as any amount of performance improvement is expected, which is likely to be the case because a lightly loaded server is available and runs faster than a mobile device. Figure 4.1 shows a typical scenario in which each four-core server accepts only a single offload request to achieve the highest performance and guarantee the user-expected performance.

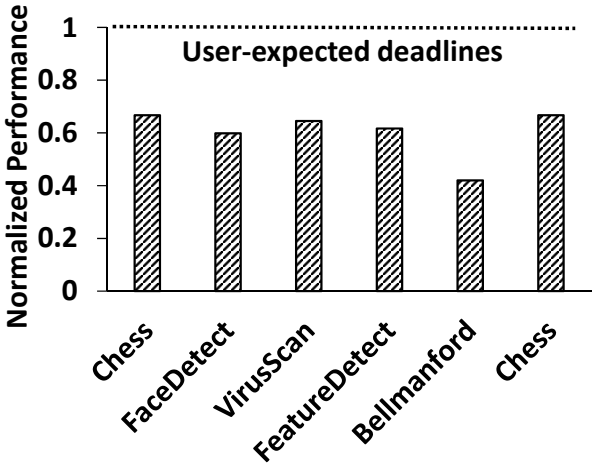


Figure 4.3 QoS failure due to incorrect profiling

However, the real-world clouds are designed to run as many applications as possible on each server to maximize the server's throughput or to minimize the number of active servers [75]. Therefore, if the existing schemes run only a small static load per server, the costs of operating the server and thus the user service fee will be significantly increased, which makes the mobile cloud computing business infeasible. Figure 4.2 shows a scenario in which multiple offload requests are serviced on a single server with a tradeoff between the server utilization and the offload performance.

Costs of service failures. To reduce the costs of operating the cloud and the user service fee, the cloud provider must allow to offload as many mobile applications to each server as possible. However, offloading too many applications to each server incurs a new challenge in guaranteeing the user-expected offload performance because multiple applications come to contend for the sharing server resources such as cores and caches.

We define the number of offloaded applications completing within the user-expected deadline over the number of all offloaded applications as the offload service's quality of service (QoS). It should be noted that even a small QoS violation is unacceptable in

the cloud business, as the users only pay the fee as long as the expected performance is achieved. Figure 4.3 shows a scenario in which five applications in Figure 4.1 are now offloaded to a single four-core server and all applications fail to complete within the user-expected deadlines. In this case, five applications contend for four cores and the last-level cache (LLC) available on the server.

Costs of profiling. The existing schemes assume that performance has been previously profiled for the target server and the offloading always achieves the profiled performance. However, this assumption is broken when an application is now offloaded to a target server which is running other applications to reduce the server costs. To enable an accurate performance estimation, the existing schemes must have profiled for all possible load states of each server. However, it is unrealistic for the cloud provider to statically profile every application for all possible server load states.

4.3 CMcloud offloading

In this section, we first describe CMcloud's key design goals. Next, we present its basic operation model and architecture model consisting of three key components.

4.3.1 Design Goals

CMcloud must satisfy the following design goals to enable a cost-effective offload service. First, CMcloud must target a real-world commercial cloud environment, where servers are highly utilized by running multiple applications per server, Second, the cloud provider must be able to find the most cost-effective target server whose remaining resource is just large enough to achieve the target performance, regardless of its utilization. Finally, once an application is offloaded to the cloud, CMcloud must deliver the user-expected performance by considering the QoS success as a primary requirement.

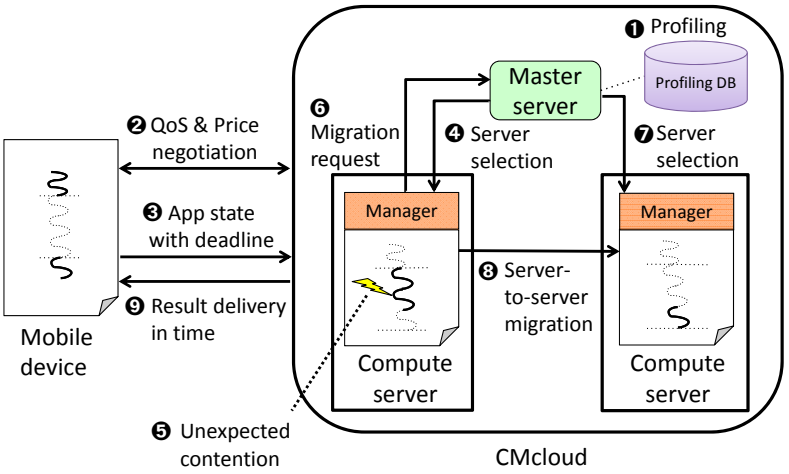


Figure 4.4 CMcloud's example operation model.

4.3.2 Operation Model

Figure 4.4 illustrates how CMcloud performs an offloading once a user agrees to purchase the offload service. Therefore, the cloud provider now has a target deadline for each application to be completed by also considering a variation in the mobile-to-cloud transfer latency. The cloud provider must satisfy the deadline using the minimum server resources.

(1) Profiling on a reference server. CMcloud chooses a reference-model server in the cloud which is used to profile all offload-enabled mobile applications. Any server can be chosen as a reference-model server as long as it is equipped with a basic set of performance counters. CMcloud profiles the application's execution when the reference-model server is idle, and stores the information in the profiling DB. Section 4.4.1 describes the profiling mechanism in more detail.

(2)–(3) Offloading the application. The user requests an offload service, agrees on the service fee, and transfers the application with the termination point and the target deadline. Section 4.3.3 describes CMcloud's mobile-to-cloud offload mechanism in

more detail.

(4) Selecting a target server. The cloud provider finds the most cost-effective server to complete the application within the target deadline using the minimum amount of resources. At this step, CMcloud applies a performance modeling methodology to estimate the application's performance on the target server by differences in server specifications (e.g., clock frequency, cache size) and load states between the reference-model server and the target sever. Section 4.4.2 describes the modeling mechanism in detail.

(5) Detecting a QoS failure. While running the application, the target server monitors the application's progress to detect a potential failure of completing the application within the target deadline, due to either an unexpected performance contention or inaccurate performance estimation. Section 4.4.3 describes the monitoring mechanism in detail.

(6) Migrating to another server. On detecting a potential QoS failure, CMcloud accelerates the application by migrating it to a faster server. Section 4.4.3 describes the performance monitoring mechanism in detail. Section 4.4.4 describes the server-to-server migration mechanism in detail.

(7)–(8) Migration server selection. Similar to the step (2)–(4), the cloud provider selects the best target server based on the cost effectiveness and migrates the application to a new server. The cloud provider can repeat the steps from (5) to (8) to maximize the server throughput, while satisfying the QoS requirement.

(9) Completion. On reaching the offload termination point, the application is migrated back to the mobile device.

As a result, the user always achieves the expected performance for the paid service fee, while preserving the mobile device's battery. At the same time, the cloud provider can increase the server utilization to reduce both the datacenter operation costs and the offload service fee.

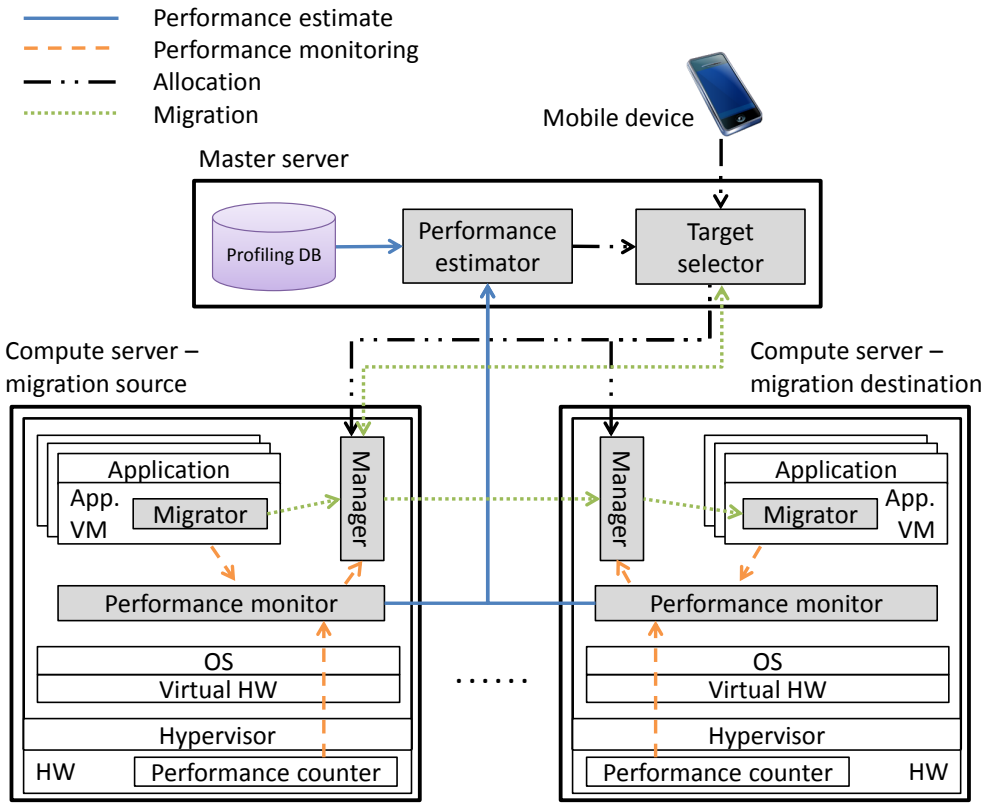


Figure 4.5 CMcloud's basic architecture model.

4.3.3 Architecture Model

In this section, we describe our CMcloud architecture, which consists of a single *master server* and the rest of servers as *compute servers*, as shown in Figure 4.5.

Master server. The master server consists of three components: *profiling DB*, *performance estimator*, and *target selector*. First, the profiling DB contains the profiled execution information on all offload-enable mobile applications on the reference-model server. Next, the performance estimator predicts the application's performance on a current candidate target server analyzing the profiled information on the reference-model server, and differences in server specifications and utilizations between the

reference-model server and the candidate target server. Finally, the target selector finds the most cost-effective target server which will deliver the user-expected performance at the minimum costs.

Compute server. The compute server consists of three components: *manager*, *performance monitor*, and *migrator*. First, the manager communicates with other components and servers by handling requests and replies. Next, the performance monitor measures the application's on-going performance to detect a potential QoS failure (i.e., failing to meet the user-requested deadline) by exploiting the current server's performance counters and the execution profile stored in the profiling DB. Finally, on detecting a potential QoS failure, the migrator embedded in the application virtual machine (VM) suspends the application's execution, migrates its execution state, and continues to execute on a new target server.

Offload-ready mobile device The user's mobile device and operating system must be able to offload a mobile application to the cloud. In this work, we implemented a MAUI-like model as proposed in [6]. For example, the offload handler predetermines offload-enabled regions as remote-executable methods (RM). Therefore, the master server must profile the RM methods and store the profiled information in the profiling DB. Even though we used a MAUI-like model for this work as it does not require to modify the operating system, CMcloud implementation is orthogonal to the mobile-to-cloud offload implementation. CMcloud focuses on providing the cost-effective cloud platform. Therefore, CMcloud can be implemented with other mobile-to-cloud offload models.

Network modeling We modeled 3G and Wi-Fi networks between mobile devices and the cloud using normal distributions of the bandwidth with empirically observed average and deviations. The detailed information is described in Section 4.5.

4.4 CMcloud mechanism

In this section, we describe CMcloud operation mechanisms in detail: reference-server profiling, performance estimation and monitoring, and migration techniques.

4.4.1 Reference-model Server Profiling

The existing offload schemes assume that the offloaded application's performance has been previously profiled for the target server so that they can estimate the application's post-offload performance before making an offload decision. However, if the target server runs different sets of applications from the profiling time, which is the basic operation model of CMcloud, the existing schemes must perform an unbounded number of profiling processes for all kinds of different utilization status even for a single server.

On the other hand, CMcloud still performs a static profiling on a single reference-model server with a few inputs, which can be later translated to the performance for a different target server running any combination of applications. To enable such performance estimation, CMcloud collects the following statistics on the reference-model server using HW performance counters and a memory access tracer.

- **CPI stack.** Execution time breakdown to each performance bottleneck component
- **Feature values.** Values of features characterizing dynamic behaviors of the application on given input
- **Temporal locality information.** Memory access patterns affecting cache hit and miss rates

CMcloud can choose any machine equipped with basic performance counters as a reference-model server. However, as our performance modeling assumes that the

server's pipeline microarchitecture (e.g., branch predictor, issue order) is maintained, CMcloud must profile an application on all reference-model servers representing unique pipeline microarchitecture families (e.g, one reference machine for all Sandy Bridge family processors.) Other than the pipeline structure, CMcloud does not require extra profiling due to different clock speeds or different sizes of last-level caches (LLC). More importantly, CMcloud does not require extra profiling due to different server utilization status.

Therefore, CMcloud's static profiling overhead is much smaller than that of existing offload schemes [6, 7] required to estimating the post-offload performance when servers are highly utilized.

4.4.2 Performance Estimation

Performance Estimation Overview

In figures 4.7, we show CMcloud's performance estimation process. In the off-line stage, CMcloud collects target application's information which is necessary to estimate the application's post-offloaded performance. Then, it generates an *instruction count predictor* with the profiled data. In the on-line stage, CMcloud estimates post-offloaded performance with specification and resource usage of the target server, the profiled data and predicted instruction count.

CPI stack Estimation for Different Servers

(a) Performance Analysis using CPI Stack.

CPI stack [76, 77] is a performance analysis tool widely used to understand how much each performance losing events (e.g., cache miss, branch misprediction) contributes to the overall performance. As cycle-per-instruction (CPI) explains how many cycles are spent to execute a single instruction on average, it is possible to separate the different impacts from different bottlenecks. If the CPU experiences performance los-

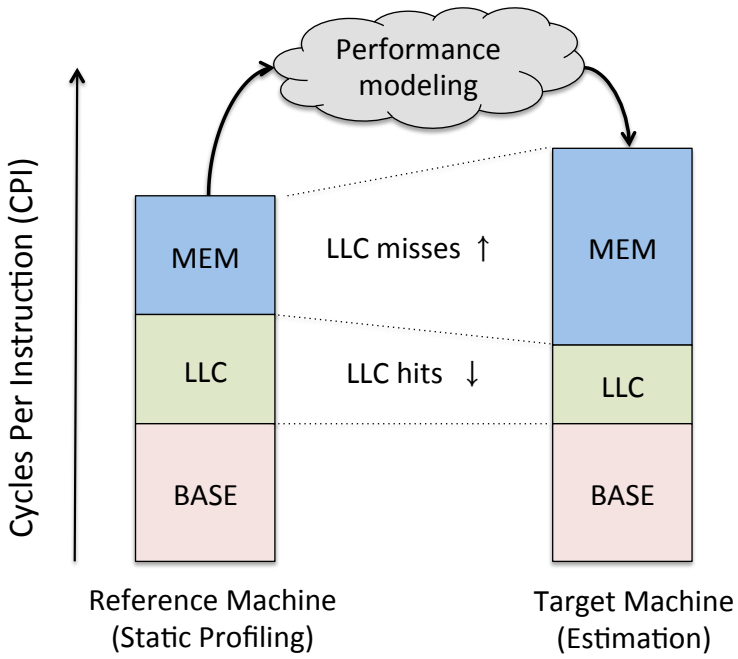


Figure 4.6 Performance estimation using CPI stack

ing events such as a cache miss, the final CPI can be obtained by adding the ideal CPI and the extra CPI caused by the cache miss. Therefore, if we are aware of how each event's CPI impact would change on a target architecture, it is possible to construct a target CPI, as shown in Figure 4.6.

(b) CPI stack estimation for different idle servers.

CMcloud applies the CPI stack method to predict the target application's post-offload performance on a target server, using the profiled performance on the reference-model server. CMcloud first takes the CPI stack collected on the reference-model server, analyzes how key performance losing events will change on a target server, and constructs a new CPI stack to measure the post-offload performance, as shown in Figure 4.7.

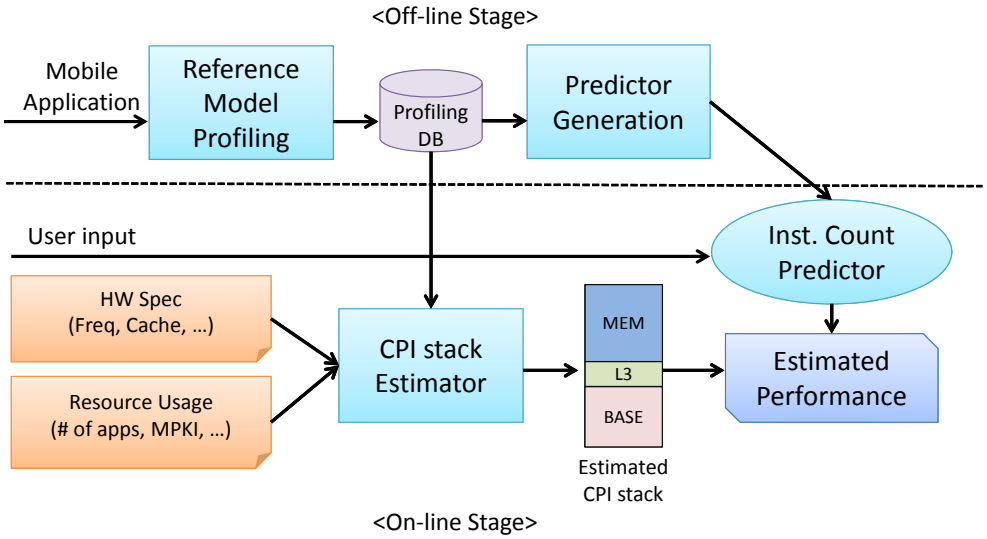


Figure 4.7 CMcloud's Performance estimation process using architecture performance modeling

In this work, CMcloud focuses mainly on four performance impact factors, CPU frequency, LLC hit, LLC miss, and store buffer full, because the number of memory instructions and LLC miss rates affect the overall performance most significantly. Even though we consider only four major performance factors in this work, CMcloud can apply more fine-grain bottleneck components as proposed in [77, 78, 79, 80].

Once such CPI stack becomes available, CMcloud can estimate the performance on a target server by adjusting the impact of each CPI stall event as follows. First, CMcloud breaks the overall CPI down to a combination of four sub-CPI events (i.e., ideal latency ($base$), last-level cache hit (llc), memory access (mem), store buffer full ($sfull$)) as follows.

$$CPI = CPI_{base} + CPI_{llc} + CPI_{mem} + CPI_{sfull} \tag{4.1}$$

Next, CMcloud measures CPI adjusting factors, $CPI_{ratio,mem}$, $CPI_{ratio,llc}$, and $CPI_{ratio,sfull}$. The factors are used for adjusting the corresponding CPI event for the target server.

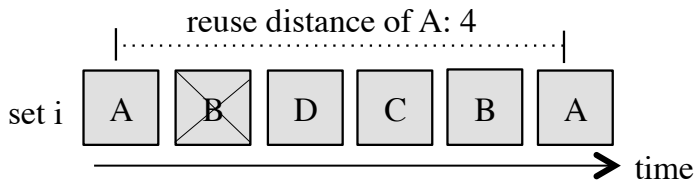


Figure 4.8 An example reuse distance of four for A.

If the CPU clock frequency of the target machine is different from that of the reference server, both CPI_{llc} and CPI_{mem} are scaled for the target CPU. If the target server's memory access latency is different from that of the reference server, the ratio is applied to CPI_{mem} as well.

$$\begin{aligned} Freq_{ratio} &= Freq_{target}/Freq_{ref} \\ CPI_{ratio,llc} &= LLC\_Hit_{ratio} \times Freq_{ratio} \\ CPI_{ratio,mem} &= LLC\_Miss_{ratio} \times Freq_{ratio} \times Lat_{ratio} \end{aligned} \tag{4.2}$$

where $Freq$ is a CPU clock frequency, Lat is a memory access latency, and LLC_Hit and LLC_Miss are the number of LLC hits and misses, respectively.

However, it is difficult to estimate the number of LLC hits and misses, when cache architectures differ between the reference server and the target server. To address the issue, we assume that the size of an LLC differs by the degree of associativity and its cache block replacement policy is based on a LRU policy. In fact, as modern LLCs exploit variations of index hashing mechanisms to effectively increase the degree of associativity, even caches scaled by the number of sets show similar hit and miss patterns as the caches scaled by the associativity.

To discover the application's temporal locality, we leverage the reuse distance (RD) analysis[81], in which RD is the number of distinct and different memory accesses between two consecutive references. Figure 4.8 shows an example reuse distance of

four between two consecutive memory accesses to A. To collect the reuse distances, we use our memory tracing scheme implemented in the QEMU emulator.

With the LLC scaling and the reuse distances available, LLC hit and miss rates can be estimated for differently sized LLCs. For example, when the LLC's associativity increases from x to y, the number of LLC misses decreases by $\sum_{n=x+1}^{y} C_{RD=n}$ where $C_{RD=n}$ is the number of accesses with the reuse distance of n. Therefore, LLC_Miss_{ratio} can be calculated as follows:

$$LLC\_Miss_{ratio} = 1 \pm \sum_{n=x+1}^{y} C_{RD=n}/LLC\_Miss_{ref} \tag{4.3}$$

Finally, the penalty caused by the store buffer full depends on some factors including the issue width (W), in-flight store instructions, memory latency and clock frequency. Frequent LLC misses of store instructions can incur a high penalty by filling up the store buffer, which stalls the entire piepline. We estimate such store buffer full cycles using the measured CPI_{sfull} and average store instructions per cycle. We approximately calculate the changed penalty of store buffer full event as follows.

$$CPI_{ratio,sfull} = 1 + \frac{Freq_{ratio} \times Lat_{ratio} - 1}{W \times \%stores} \tag{4.4}$$

By combining equations for each CPI event, we obtain the final target CPI estimation model for different, but idle target servers:

$$CPI_{target} = CPI_{base} + \sum (CPI_{ratio,event} \times CPI_{event}) \tag{4.5}$$

(c) CPI stack estimation for Different Utilization.

In highly utilized cloud environments, each server is highly utilized to achieve the maximum throughput, and thus it will be difficult to find an idle target server for offloading. If an application is offloaded to a target server currently running other applications, the available CPU clock cycles and LLC capacity will be smaller due to the resource sharing among applications.

To calculate the available clock cycles with a core contention, we simply scale the baseline frequency down by the number of applications. We assume that all applications are evenly scheduled with same priorities. If the operating system applies different priorities, this method can be easily adjusted to consider the relative weights as cycles available.

In addition, to estimate the miss rates of the LLCs experiencing a contention, we exploit the miss rate estimation model as proposed in [82]:

$$LLC\_Misses = C_{RD>A} + \sum_{x=1}^{A} P_{miss}(x) \times C_{RD=x} \tag{4.6}$$

where A is the LLC's associativity, $C_{RD=x}$ is the number of accesses with the reuse distance of x, and $P_{miss}(x)$ is the possibility of miss for the access with the reuse distance of x.

P_{miss} depends on which applications are co-located in the same server. This estimation requires the histogram information such as per-application reuse distances. In our work, as the phase of each application varies over time, we collect the information periodically (e.g, one billion instructions.) Then, we adjust the LLC miss estimation model by considering progresses of background applications in the server where a new application is offloaded.

Once such information becomes available, we apply the modified frequency and miss information to the formulas developed in the Section 4.4.2.(b)

Instruction Count Prediction for Different Inputs

To apply the CPI stack method to predict the target application's post-offload performance on a target server for different inputs, it is necessary to obtain an instruction count for executing the application on each input. For some applications, behaviors of them are not affected by their inputs. In these cases, the value of instruction counts are constant, which can be calculated from profiling data with only an input. The other

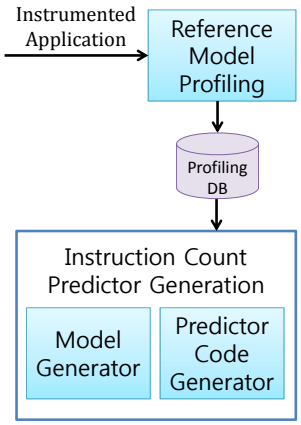


Figure 4.9 Instruction Count Predictor Generatio Overview

applications, however, have dynamic behaviors on input changes. Instruction counts for those applications also can be obtained by profiling for all possible inputs, but It is time-consuming. To overcome the problem, in this work, we chose it Mantis, which we proposed in Chapter 2, to generate a predictor of program performance. Mantis generates a performance predictor for a mobile devices. We modify the techniques of Mantis in order to apply it to instruction count estimation on a server.

Instruction Count Predictor Generation Overview

In Figure 4.9, we show the architecture of our instruction count predictor generation. The architecture takes as input a profiling DB which is computed from an instrumented application. The instrumented application collect the values of features and instruction counts. With the profiling DB, a model generator then performs sparse linear regression on the feature values and the instruction counts, and produces a function that approximates the program's instruction count using a subset of the features. As a final step, a predictor code generator produces a program code which predicts the application's instruction count from its input.

Application Instrumentation and profiling The instruction count 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 feature, 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 instruments 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. After instrumenting, a profiler computes the feature values with test inputs by running the instrumented code.

Model and Predictor Code Generation The profiler produces a large number of features. Among those features, only a handful of features are enough to make an accurate and efficient predictor. To seek compact performance model, which are functions of just a few features that accurately approximates instruction count, we use the SPORE-FoBa algorithm [15]. 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 model generator is used for produce the predictor code generator.

4.4.3 Performance Monitoring

In this section, we describe CMcloud's performance monitoring mechanism. The monitoring mechanism detects the applications' potential QoS failure caused by either an incorrectly estimated post-offload performance or a resource contention in the servers.

Performance Evaluation

The *performance monitor* exploits HW performance counters to check the progress of the target application. Our implementation collects the million instructions per second (MIPS) of each application using a modified version of `perf` [84]. Based on the performance estimation model described in Section 4.4.2, the performance is periodically

measured and compared as the number of retired instructions for the given period (e.g., one second.)

QoS Violation Detection

The performance monitor detects a QoS violation as follows. First, as CMcloud profiles applications only on a single idle reference-model server, the performance monitor estimates the expected performance on the current target server using the model described in Section 4.4.2. We describe the QoS violation detection method as follows:

(1) Estimate instruction count on the input The performance monitor estimates instruction count using the instruction count predictor.

(2) Determine a comparison period. The performance monitor determines a small period of region (e.g., three past seconds) to compare the MIPS. We use few-second comparison periods to tolerate sudden fine-grain performance variations.

(3) Obtain the original completion time. The performance monitor computes the time spent to complete the application based on the originally estimated post-offload performance.

(4) Compute the newly expected completion time. By applying the relative performance difference between the expected post-offload performance and the currently monitored performance, the performance monitor can estimates the application's expected completion time.

(5) Detect a QoS failure. The performance monitor can now detect a potential QoS failure by comparing the newly expected completion time against the target deadline agreed between the user and the cloud provider.

4.4.4 Migration

On detecting a potential QoS failure, CMcloud guarantees the application's QoS requirements by migrating the corresponding applications to a faster server. The *migra-*

tor shown in Figure 4.5 performs a low-cost live VM migration.

Destination selection. On detecting a QoS failure, the migrator must find a right destination server. When a migration request is forwarded to the target selector, the target selector finds a right destination node using the performance estimator, as described in Section 4.4.2. The performance estimator exploits not only the status of each server (e.g., the number of active cores, current server utilizations,) but also the application-specific information (e.g., the number of retired instructions, the elapsed run time.)

Performance overhead. Migration can incur non-trivial performance overhead when the large amount of data is transferred over the network. Therefore, CMcloud performs fast inter-server live migrations to minimize a downtime. We assume that servers already contain key application binaries to avoid migrating binaries.

4.4.5 Cost-aware Application Scheduling in Cloud

To minimize the datacenter operation costs, CMcloud targets to improve server utilizations, while maintaining only a smallest number of active servers in the cloud. To achieve the goal, CMcloud first starts with a small number of nodes and populates the small pool with offloaded applications. Next, on receiving a mobile-to-cloud offload request, the performance estimator collects the estimated performance from the servers. Using this information, the target selector finds the most cost-effective server whose remaining resources are just enough to satisfy the agreed post-offload performance. If the target selector cannot find such server, a new server is activated and added to the current pool of active servers.

4.5 Evaluation

In this section, we first explain our evaluation platform and workloads, and next evaluate CMcloud's accurate performance modeling and its overall cost effectiveness.

Server platform. Our datacenter consists of eight server nodes connected with 10Gbps network, where each server has two CPU sockets. All servers run Ubuntu 12.04 64-bit with Linux Kernel 3.5.0 and KVM [74] with qemu 1.4.0. The KVM release supports both hypervisor and users to access low-level performance counters. To support offloading between mobile phones and x86 servers, we use Android-x86 [85] VMs to run an Android application on a server.

Table 4.1 lists CPU architectures used as reference-model and target servers. We use reference models for different pipeline micro-architecture CPU families (e.g., Nehalem, Sandy Bridge) to avoid inaccurate performance estimation across different micro-architectures. As a result, we use two unique reference CPU models in this work because the target servers use one of the pipeline architectures, but differ in the clock frequency and the cache size.

Network. We modeled a Wi-Fi network using a normal distribution of the bandwidth with empirically obtained 18.5Mbps average and a 3.5 standard deviation. Each

Table 4.1 CPUs used for tests.

	Processor	Frequency	Cache Size
Reference	Intel Core i7-930¶	2.80 GHz	8 MB
	Intel Core i7-2600‡	3.40 GHz	8 MB
Target	Intel Xeon X5650¶	2.66 GHz	12 MB
	Intel Xeon E5-2630‡	2.30 GHz	15 MB
	Intel Xeon E5-2670‡	2.60 GHz	20 MB

¶Nehalem (Westmere) processor
‡Sandy Bridge processor

Table 4.2 Workloads and average performances (in i7-2600).

	Execution time	Total insts	LLC refs/sec	LLC misses/sec
Chess	293 s	1181 B	0.7 M	0.02 M
FaceDetect	37 s	142 B	3.0 M	2.05 M
VirusScan	91 s	495 B	0.7 M	0.69 M
FeatureDetect	21 s	76 B	3.2 M	2.44 M
Bellmanford	173 s	473 B	20.5 M	5.66 M

offload request obtains a unique bandwidth following the distribution. We modeled both 3G and Wi-Fi networks, but used Wi-Fi environments to focus more on the sever-side performance for evaluation. CMcloud can be equally applied to 3G network as well.

Workloads. We implemented five real-world mobile applications listed in Table 4.2. We carefully selected these workloads for the reasonable execution latency, while they contend for the shared resources (e.g., last-level cache.) Chess calculates the latency for a computer to find the next move, FaceDetect and FeatureDetect identify human faces and various features from a given image, VirusScan compares input virus signatures with 1GB of cloud data, and Bellmanford finds a shortest path based on maps which include a real map(NY-city) and randomly generated maps. We generated 1,000 random test inputs for each application. The instruction count predictor for each application is trained on 100 inputs. To support offloading, we modified these workloads as proposed in [6]. We also applied Native Interface (JNI) to evaluate memory-intensive workloads.

Clients. We modeled clients as an inflow of offloading requests based on Poisson distribution with 30 requests per minute for the 24-socket cloud. To finish 30 applications per minute, we configured the application ratio as Chess (19.2%), VirusScan (9.7%), FaceDetect (32.3%), FeatureDetect (32.3%), and Bellmanford (6.5%.)

4.5.1 Estimating Target CPI stack

We first evaluate the accuracy of the proposed CPI stack estimation method using idle target servers by with the reference-model profiling described in Section 4.4.2. We use the estimated CPI stack with the profiled runtime progress information to estimate the performance of CMcloud. Figure 4.10 compares the estimation accuracy between the real performance obtained on the target server and the estimated performance. The x-axis indicates three target-server runs for six workloads, whereas the y-axis shows the

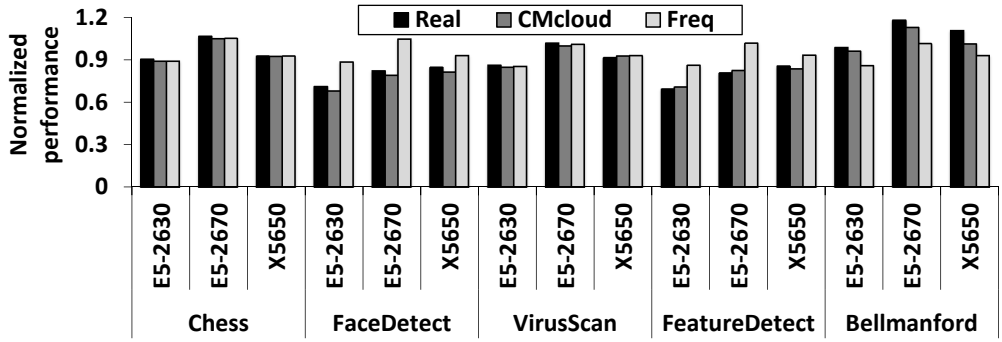


Figure 4.10 Accuracy of the performance prediction for idle server

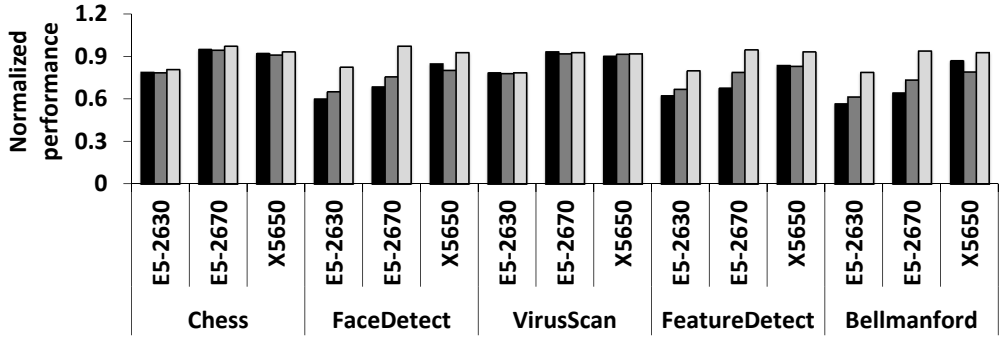


Figure 4.11 Accuracy of the performance prediction for busy server running the five background jobs

performance normalized to the reference machine as shown in Table 4.1. Real bar indicates the actual post-offload performance, while CMcloud bar indicates the predicted performance. The results indicate that CMcloud predicts the performance of idle target servers with the average error of only 2.9%. Freq bar indicates the performance only when the CPU frequency is considered for the estimation, which leads to the average error of 10.3%.

Next, we repeat the same experiments when each target server runs a group of five baseline applications in background. Figure 4.11 indicates that CMcloud's performance estimation is also accurate even for the highly utilized target servers. The results

indicate that CMcloud predicts the performance of busy target servers with the average error of only 5.3%, compared to the 13.4% error of the frequency-only estimation.

4.5.2 Predicting Instruction Count

We evaluate the accuracy and efficiency of instruction count prediction. In Table 4.3, we achieve accuracy with prediction error within 4.3% in all cases, while each predictor runs around 0.4 s. We show the effect of the number of training samples on prediction error in Figure 4.12. In most cases, the curve of their prediction error plateaus before 20 input samples for training.

4.5.3 Cost Effectiveness with QoS requirements

This section evaluates the cost effectiveness of CMcloud by analyzing the improved server throughput and reduced server costs.

Improved server throughput. Figure 4.13 compares the performance and costs of CMcloud against conventional static server allocation schemes. The X-axis lists seven target server allocation schemes: three static allocation schemes and four dynamic allocation schemes including CMcloud. For static allocation schemes, we configured the cloud provider to assign only one application to each socket (17% load,) three applications to each socket (50% load), and five applications to each socket (83% load.) For dynamic allocations schemes, we evaluated a frequency-only estimation model and CMcloud with/without intra-server migration capability. The Y-axis shows, among 500 offload requests, the number of requests successfully completed within the user-

Table 4.3 Prediction error and prediction time.

	Prediction error	Prediction time
Chess	4.30%	0.45 s
FaceDetect	3.88%	0.27 s
VirusScan	0.03%	0.35 s
FeatureDetect	0.97%	0.36 s
Bellmanford	0.33%	0.33 s

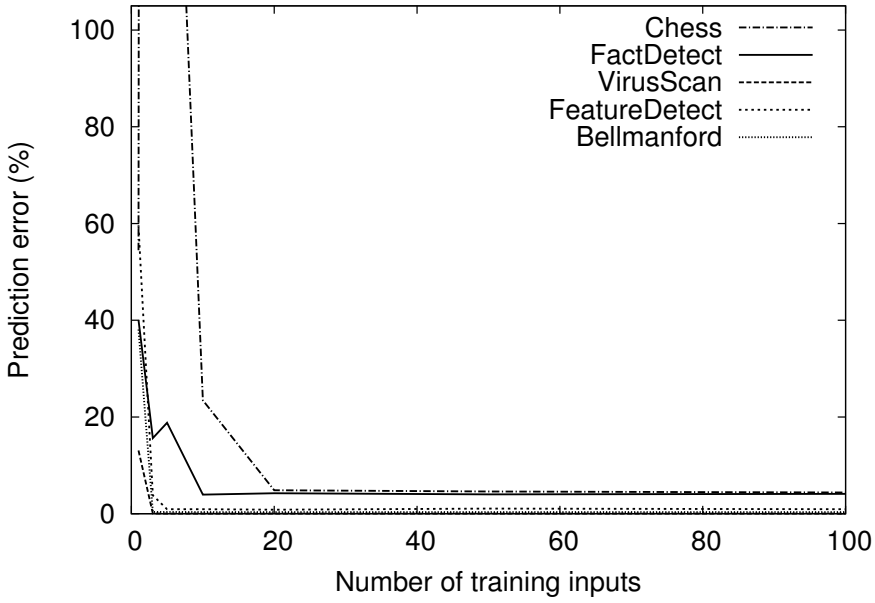


Figure 4.12 Prediction errors for instruction count varying the number of input samples. The y-axis is truncated to 20 for clarity

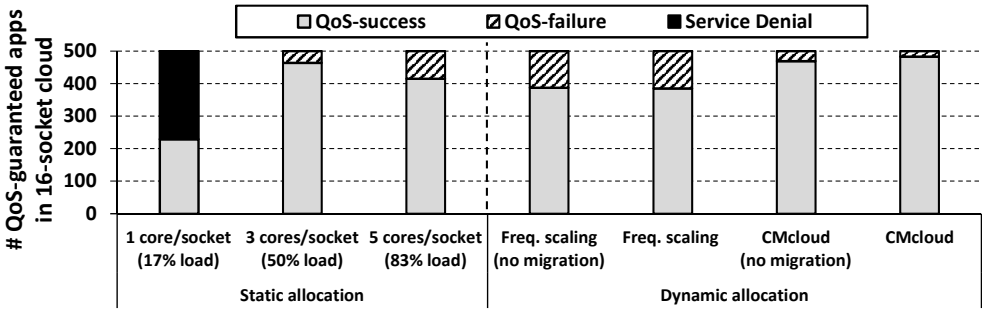


Figure 4.13 Datacenter throughput (out of 500 requests.)

agreed deadline (QoS-success) for the entire cloud, the number of requests violating the deadline (QoS-failure,) and the number of requests turned down by the cloud due to insufficient servers.

Among the static allocation schemes, the 17% load scheme shows the lowest per-

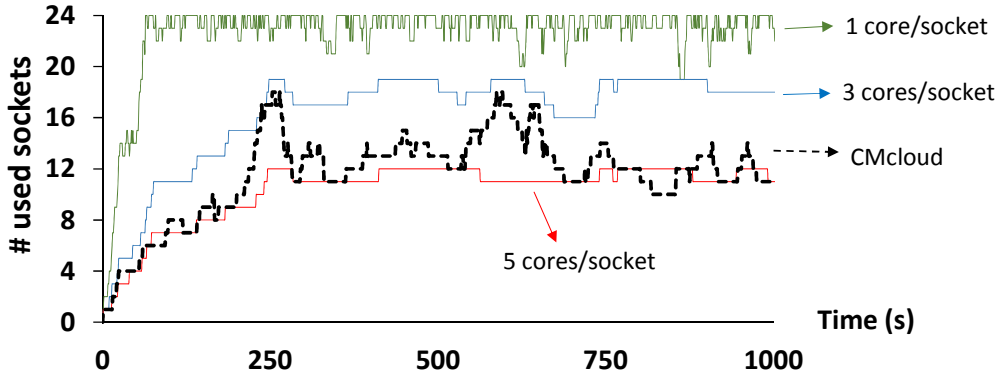


Figure 4.14 Datacenter utilization (out of 16 sockets.)

socket throughput by utilizing only one core per 6-core socket. The 17% load rejects almost half of the requests due to insufficient servers. On the other hand, the 83% load scheme achieves 83% server throughput, while 17% of workloads fail to complete within the deadline. Even though the 50% load shows 93% throughput in return of 50% server efficiency, this sweet spot will change for different workloads. Therefore, considering the server underutilization and the QoS failure are unacceptable for the cloud business, the static allocations cannot be applied as a cost-effective offload scheme.

Among dynamic allocations, CMcloud achieves almost the ideal throughput and even CMcloud without migration capability outperforms two frequency-only estimation models. The result shows that CMcloud improves the server throughput by 111% over the 17% load scheme. Compared to the 83% load scheme, CMcloud reduces the number of service failures by 80%, while even improving the throughput by 16%. The results also show that both the performance modeling and inter-server migration of CMcloud contributed to the improved server throughput separately.

Reduced server costs. Figure 4.14 shows the number of sockets running applications for the first 1000 seconds. In this experiment, we evaluate the server costs

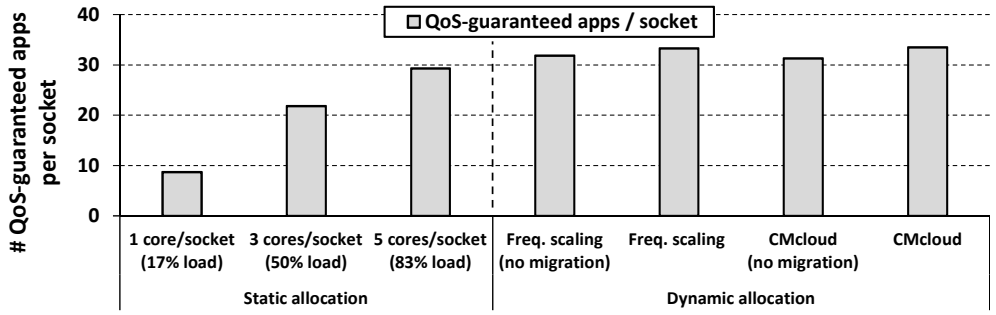


Figure 4.15 Per-socket cost effectiveness.

of CMcloud against three static load schemes. As expected, the higher-load allocation policies utilize a smaller number of sockets than lighter-load allocation policies. However, CMcloud only activates the minimum number of sockets by maximizing the throughput, as long as the QoS of applications is not violated. Considering the CMcloud's high throughput shown in Figure 4.13, it is clearly shown that CMcloud consistently operates at lower costs than the 17% and 50% static allocation schemes.

Cost effectiveness. Considering the improved throughput and reduced server costs of CMcloud, Figure 4.15 compares the cost effectiveness of CMcloud against the static allocation schemes. In this figure, we measure the cost effectiveness of the number of applications successfully completed within the deadline per socket, which indicates each socket's cost effectiveness. The results show that CMcloud outperforms all schemes significantly. CMcloud provides a 2.9x higher per-socket throughput over a static light-load scheme (i.e., 17% load.) It should be noted that the relatively high cost-effectiveness of high-load static allocation policy (i.e., 83% load) comes with many QoS failures. On the other hand, CMcloud does not incur unacceptable QoS failures as shown in Figure 4.13.

4.5.4 Offloading/migration Overhead

Table 4.4 shows the overhead of performance monitoring, inter-server migration, and reference-model profiling. Both monitoring and profiling overheads are normalized to the execution latency without profiling. The monitoring overhead is small and thus shown in percentage.

Once applications are offloaded to servers, CMcloud must monitor all applications to detect the potential QoS violations and trigger server-to-server migrations to improve the performance. We use KVM's native live migration method, which can migrate an application paying only the minimum performance loss. By modifying the KVM's live-migration source code, we measure the latency from when the VM stops at the source node to when it restarts at the destination node. The table 4.4 shows that both monitoring and migration overheads are minimal.

The static profiling can take a long time as it includes the reuse distance analysis obtained by QEMU emulator. However, it is only a one-time overhead paid by the cloud provider and the overhead is not exposed to users. Moreover, CMcloud requires only a single reference-machine profiling, regardless of its current utilization. It should be noted that a similar kind of static profiling is also required by the existing seminal works[6, 7]. Many proposals to reduce the profiling overhead has been proposed, which is orthogonal to our work.

Table 4.4 Offloading overheads.

	Esti.	Moni.	Migr.	Prof.
Chess	1.03%	1.58%	45 ms	x150
FaceDetect	0.51%	0.14%	21 ms	x127
VirusScan	0.18%	4.15%	14 ms	x180
FeatureDetect	0.56%	0.71%	12 ms	x124
Bellmanford	1.34%	3.66%	50 ms	x136

4.6 Related Work

In this section, we discuss previous work related to CMcloud in the areas of dynamic offloading, performance prediction, performance monitoring, and migration.

Dynamic offloading. Dynamic offloading techniques, MAUI [6] and CloneCloud [7], allow users to execute a mobile application on a cloud. However, these schemes are not suitable for the real-world cloud environment due to the lack of the QoS guarantee of applications and a cost model [86, 87]. ThinkAir [88] proposes an on-demand resource allocation for user-side cost and parallel method execution of a mobile application for the QoS guarantee, but focuses on one automatically parallelizable application instead of simultaneous execution of several applications. [89] focuses on reduction in migration overhead by transferring only essential heap objects. Instead, our scheme targets to mobile cloud computing for simultaneous execution of several applications, the QoS guarantee of applications, and minimization of server cost.

Performance prediction. In heterogeneous multi-core systems, PIE [90] and Regression analysis [91] estimate the performance of other cores and assign an appropriate application to an optimal core. These schemes assume that caches have the same size and there is no resource contention. Bubble-Up [92] and Bubble-Flux [93] guarantee QoS of a latency sensitive application. However, the former performs many sensitivity tests with various memory pressures in advance, and the latter does not allow colocation of multiple latency sensitive applications. Our prior work [83] automatically estimate the execution time of the application on various inputs by extracting features related to the performance from a mobile application. To estimate the performance of an application on servers, we apply this technique to predicting the instruction count of the application.

Performance monitoring. Many researches [94, 95] widely use resource monitoring to detect performance interference. Perf [84] and Oprofile [96] monitor the system

resource usage of each application through hardware performance counters. Pin [97] and Valgrind [98] measure what kinds and how many instructions are executed through dynamic instrumentation.

Migrations. Cloud systems migrate VMs to another server for guaranteeing QoS and improve cost effectiveness of clouds. To reduce the downtime of VMs, we adopt Pre-copy [99] as a live migration scheme. We can adopt other live migration schemes [100, 101].

4.7 Conclusions

In this chapter, we proposed CMcloud, a novel cost-effective mobile cloud platform, which works nicely under the real-world cloud environments. CMcloud reduced the cost of offloading by improving the server utilization significantly, while achieving the user-expected offload performance. Our implementation shows that CMcloud can improve the datacenter throughput by 84% over a conventional static light-load scheme (or a 2.7x higher per-socket throughput.) Alternatively, CMcloud reduces the number of service failures by 83% over a static high-load scheme, while even improving the throughput by 31%. To the best of our knowledge, CMcloud is the first cost-effective mobile cloud platform which allows an oversubscribed offloading without affecting the QoS of mobile applications.

Chapter 5

Conculsion

The growing needs of processing pwoer and energy efficiency for new smartphone applications have rendered the existing standalone execution no longer appropriate to satisfy high performance and low power consumption demands. On the other hands, mobile execution offloading is an alternative to the problem. To derive benefit from mobile execution offloadding, it is necessary to predict computational resource consumption of offloaded applications.

In this dissertation, I address the problem of accuracy and efficiency of the prediction. To obtain predictors which are sufficient to use for mobile execution offloading, I propose Mantis, a framework that automatically generates the predictors. We showed that the use of Mantis can predict computational resource consumption with estimation error mostly under 5% by executing slice that spend at most 1.3% of the total execution time of these programs.

From the prediction technique, I propose f_Mantis which is a runtime method-wise performance prediction generator. By utilizing f_Mantis to mobile execution offloading solver, I showed the possibility of it to make more precise offloading decisions. As the

result, we reduce the execution time by up to 31.7% and save the energy of smartphone by up to 57.2%.

To make mobile execution offloading work nicely under the real-world cloud environments, I propose CMcloud, a novel cost-effective mobile cloud platform. CMcloud estimates post-offloaded performance with specification and resource usage of the target server, the profiled data and predicted instruction count. CMcloud reduced the cost of offloading by improving the server utilization, while achieving the user-expected offload performance. The implementation results show that CMcloud can improve the datacenter throughput by 84% over a conventional static light-load scheme. Thus, CMcloud reduces the number of service failures by 83% over a static high-load scheme, while even improving the throughput by 31%.

Bibliography

[1] S. Yang, Y. Kwon, Y. Cho, H. Yi, D. Kwon, J. Youn, and Y. Paek, "Fast dynamic execution offloading for efficient mobile cloud computing," *2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, vol. 0, pp. 20–28, 2013.

[2] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: elastic execution between mobile device and cloud," in *Proc. ACM EuroSys*, 2011, pp. 301–314.

[3] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proc. ACM MobiSys*, 2010, pp. 49–62.

[4] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. IEEE INFOCOM*, 2012, pp. 945–953.

[5] D. Kovachev, T. Yu, and R. Klamma, "Adaptive computation offloading from mobile devices into the cloud," in *Proc. IEEE ISPA*, 2012, pp. 784–791.

[6] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making smartphones last longer with code offload," in *MobiSys*, 2010.

[7] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *EuroSys*, 2011.

[8] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently," in *OSDI*. Berkeley, CA, USA: USENIX Association, 2012, pp. 93–106. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387890

[9] W. Smith, "Prediction services for distributed computing," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, March 2007, pp. 1–10.

[10] C. Gupta, A. Mehta, and U. Dayal, "PQR: Predicting query execution times for autonomous workload management," in *ICAC*, 2008.

[11] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *ICDE*, 2009.

[12] P. Barlet-Ros, G. Iannaccone, J. Sanjuas-Cuxart, D. Amores-Lopez, and J. Sole-Pareta, "Load shedding in network monitoring applications," in *USENIX*, 2007.

[13] S. Goldsmith, A. Aiken, and D. Wilkerson, "Measuring empirical computational complexity," in *FSE*, 2007.

[14] E. Brewer, "High-level optimization via automated statistical modeling," in *PPoPP*, 1995.

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

[16] M. Weiser, "Program slicing," in *ICSE*, 1981.

[17] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, 1995.

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

[19] R. Tibshirani, "Regression shrinkage and selection via the lasso," *J. Royal. Statist. Soc B.*, 1996.

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

[21] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least angle regression," *Annals of Statistics*, vol. 32, no. 2, 2002.

[22] S.-J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky, "An interior-point method for large-scale l1-regularized least squares," *IEEE J-STSP*, vol. 1, no. 4, 2007.

[23] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *COMPSTAT*, 2010.

[24] T. W. Reps, S. Horwitz, S. Sagiv, and G. Rosay, "Speeding up slicing," in *FSE*, 1994.

[25] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *PLDI*, 1988.

[26] O. Lhoták, "Program Analysis using Binary Decision Diagrams," Ph.D. dissertation, School of Computer Science, McGill University, 2006.

[27] M. Sridharan, S. Fink, and R. Bodik, "Thin slicing," in *PLDI*, 2007.

[28] Javassist, www.csg.is.titech.ac.jp/ chiba/javassist, 2012, product page.

[29] Octave, www.gnu.org/software/octave, 2013, product page.

[30] JChord, code.google.com/p/jchord, 2012, product page.

[31] Jasmin, jasmin.sourceforge.net, 2004, product page.

[32] PowerMonitor, http://www.msoon.com/LabEquipment/PowerMonitor/, 2012, product page.

[33] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," in *HotCloud*, 2009.

[34] P. Shivam, S. Babu, and J. S. Chase, "Learning application models for utility resource planning," in *ICAC*, 2006.

[35] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang, "WebProphet: Automating performance prediction for web services," in *NSDI*, 2010.

[36] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, "Answering what-if deployment and configuration questions with wise," in *SIGCOMM*, 2008.

[37] S. Chen, K. Joshi, M. A. Hiltunen, W. H. Sanders, and R. D. Schlichting, "Link gradients: Predicting the impact of network latency on multitier applications," in *INFOCOM*, 2009.

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

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

[40] 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

[41] K. Vaswani, M. Thazhuthaveetil, Y. Srikant, and P. Joseph, "Microarchitecture sensitive empirical models for compiler optimizations," in *CGO*, 2007.

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

[43] F. Aleen, M. Sharif, and S. Pande, "Input-driven dynamic execution behavior prediction of streaming applications," in *PPoPP*, 2010.

[44] Y. Jiang, E. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao, "Exploiting statistical correlations for proactive prediction of program behaviors," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 248–256. [Online]. Available: http://doi.acm.org/10.1145/1772954.1772989

[45] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *PACT*, 2001.

[46] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS*, 2002.

[47] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *PLDI*, 2005.

[48] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," in *RTSS*, 2006.

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

[50] R. Wilhelm, "Determining bounds on execution times," *Handbook on Embedded Systems*, 2005.

[51] S. Seshia and A. Rakhlin, "Game-theoretic timing analysis," in *ICCAD*, 2008.

[52] S. Seshia and A. Rakhli, "Quantitative analysis of systems using game-theoretic learning," *ACM TECS*, 2010.

[53] R. Rugina and K. E. Schauser, "Predicting the running times of parallel programs by simulation," in *IPPS/SPDP*, 1998.

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

[55] B. Gulavani and S. Gulwani, "A numerical abstract domain based on expression abstraction and max operator with application in timing analysis," in *CAV*, 2008.

[56] S. Gulwani, K. Mehra, and T. Chilimbi, "SPEED: Precise and efficient static estimation of program computational complexity," in *POPL*, 2009.

[57] A. Matsunaga and J. A. B. Fortes, "On the use of machine learning to predict the time and resources consumed by applications," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*,

ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 495–504. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2010.98

[58] H. Leather, E. Bonilla, and M. O'Boyle, "Automatic feature generation for machine learning based optimizing compilation," in *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, March 2009, pp. 81–91.

[59] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, Dec 2009, pp. 45–55.

[60] J. K. J. K. J. Y. S. C. Y. K. Y. P. Y. Chae, D. Kim, "Cmcloud: Cloud platform for cost-effective offloading of mobile applications," in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014.

[61] AspectJ, https://eclipse.org/aspectj/, 2005, product page.

[62] Kyro, code.google.com/p/kryo/, 2005, product page.

[63] OpenCV library, http://sourcefouge.net/projects/opencvlibrary/, 2005, product page.

[64] libgdx, http://libgdx.badlogicgames.com/, 2005, product page.

[65] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek, "Mantis: Automatic performance prediction for smartphone applications," in *Presented as part of the 2013 USENIX Annual Technical Conference*. San Jose, CA: USENIX, 2013, pp. 297–308. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/papers/kwon

[66] 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, and H. A. Lagar-Cavilla, "Pervasive Personal Computing in an Internet Suspend/Resume System," *IEEE Internet Computing*, vol. 11, no. 2, pp. 16–25, 2007.

[67] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based Cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14 –23, 2009.

[68] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, "Wishbone: profile-based partitioning for sensornet applications," in *Proc. USENIX NSDI*, 2009, pp. 395–408.

[69] R. Kemp, N. Palmer, T. Kielmann, and H. E. Bal, "Cuckoo: A computation offloading framework for smartphones," in *Proc. MobiCASE*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 76. Springer, 2010, pp. 59–79. [Online]. Available: http://dblp.uni-trier.de/db/conf/mobicase/mobicase2010.html

[70] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: enabling mobile phones as interfaces to cloud applications," in *Proc. ACM/IFIP/USENIX Middleware*, 2009, pp. 83–102.

[71] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, "Adaptive offloading inference for delivering applications in pervasive computing environments," in *Proc. IEEE PerCom*, 2003, pp. 107–114.

[72] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *Proc. ACM OSDI*.

Berkeley, CA, USA: USENIX Association, 2012, pp. 93–106. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387890

[73] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *Proc. ACM MobiSys*, 2011, pp. 43–56.

[74] A. Kivity, Y. Kama, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in *Proceedings of the Linux Symposium*, 2007, pp. 225–230.

[75] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu, "VMware Distributed Resource Management: Design, Implementation, and Lessons Learned," *VMware Technical Journal*, pp. 45–64, 2012.

[76] P. G. Emma., "Understanding some simple processor-performance limits," in *IBM journal of Research and Development*, May 1997.

[77] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," in *ASPLOS '06*.

[78] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, "Interaction cost and shotgun profiling," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 1, no. 3, pp. 272–304, Sep. 2004.

[79] "Intel 64 and ia-32 architectures optimization reference manual," no. 248966-026, April 2012.

[80] Q. Liang, "Performance monitor counter data analysis using counter analyzer," in *IBM developerWorks*, Feb 2009.

[81] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *PLDI '03*.

[82] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, ser. HPCA '05, 2005.

[83] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek, "Mantis: Automatic Performance Prediction for Smartphone Applications," in *ATC' 13*.

[84] A. C. de Melo, "The new linux 'perf' tools," Slides from Linux Kongress, http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf, 2005.

[85] "Android-x86," http://www.android-x86.org.

[86] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, 2011.

[87] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 84–106, Jan. 2013.

[88] S. Kosta, A. Aucinas, and R. Mortier, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *INFOCOM '12*.

[89] S. Yang, Y. Kwon, Y. Cho, H. Yi, D. Kwon, J. Youn, and Y. Paek, "Fast Dynamic Execution Offloading for Efficient Mobile Cloud Computing," in *PerCom '13*.

[90] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE)," in *ISCA '12*.

[91] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite Cores: Pushing Heterogeneity into a Core," in *MICRO '12*.

[92] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Colocations Categories and Subject Descriptors," in *MICRO '11*.

[93] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ISCA '13*.

[94] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "Contention aware execution: online contention detection and response," in *CGO '10*.

[95] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An Analysis of Performance Interference Effects in Virtual Environments," in *ISPASS '07*.

[96] J. Levon and P. Elie, "Oprofile: A system profiler for linux," http://oprofile.sf.net, 2004.

[97] C.-k. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, S. Wallace, V. Janapa, and G. Lowney, "Pin: Building Customized Program Analysis Tools," in *PLDI '05*.

[98] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *PLDI '07*.

[99] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *NSDI '05*.

[100] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *VEE '09*.

[101] J. Kim, D. Chae, J. Kim, and J. Kim, “Guide-copy: Fast and silent migration of virtual machine for datacenters,” in *SC ’13*.

초록

높은 스마트폰의 성능 요구를 만족시키기 위해서 클라우드의 자원을 활용하여 스마트폰 성능을 높일 수 있는 모바일 클라우드 컴퓨팅 기술들이 제안되어 왔다. 그러한 기술들 중 오프로딩 기술은 모바일 기기와 서버 사이에서 쓰레드를 이전 시킴으로써 모바일에서 수행할 코드를 서버에서 대신 수행하고 돌아오게 된다. 모바일 오프로딩 기술에서는 어떤 영역의 코드를 서버로 이전 시킬지의 결정을 동적으로 결정하게 된다. 최적화된 결정을 위해서는 오프로딩의 득과 실을 정확하게 예측해야 할 필요성이 있지만, 이전의 연구들은 정확한 예측을 하는데 필요한 자원 손실 때문에 시도 하지 못했다. 특히, 기존 연구들은 클라우드의 자원에 대한 비용을 무시하였다. 그들은 서버가 항상 휴면상태이고 항상 무료로 이용가능하다고 가정하였다. 그러하여 기존 연구들은 적은 비용으로 최고의 성능을 이끌어 내며 사용자들에게 과금을 하는 실제의 상용 클라우드에는 적용하지 못한다.

따라서 본 연구에서는, 프로그램의 성능을 정확하고 효율적이며 자동적으로 예측할 수 있는 Mantis를 제안하였다. Mantis는 프로그램 분석 기술과 머신러닝 기술을 융합하여 많은 프로그램 Feature들 중 프로그램의 성능과 밀접한 관계가 있는 Feature만을 이용하여 예측 모델을 생성한다. Program Slicing 기법은 효율적으로 Feature 값을 추출하여 생성된 모델을 계산하는 코드를 자동적으로 생성한다. 본 논문에서는 Mantis를 이용하여 프로그램의 성능을 예측하여 오프로딩 성능을 개선할 수 있음을 실험으로 보였다. 또한 효율적인 클라우드 오프로딩 플랫폼인 CMcloud를 제안하여 서버의 운영 비용을 줄이는 동시에 사용자의 요구 성능을 최대한 만족시켜 줄 수 있었다.

주요어: 모바일 클라우드 컴퓨팅, 스마트폰, 성능 예측, 모바일 오프로딩
학번: 2010-30209